FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Master Thesis

Integration and Evaluation of Kubernetes and Docker swarm in combination of Docker Compose

Submitted by: Tanzeem Md Zillu
First examiner: Professor Dr. Armin Lehmann
Second examiner: Professor Dr. -Ing. Ulrich Trick
Date of start: 06.04.2017
Date of submission: 05.09.2017

Statement

Content

1 Introduction 5

2 Theoretical Background 6

2.1 Container Technology 6
2.1.1 History of Container Technology 7
2.1.2 Difference between Containers and Virtual Machines 8

2.2 Docker Container 9
2.2.1 Introduction 9
2.2.2 Docker Architecture 9
2.2.3 The Underlying Technology of Docker 11

2.3 Docker Compose 12

2.4 Cloud Computing 12

2.5 Orchestrator 12

2.6 Application of Orchestrator in Cloud Computing 13

2.7 Container in Cloud Computing 13

2.8 Container Orchestrator 13
2.8.1 General Container Orchestrator Architecture 14

2.9 Docker Swarm 14
2.9.1 Docker Swarm Definition 14
2.9.2 Docker Swarm Architecture 15
2.9.3 Swarm Mode Key Concepts 17
2.9.4 How Swarm Mode Works 17

2.10 Kubernetes 21
2.10.1 Kubernetes Architecture 21
2.10.2 Kubernetes Key Concepts 23

2.11 Summary of Theoretical Background 25

3 Requirements Analysis 26

3.1 General Objectives 26

3.2 Clarifying the Requirements 26

3.3 Time frames 28

3.4 Target State 29
3.4.1 Docker Swarm Implementation 29
3.4.2 Kubernetes Implementation 32

3.5 Use Cases for the Prototype 34

3.6 Software Requirements 40

4 Realization 41

4.1 Prepare Environment 41
4.1.1 Software tools installation 41

4.1.2 Create Virtual Machine 43
4.2 Network Configuration for Cluster 48
4.3 Setting up Docker swarm cluster 50
4.4 Setting up Kubernetes cluster 52
4.5 Description of the Containers 54
4.5.1 A web application printing the container/pod id 54
4.5.2 Web server based on Apache tomcat 56
4.5.3 Database server based on MySQL 56
4.5.4 Prime number counter which counts all prime number between 1 million 56
4.6 Deployment of Application in Docker Swarm Cluster 58
4.6.1 Deploy web application printing the container/pod id 58
4.6.2 Deploy Tomcat Container in Docker Swarm Cluster 59
4.6.3 Deploy MySQL Container in Docker Swarm Cluster 60
4.7 Deployment of Application in Kubernetes Cluster 60
4.7.1 Deploy web application printing the container/pod id 61
4.7.2 Deploy Tomcat Container in Kubernetes Cluster 63
4.7.3 Deploy MySQL Container in Kubernetes Cluster 66
4.8 Auto Scaling in Docker Swarm Cluster 69
4.9 Auto Scaling in Kubernetes Cluster 70
4.10 Load Balancing in Docker Swarm 72
4.11 Load Balancing in Kubernetes 73
4.12 Container Update and Rollback in Docker Swarm 75
4.13 Container Update and Rollback in Kubernetes 77
4.14 Live Migration 80
4.15 Performance Analysis 80
4.16 Rancher 86
4.16.1 Docker Swarm in Rancher 87
4.16.2 Kubernetes in Rancher 93

5 Summary and Perspectives 100
5.1 Summary 100
5.2 Future Scope and Conclusion 101

6 Abbreviations 102

7 References 104

8 Appendix 107

1 Introduction

The main purpose of this master thesis is to build two different clusters based on Docker swarm and Kubernetes respectively and evaluate the performance of both clusters based on some defined use cases. In the theoretical part of this thesis work will cover the container technology, Docker container, Docker compose, container orchestrator, Docker swarm, Kubernetes etc. The thesis implementation will cover the cluster setup for both Docker swarm and Kubernetes, necessary networking for cluster, application deployment, scaling, load balancing, container update and rollback, container live migration and performance analysis. Finally, in this thesis there is an evaluation between Docker swarm and Kubernetes based on the results from all experiments

The names of the main chapters of this thesis documentation are in below:

1. Introduction
2. Theoretical Background
3. Requirements Analysis
4. Realisation
5. Summary and Future Perspectives
6. Abbreviations
7. References

2 Theoretical Background

This chapter explains the all theoretical background of all technologies which used in this thesis work. To understand the implementation of this research work it is necessary to do the theoretical discussion from all perspective which are related with the technologies. Therefore, it would be useful and necessary to understand the theories, architectures, protocol and software which has used in this research work.

This chapter contains the following major sections

- Container technology
- Docker Container
- Cloud computing
- Orchestrator
- Application of orchestrator in cloud computing
- Containers in Cloud Computing
- Orchestrator for containers
- Docker swarm
- Kubernetes
- Summary of theoretical background

In every section, there are several sub sections which covers the theoretical background and relation of that technology with this work. Overall this chapter is structured in a way that the mentioned ten major section will cover all the theories which is necessary to understand this work and enlighten the relation with the implementation.

2.1 Container Technology

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment. Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure. (docker, 2017)

The following figure 2.1 will demonstrate the general architecture of container

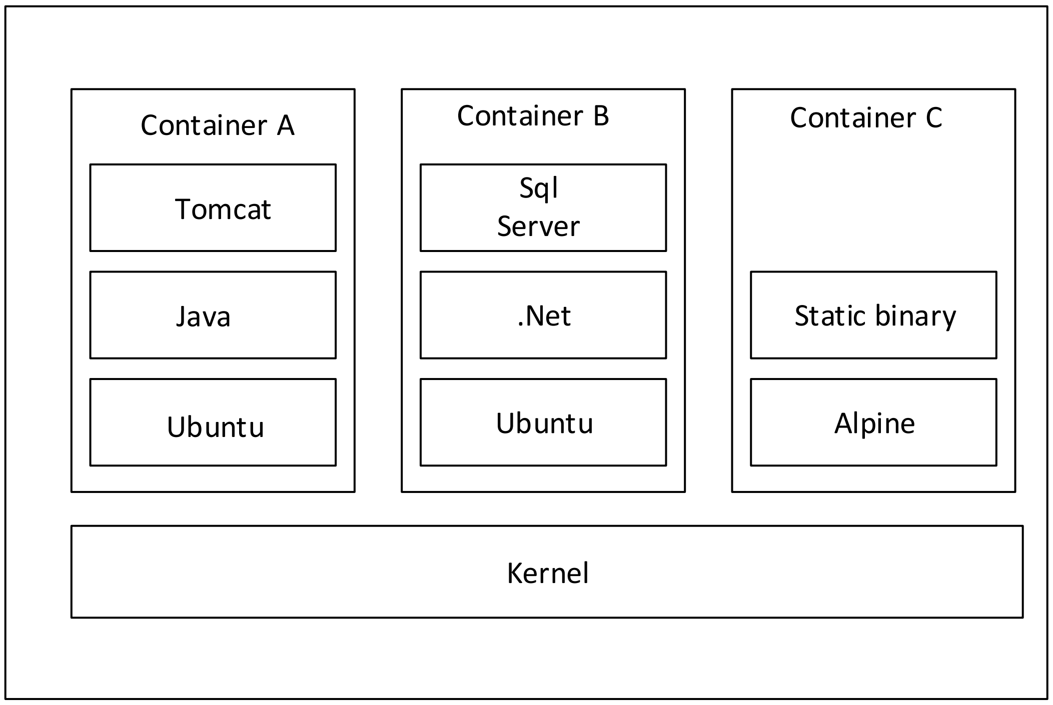


Figure 2.1: Container Architecture (What is a container,2017)

From figure 2.1 it is seen that in one single host there are three containers running. The containers are tomcat, sql-server and alpine respectively. Each container contains the necessary environment variable inside. So, it is not necessary to have the all environment variable before in a host to run the application. Container itself will create the environment to run the application.

So as a basic definition, containers could also be called operating system virtualization where the virtual layer runs as an application within one OS. When combined with a cloud platform, user do not have to worry about investing in multiple servers and other infrastructure to run your applications.

2.1.1 History of Container Technology

The earliest versions of container technology date back to 1979. Unix V7 usually gets the credit for the earliest representation through chroot system call, creating early process isolation. The Unix operating system was going strong then, though it quickly went more commercial when AT&T bought them out in the early 1980s. Many consider Unix V7 the last true Unix of the era, though that early formation of container technology went stagnant for a while. Not until 2000 did containers become a major technology. At this point, FreeBSD jails came to prominence via Derrick T. Woolworth at R&D Associates. Woolworth created the basic technologies behind containers, yet was still quite rudimentary.

In 2001, Linux VServer took from the above jail system and enhanced it by securely partitioning resources on any computer system. Linux kicked off a long wave of other container technologies, including Solaris containers in 2004. Open VZ was yet another in this container string and somewhat like Solaris.

Google got into the container business by 2006 with their Process Containers. One year later, they renamed this Control Groups and added to the Linux Kernel.

Throughout the 2010s, container companies like LXC, Warden, and LMCTFY try to provide containers to the masses. Not until 2013, however, did containers finally start to become popular through Docker.

Docker is the most popular container technology now a day. It started in 2013 as a PaaS called dotCloud, then renamed to its current brand name. (Kelly, 2016)

2.1.2 Difference between Containers and Virtual Machines

From the previous discussion it is clear that Container is one of the most popular virtualization technology now. But there are other virtualization techniques which are widely used such as virtual machines. But there are some basic difference in between virtual machines and containers. In below it is widely discussed

Containers and virtual machines are two ways to deploy multiple, isolated services on a single platform. They both provide a way to isolate applications and provide a virtual platform for application to run on. This concept is explained in below figure 2.2

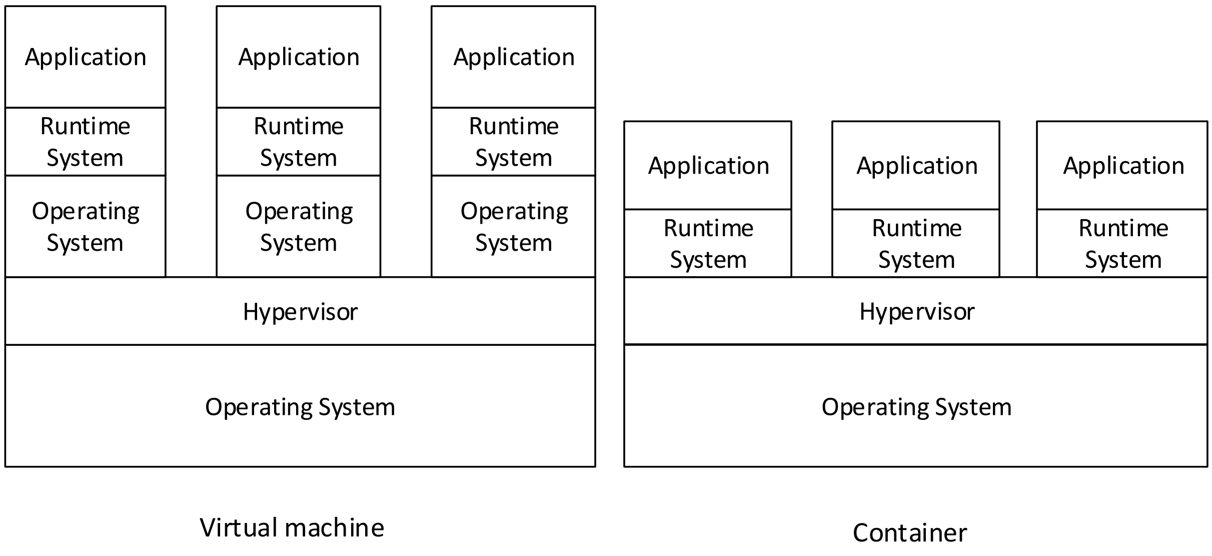


Figure 2.2: Container and Virtual machine architecture comparison (kelly, 2016)

Virtual machines (VM) are managed by a hypervisor and utilize VM hardware, while container systems provide operating system services from the underlying host and isolate the applications using virtual-memory hardware. In a nutshell, a VM provides an abstract machine that uses device drivers targeting the abstract machine, while a container provides an abstract OS. A para-virtualized VM environment provides an abstract hardware abstraction layer (HAL) that requires HAL-specific device drivers. Applications running in a container environment share an underlying operating system, while VM systems can run different operating systems. Typically, a VM will host multiple applications whose mix may change over time versus a container that will normally have a single application. However, it's possible to have a fixed set of applications in a single container.

Containers provide many advantages over VMs, although some can be addressed using other techniques. One advantage is the low overhead of containers and, therefore, the ability to start new containers quickly. This is because starting the underlying OS in a VM takes time, memory, and the space needed for the VM disk storage. Normally, a VM needs at least one unique image file for every running instance of a VM. It contains the OS and often the application code and data as well. Much of this is common among similar VMs. In the case of a raw image, a complete copy of the file is needed for each instance. This could require copying multiple gigabytes per instance. an initial instance of the VM is set up and the operating system is installed possibly with additional applications. On the other hand, container does not need a host operating system to initialize the application. So, the overhead is less in container than a VM (Kelly,2016). Because of this it is possible to run more containers than VM in a specific hardware.

In this thesis work, Docker container has been used so, all the further discussion is based on Docker container.

2.2 Docker Container

2.2.1 Introduction

Docker is one of world's leading software container platform. Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers. Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density. Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux and Windows Server apps.

Docker is an open platform for developing, shipping, and running applications. Docker enables user to separate applications from infrastructure so user can deliver software quickly. With Docker, user can manage his/her infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, user can significantly reduce the delay between writing code and running it in production. The isolation and security allow user to run many containers simultaneously on a given host (Docker overview, 2017). As containers are very light weighted so user can run many applications on a single host. Docker provides tools and platform to manage the lifecycle of containers (Docker overview, 2017).

2.2.2 Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface (Docker overview, 2017). More explanation about all components are in below

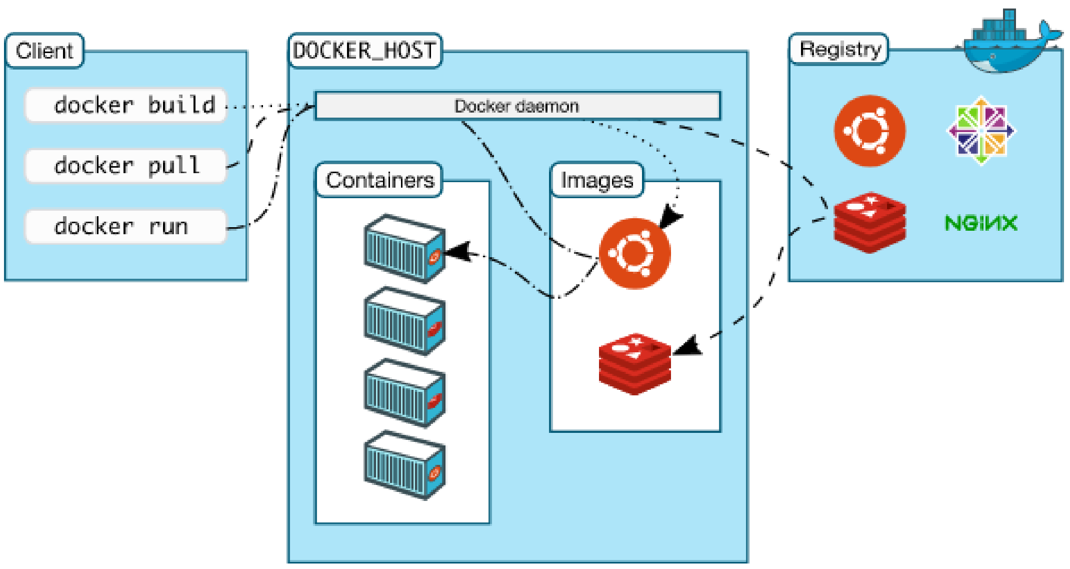


Figure 2.3: Docker Architecture (Docker overview, 2017)

a) Docker Demon

The Docker daemon (`Dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services (Docker overview, 2017).

b) Docker Client

The Docker client (`Docker`) is the primary way that many Docker users interact with Docker. When user uses commands such as `Docker run`, the client sends these commands to `Dockerd`, which carries them out. The Docker command uses the Docker API. The Docker client can communicate with more than one daemon (Docker overview, 2017).

c) Docker Registries

A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. User can even run his/her own private registry. If you use Docker Datacenter (DDC), it includes Docker Trusted Registry (DTR) (Docker overview, 2017).

When user uses the Docker pull or `Docker run` commands, the required images are pulled from your configured registry. When user uses the `Docker push` command, image is pushed to your configured registry.

Docker store allows user to buy and sell Docker images or distribute them for free. For instance, user can buy a Docker image containing an application or service from a software vendor and use the image to deploy the application into his/her testing, staging, and production environments. User can upgrade the application by pulling the new version of the image and redeploying the containers (Docker overview, 2017).

d) Docker Images

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, user may build an image which is based on the ubuntu image, but installs the Apache web server and the application, as well as the configuration details needed to make the application run.

User might create his/her own images or user might only use those created by others and published in a registry. To build user's own image, user should create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When user change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies (Docker overview, 2017).

e) Docker Container

A container is a runnable instance of an image. User can create, run, stop, move, or delete a container using the Docker API or CLI. User can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. User can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or run it. When a container stops, any changes to its state that are not stored in persistent storage disappears (Docker overview, 2017).

2.2.3 The Underlying Technology of Docker

Docker is written in Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality.

a) Namespaces

Docker uses a technology called namespaces to provide the isolated workspace called the container. When user run a container, Docker creates a set of namespaces for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Engine uses namespaces such as the following on Linux:

- The pid namespace: Process isolation (PID: Process ID).
- The net namespace: Managing network interfaces (NET: Networking).
- The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
- The mnt namespace: Managing filesystem mount points (MNT: Mount).
- The uts namespace: Isolating kernel and version identifiers. (UTS: UT).

b) Control Groups

Docker Engine on Linux also relies on another technology called control groups (`cgroups`). A `cgroup` limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, user can limit the memory available to a specific container.

c) Union File Systems

Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast. Docker Engine uses UnionFS to provide the building blocks for containers. Docker Engine can use multiple UnionFS variants, including AUFS, btrfs, and DeviceMapper.

`AUFS` a union filesystem. The aufs storage driver is the default storage driver used for managing images and layers on Docker for Ubuntu. AUFS is the most mature storage driver used by Docker. It provides fast container startup time, as well as efficient usage of memory and storage (Use the AUFS storage driver, 2017).

`Btrfs` is a next generation copy-on-write filesystem that supports many advanced storage technologies that make it a good fit for Docker. Btrfs is included in the mainline Linux kernel. Docker's btrfs storage driver leverages many Btrfs features for image and container management. Among these features are block-level operations, thin provisioning, copy-on-write snapshots, and ease of administration. You can easily combine multiple physical block devices into a single Btrfs filesystem (Use the BTRFS storage driver, 2017).

`Device Mapper` is a kernel-based framework that underpins many advanced volume management technologies on Linux. Docker's devicemapper storage driver leverages the thin provisioning and snapshotting capabilities of this framework for image and container management. For the systems where it is supported, devicemapper support is included in the Linux kernel. However, specific configuration is required to use it with Docker. For instance, on a stock installation of RHEL or CentOS, Docker will default to overlay, which is not a supported configuration. The devicemapper driver uses block devices dedicated to Docker and operates at the block level, rather than the file level. These devices can be extended by adding physical storage to your Docker host, and they perform better than using a filesystem at the level of the operating system (Use the Device Mapper storage driver, 2017).

d) Container Format

Docker Engine combines the namespaces, control groups, and `UnionFS` into a wrapper called a container format. The default container format is `libcontainer`. In the future, Docker may support other container formats by integrating with technologies such as BSD Jails or Solaris Zones (Docker overview, 2017).

2.3 Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, user uses a Compose file written in YAML or JSON to configure the application's services. Then, using a single command, user create and start all the services from the configuration.

Docker compose is basically three-way steps

- Define the app's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up the app in yaml file so they can be run together in an isolated environment.
- Lastly, run `Docker-compose up` and Compose will start and run your entire app.

Docker compose can manage the whole lifecycle of the application, such as

- Start, stop and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service (Overview of Docker Compose, 2017)

This all facilities also provided by Docker swarm. There is a detail discussion on Docker swarm in section 2.9.

2.4 Cloud Computing

Cloud computing is a modern computing paradigm that provides resources such as processing, storage, network and software that can be accessed over the Internet from any remote locations. Users can access cloud applications using web browsers, thin client machines or mobile devices, while all the data are processing. It functions more efficiently since it offloads management and maintenance tasks, thereby making computing a service rather than a product. The sharing of hardware resources is expected to reduce idle time of machines and increase their productivity. Due to the shift from computing being a resource to a utility, cloud computing introduces billing models that are based on time and utilities. On demand availability, ease of provisioning, dynamic and virtually infinite scalability are some of the key attributes of cloud computing. Services within the cloud are typically provided under the following three categories:

- Infrastructure as a Service (IaaS).
- Platform as a Service (PaaS).
- Software as a Service (Saas). (Azhagappan, 2015)

2.5 Orchestrator

Orchestration is the automated arrangement, coordination, and management of complex computer systems, middleware and services.

This usage of orchestration is often discussed in the context of service-oriented architecture, virtualization, provisioning, converged infrastructure and dynamic data centre topics. Orchestration in this sense is about aligning the business request with the applications, data, and infrastructure (Menychtas et al., 2011). It defines the policies and service levels through automated workflows, provisioning, and change management. This creates an application-aligned infrastructure that can be scaled up or down based on the needs of each application. Orchestration also provides centralized management of the resource pool, including billing, metering, and chargeback for consumption.

2.6 Application of Orchestrator in Cloud Computing

The conventional cloud solutions come with several limitations regarding the involvement of players with competitive applications in the cloud ecosystem because of the various, often complex and contradicting, business and technical requirements. In addition, the process for third parties to deploy their applications on cloud infrastructures, create new business models and establish synergies is very complicated since their requirements cannot be fulfilled from a single provider. In the cloud computing paradigm, the marketplace concept is expected to minimize the business complexities for providing and consuming all types of services by supporting all phases of the service lifecycle (knowledge, intentions, contract and settlement). The marketplaces offer the providers the ability to publish services and applications in a managed environment, which controls the business terms and conditions (price, revenue sharing, promotion, etc.), including integrated rating and billing capabilities. This requires automated selection processes not only for products but also of pricing models and SLAs (Service Level Agreements) through which these products are provisioned (Menychtas et al., 2011). Here Orchestrator plays a major role. Orchestrator provides the facility to the cloud administrator to manage the infrastructure, platform in an automated workflow. For instance, if a user wants to deploy some instances in some specific nodes in various networks then it is possible to do automatically by using the orchestrator.

2.7 Container in Cloud Computing

From the discussion of section 2.1 it can be realized that containers are a solution to the problem of how to get software to run reliably when moved from one computing environment to another. On the other hand, containers are very tiny and easy to deploy. In a cloud environment, there are lots of applications are running. So, it is essential to use containers as it is easy to deploy, scale and maintain.

Moreover, cloud could be a production environment for any company. And it is quite possible to vary the environment in production and development which usually creates problem during production. Container is a wonderful solution for this. Because a container has the application and the complete runtime environment variables which provide the platform to run the application in any environment.

Finally, containers are very light weighted compare to VM and provides complete isolated environment for each application. So, it provides much more scope to scale than VM on same infrastructure. These all things make containers very popular in cloud computing field.

2.8 Container Orchestrator

Before containers came along, orchestration referred to the process of managing any kind of infrastructure, usually with the assistance of automation. When containers became popular, people realized they needed an automated way to manage the hundreds or thousands of containers inside a cluster.

Orchestration is occasionally used in a broader sense to refer to platforms designed to manage all aspects of setting up and deploying a containerized application (Tozzi, 2017). Orchestration tools extend lifecycle management capabilities to complex, multi-container workloads deployed on a cluster/cloud of machines.

The process of orchestration typically involves tooling that can automate all aspects of application management from initial placement, scheduling and deployment to steady-state activities such as update, deployment, update and health monitoring functions that support scaling and failover. These capabilities have come to characterize some of the core features user's expectations offer modern container orchestration tools. Orchestration tools provide an option for DevOps teams to declare the blueprint for an application workload and its configuration in a standard schema, using languages such as YAML or JSON. These definitions also carry crucial information about the repositories, networking (ports), storage (volumes) and logs that support the workload. Moreover, in a distributed deployment consisting of containers running on multiple hosts, container discovery becomes critical. Web servers need to dynamically discover the database servers, and load balancers need to discover and register web servers. Orchestration tools provide, or expect, a distributed key-value store, a lightweight DNS or some other mechanism to enable the discovery of containers (MSV, 2016).

2.8.1 General Container Orchestrator Architecture

To explain the higher-level architecture the below figure 2.4 has used

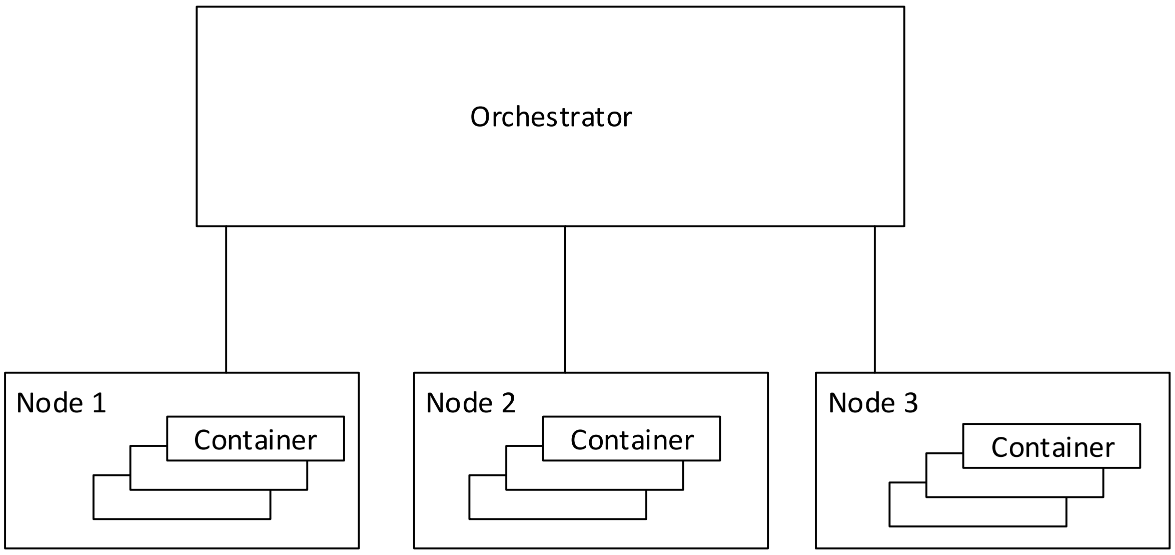


Figure 2.4: Higher level architecture of container orchestrator

In figure 2.4 it is seen that there is cluster of three nodes. The orchestrator able to maintain this cluster in an automated way, such as deploy containers, discovery service, load balancing, health checking of the containers and many more. If a user uses the orchestrator, then it is not required to do the application management and deployment in each host. Administrator can maintain the applications in an automated workflow which give him/her the facilities to maintain and scale a large scale of applications.

There are many container orchestrators is used in the industry such as Docker swarm, Kubernetes, Google container, CentOS Fleet, Rancher etc. In this thesis, the focus is mainly on Docker Swarm and Kubernetes. In the next section, there is a discussion on Docker swarm and Kubernetes.

2.9 Docker Swarm

2.9.1 Docker Swarm Definition

Docker swarm is the cluster management and orchestration features embedded in the Docker Engine are built using SwarmKit. Docker engines participating in a cluster are running in swarm mode. User enables swarm mode for an engine by either initializing a swarm or joining an existing swarm.

A swarm is a cluster of Docker engines, or nodes, where user deploys services. The Docker Engine CLI and API include commands to manage swarm nodes (e.g., add or remove nodes), and deploy and orchestrate services across the swarm.

When you run Docker without using swarm mode, user execute container commands. When user runs the Docker in swarm mode, user orchestrates services. User can run swarm services and standalone containers on the same Docker instances (Swarm mode key concepts, 2017).

2.9.2 Docker Swarm Architecture

To explain the Docker swarm the below figure 2.5 has used

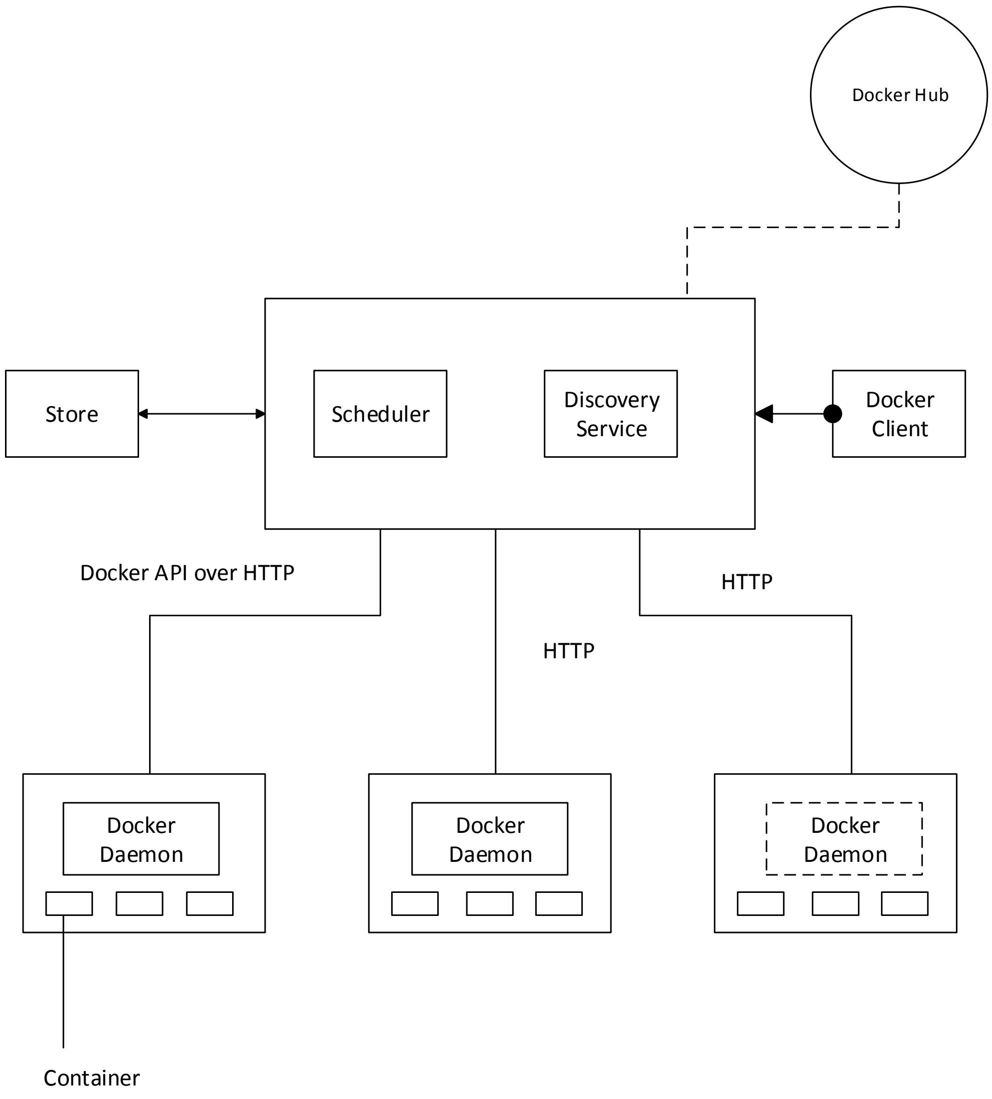


Figure 2.5: Docker Swarm Architecture

There are four key components in Docker swarm.

- Swarm Manager
- Swarm node
- Scheduler
- Discovery

a) Swarm Manager

Swarm Manager manages the nodes in the cluster. It usages Docker API to access the Docker demon running on each node. Nodes are added to the swarm manager by call back from Discovery service's fetch function (Rajdeep Dua, 2017).

b) Swarm Node

This is the runtime instance which represent the nodes in the cluster. It talks to actual host using Docker go client. It created from a discovery entry fetched from a discovery service. The major elements of swarm node is

- Id
- IP address
- Map of containers
- Map of Images
- Health state of the node
- Total CPUs
- Used CPUs
- Total memory
- Used memory (Rajdeep Dua, 2017)

c) Scheduler

It is responsible for scheduling a container on a node. Docker swarm scheduler currently supports two strategies.

- BinPacking Strategy
- Random Strategy

BinPacking strategy will rank their nodes using their CPU and RAM available and will return the node the most packed already. To avoid fragmentation, it will leave room for bigger containers on unused machines.

The Random strategy, as its name says pick any random node to deploy containers. This is mainly used for debug.

Scheduler uses the following filters for container placement on a node

- Affinity filter: If user creates two containers, this filter sets the second container next to first container.
- Constraint filter: When creating a container, the user can select a subset of nodes that should be considered for scheduling by specifying one or more sets of matching key/value pairs.
- Port filter: Select node where public port required by the container is not already used
- Health filter: Select only healthy nodes from the cluster

d) Swarm Store

Swarm store stores the state of the cluster. State is loaded in memory when the cluster starts. It has implemented a JSON file. The life cycle events of a store are in below

- Get state for a key
- Store the state of a container
- Load all the data stored
- Replace the state of the key with a new state
- Delete the state (Rajdeep Dua, 2017)

e) Discovery Service

Discovery service helps swarm manager to discover nodes. There are three main functions

- Resister: include a new node in the cluster
- Watch: Call back method
- Fetch: Fetch the list of entities

2.9.3 Swarm Mode Key Concepts

a) Node

A node is an instance of the Docker engine participating in the swarm. User can also think of this as a Docker node. User can run one or more nodes on a single physical computer or cloud server, but production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines.

To deploy the application to a swarm, user submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default, manager nodes also run services as worker nodes, but you can configure them to run manager tasks exclusively and be manager-only nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker (Swarm mode key concepts, 2017).

b) Services and Tasks

A service is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

When you create a service, you specify which container image to use and which commands to execute inside running containers.

In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.

For global services, the swarm runs one task for the service on every available node in the cluster.

A task carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail (Swarm mode key concepts, 2017).

c) Load Balancing

The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort or user can configure a PublishedPort for the service. User can specify any unused port. If user does not specify a port, the swarm manager assigns the service a port in the 30000-32767 range.

External components, such as cloud load balancers, can access the service on the PublishedPort of any node in the cluster whether the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance.

Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service (Swarm mode key concepts, 2017).

2.9.4 How Swarm Mode Works

Swarm mode is available with Docker from version 1.12.0 or later. There are two major parts of Docker swarm.

- Nodes
- Services

a) Nodes

Docker Engine 1.12 introduces swarm mode that enables user to create a cluster of one or more Docker Engines called a swarm. A swarm consists of one or more nodes: physical or virtual machines running Docker Engine 1.12 or later in swarm mode.

There are two types of nodes: managers and workers. The architecture of Docker nodes are shown in below figure 2.6

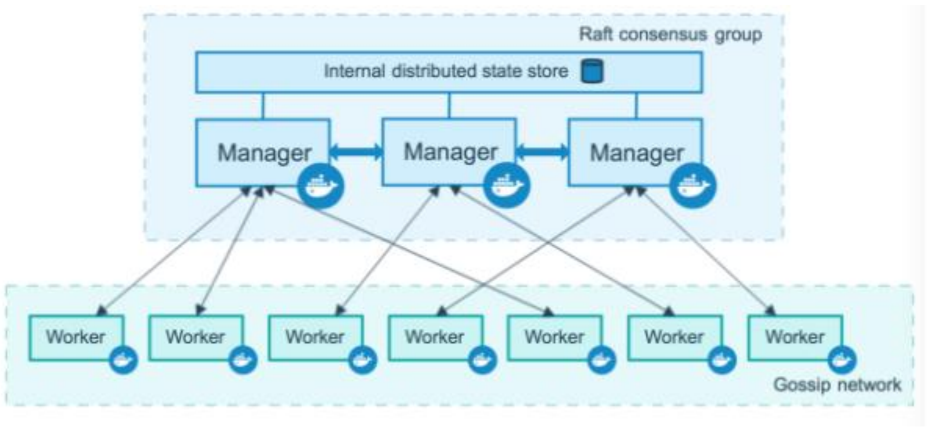


Figure 2.6: Docker nodes architecture (How nodes work, 2017)

Manager nodes handle cluster management tasks.

- Maintaining cluster state
- Scheduling services
- Serving swarm mode HTTP API endpoints

Docker swarm using Raft implementation (How nodes work, 2017). Raft is a consensus algorithm for managing a replicated log. Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus. Raft is equivalent to Paxos and it is as efficient as Paxos but its structure is different than the Paxos which make Raft more understandable than Paxos and also provides a better foundation for building physical system (Ongaro and Ousterhout, 2017).

Using a Raft implementation, the managers maintain a consistent internal state of the entire swarm and all the services running on it. For testing purpose, it is ok to run a single manager. If the manager in a single-manager swarm fails, the services will continue to run, but user will need to create a new cluster to recover. To use the swarm mode's fault tolerance system Docker recommends user to implement odd number of nodes. Swarm mode fault tolerance system follows the following formula

- An N manager cluster will tolerate the loss at most $(N-1)/2$

When user has multiple managers, user can recover from the failure of a manager node without downtime.

Worker nodes are also instances of Docker Engine whose sole purpose is to execute containers. Worker nodes don't participate in the Raft distributed state, make scheduling decisions, or serve the swarm mode HTTP API.

User can create a swarm of one manager node, but user cannot have a worker node without at least one manager node. By default, all managers are also workers. In a single manager node cluster, you can run commands like Docker service create and the scheduler will place all tasks on the local Engine (How nodes work, 2017).

It is possible to promote a worker node in a manager node by running `Docker node promote` (How nodes work, 2017).

b) Services

To deploy an application image when Docker Engine is in swarm mode, user creates a service. Frequently a service will be the image for a microservice within the context of some larger application. Examples of services might include an HTTP server, a database, or any other type of executable program that you wish to run in a distributed environment.

When user creates a service, user specify which container image to use and which commands to execute inside running containers. User also define options for the service including:

- the port where the swarm will make the service available outside the swarm
- an overlay network for the service to connect to other services in the swarm
- CPU and memory limits and reservations
- a rolling update policy
- the number of replicas of the image to run in the swarm (How services work, 2017)

The relation between Services, tasks and containers are shown in below figure 2.7

When user deploys the service to the swarm, the swarm manager accepts the service definition as the desired state for the service. Then it schedules the service on nodes in the swarm as one or more replica tasks. The tasks run independently of each other on nodes in the swarm.

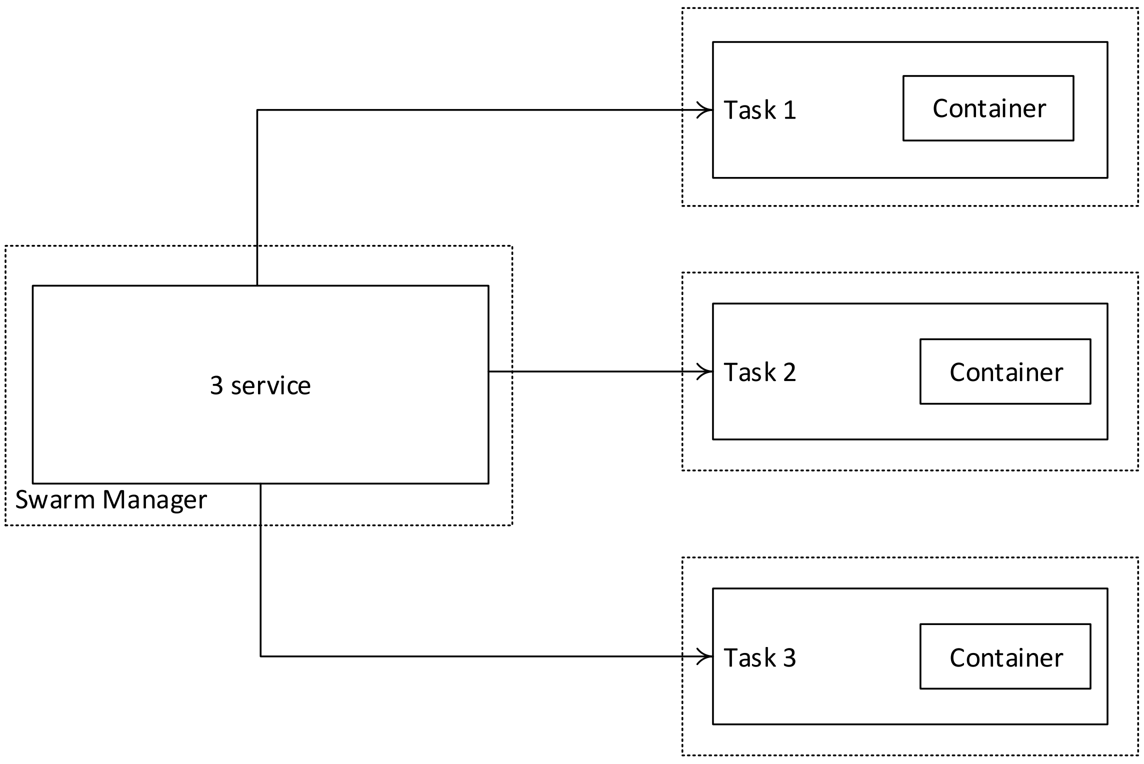


Figure 2.7: Relation of service task and container [How services work, 2017]

A task is the atomic unit of scheduling within a swarm. When user declare a desired service state by creating or updating a service, the orchestrator realizes the desired state by scheduling tasks.

A task is a one-directional mechanism. It progresses monotonically through a series of states: assigned, prepared, running, etc. If the task fails the orchestrator removes the task and its container and then creates a new task to replace it according to the desired state specified by the service.

The underlying logic of Docker swarm mode is a general-purpose scheduler and orchestrator. The service and task abstractions themselves are unaware of the containers they implement. The below figure shows how swarm mode accepts service create requests and schedules tasks to worker nodes.

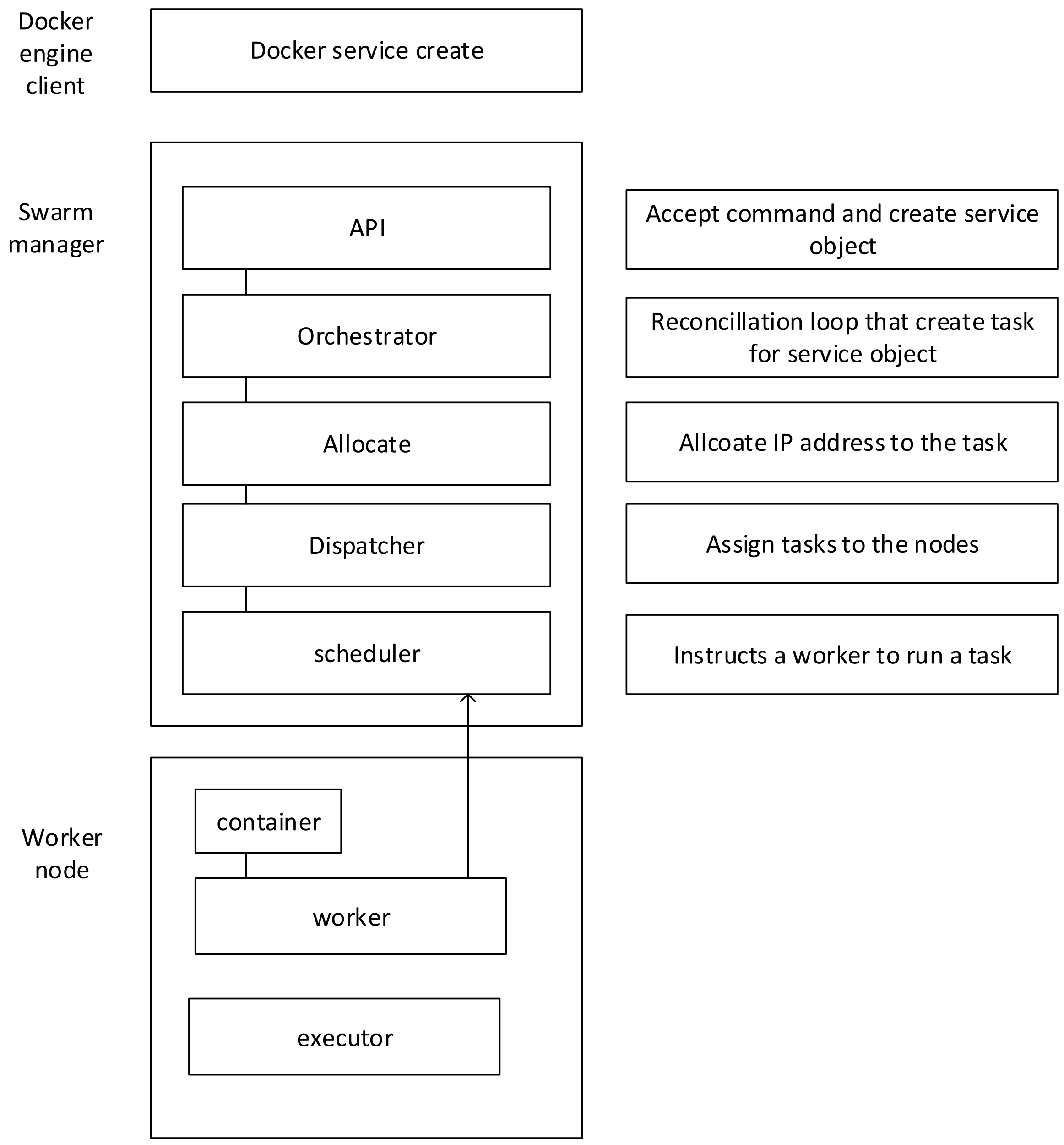

Docker engine client
Docker service create
Swarm manager
API
Orchestrator
Allocate
Dispatcher
scheduler
Accept command and create service object
Reconcillation loop that create task for service object
Allcoate IP address to the task
Assign tasks to the nodes
Instructs a worker to run a task
Worker node
container
worker
executor

Figure 2.8: Service workflow (How services work, 2017)

A service may be configured in such a way that no node currently in the swarm can run its tasks. In this case, the service remains in state pending (How services work, 2017).

There are two kinds of service

- replicated
- global

For a replicated service, user specify the number of identical tasks user wants to run. For example, user decide to deploy an HTTP service with three replicas, each serving the same content.

A global service is a service that runs one task on every node. There is no pre-specified number of tasks. Each time you add a node to the swarm, the orchestrator creates a task and the scheduler assigns the task to the new node (How services work, 2017).

2.10 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes build upon on the experience on Google, combined with the ideas and practices from community (Production-Grade Container Orchestration, 2017).

2.10.1 Kubernetes Architecture

The architecture of Kubernetes has been explained in below figure 2.9

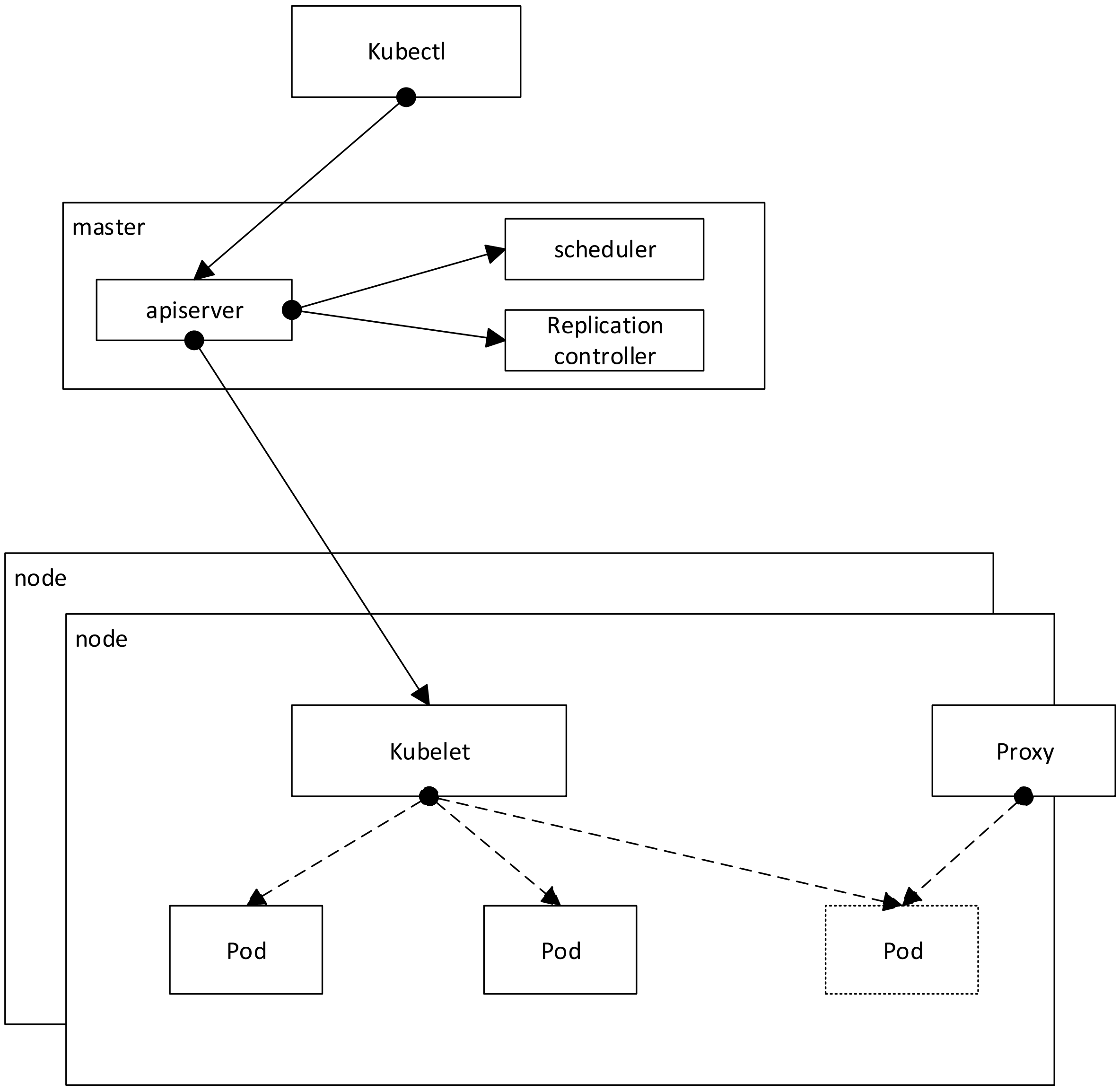


Figure 2.9: Kubernetes Architecture

To work with Kubernetes, user use Kubernetes API objects to describe the cluster's desired state: what applications or other workloads you want to run, what container images they use, the number of replicas, what network and disk resources you want to make available, and more. You set your desired state by creating objects using the Kubernetes API, typically via the command-line interface, `kubectl`.

The Kubernetes master is responsible for maintaining the desired state for cluster. When user interacts with Kubernetes, such as by using the kubectl command-line interface, user is communicating with cluster's Kubernetes master. The "master" refers to a collection of processes managing the cluster state. Typically, these processes are all run on a single node in the cluster, and this node is also referred to as the master. The master can also be replicated for availability and redundancy (Concepts, 2017). User can create deploy a service using apiserver which schedule the replication controller. And the replication controller will deploy pod or multiple pods on the nodes which are included in the cluster.

The nodes in a cluster are the machines (VMs, physical servers, etc) that run the applications and cloud workflows. The Kubernetes master controls each node; user rarely interacts with nodes directly (Concepts, 2017). User are deploying service through master. Every service is based on a replication controller and the replication controller deploys the pods on nodes based on user defined configuration. Every pod has the container or a collection of containers. So, the pod is the atomic unit for Kubernetes.

There are two major parts in Kubernetes architecture

- Nodes
- Master-Node communication (https://Kubernetes.io/docs/concepts/architecture/nodes, 2017)

a) Nodes

A node is a worker machine in Kubernetes, previously known as a minion. A node may be a VM or physical machine, depending on the cluster. Each node has the services necessary to run pods and is managed by the master components. The services on a node include Docker, kubelet and kube-proxy (https://Kubernetes.io/docs/concepts/architecture/nodes, 2017).

b) Master-Node communication

There are two major part in Master-Node communication. One is cluster to master communication and the other is cluster to master communication.

All communication paths from the cluster to the master terminate at the apiserver. None of the other master components are designed to expose remote services. In a typical deployment, the apiserver is configured to listen for remote connections on a secure HTTPS port (443) with one or more forms of client authentication enabled. Pods that wish to connect to the apiserver can do so securely by leveraging a service account so that Kubernetes will automatically inject the public root certificate and a valid bearer token into the pod when it is instantiated. The Kubernetes service is configured with a virtual IP address that is redirected to the HTTPS endpoint on the apiserver. As a result, the default operating mode for connections from the cluster (nodes and pods running on the nodes) to the master is secured by default and can run over untrusted and/or public networks .

There are two primary communication paths from the master (apiserver) to the cluster. The first is from the apiserver to the kubelet process which runs on each node in the cluster. The second is from the apiserver to any node, pod, or service through the apiserver's proxy functionality (https://Kubernetes.io/docs/concepts/architecture/master-node-communication, 2017).

The detail discussion about all key concepts of Kubernetes such as Service, replication controller, pods, nodes are in following sections.

2.10.2 Kubernetes Key Concepts

a) The Kubernetes nodes

The Kubernetes node has the services necessary to run application containers and be managed from the master systems. A node contains the following information

- Address
- Deprecated
- Condition
- Capacity
- Info

The Address in the node has three field which are HostName, ExternalIP, InternalIP. The hostname as reported by the node's kernel. Can be overridden via the `kubelet --hostname-override` parameter. Deprecated is using instead of Node phase. The condition field describe the status of all running nodes. Capacity describe the resource available in the nodes such as CPU, memory etc. Finally, the info field has the general information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), OS name. The information is gathered by Kubelet from the node (https://Kubernetes.io/docs/concepts/architecture/nodes, 2017).

b) Pods

A Pod is the basic building block of Kubernetes–the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents a running process on your cluster. A Pod encapsulates an application container or multiple containers, storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources. Each Pod is meant to run a single instance of a given application. If user wants to scale your application horizontally, user should use multiple Pods, one for each instance.

Pods are designed to support multiple cooperating processes (as containers) that form a cohesive unit of service. The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster. The containers can share resources and dependencies, communicate with one another, and coordinate when and how they are terminated. The workflow of pod is given in below figure 2.10

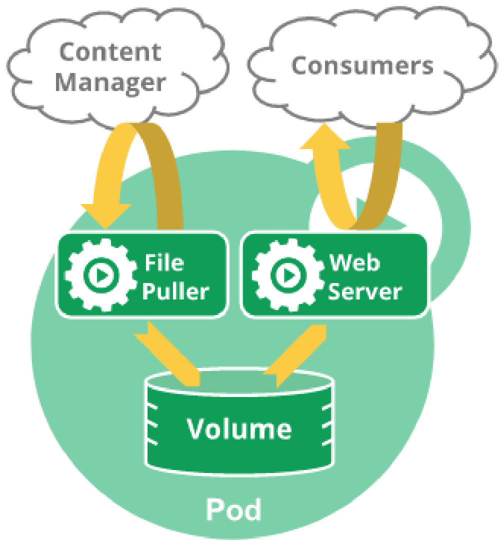


Figure 2.10: Pod workflow

Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports. Containers inside a Pod can communicate with one another using localhost. When containers in a Pod communicate with entities outside the Pod, they must coordinate how they use the shared network resources (such as ports) (https://Kubernetes.io/docs/concepts/workloads/pods/pod-overview, 2017).

c) ReplicaSet

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods along with a lot of other useful features (https://Kubernetes.io/docs/concepts/workloads/controllers/replicaset, 2017).

d) Replication Controller

A ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated (https://Kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller, 2017). For example, if a user wants to recreate the pod after doing necessary maintenance work then he/she has to use the replication controller.

e) Service

A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service. The set of Pods targeted by a Service is (usually) determined by a Label Selector (https://Kubernetes.io/docs/concepts/services-networking/service, 2017). Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system (https://Kubernetes.io/docs/concepts/services-networking/service, 2017).

Every node in a Kubernetes cluster runs a kube-proxy. kube-proxy is responsible for implementing a form of virtual IP for Services of type other than ExternalName. In Kubernetes v1.0 the proxy was purely in userspace. In Kubernetes v1.1 an iptables proxy was added, but was not the default operating mode. Since Kubernetes v1.2, the iptables proxy is the default. The workflow of service is shown in below figure 2.11

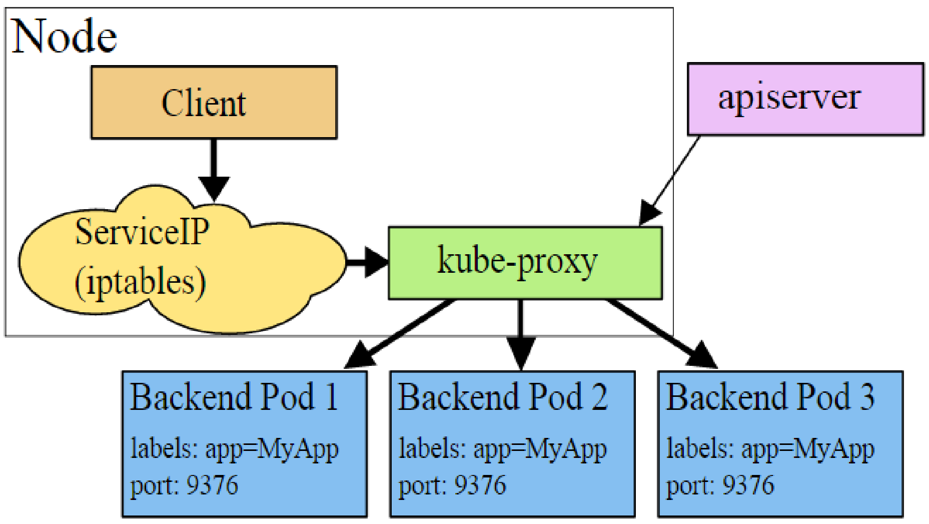


Figure 2.11: Service workflow (https://Kubernetes.io/docs/concepts/services-networking/service, 2017)

kube-proxy watches the Kubernetes master for the addition and removal of Service and Endpoints objects. For each service, it opens a port (randomly chosen) on the local node. Any connections to this "proxy port" will be proxied to one of the Service's backend Pods (as reported in Endpoints). Which backend Pod to use is decided based on the Session Affinity of the Service. Lastly, it installs iptables rules which capture traffic to the Service's clusterIP (which is virtual) and Port and redirects that traffic to the proxy port which proxies the backend Pod. The net result is that any traffic bound for the Service's IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or Services or Pods. By default, the choice of backend is round robin [https://Kubernetes.io/docs/concepts/services-networking/service, 2017].

To express the service to use the application user can expose the service in the following way

- `ClusterIP`: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default `ServiceType`.
- `NodePort`: Exposes the service on each Node's IP at a static port (the `NodePort`). A `ClusterIP` service, to which the `NodePort` service will route, is automatically created. You'll be able to contact the `NodePort` service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`. In this thesis work `NodePort` has been used largely.
- `LoadBalancer`: Exposes the service externally using a cloud provider's load balancer. `NodePort` and `ClusterIP` services, to which the external load balancer will route, are automatically created.
- `ExternalName`: Maps the service to the contents of the `externalName` field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of `kube-dns`.

2.11 Summary of Theoretical Background

In this chapter, there has been a discussion about container technology, Docker container, cloud computing, orchestrator, orchestrator in cloud computing, container in cloud computing, container orchestrator, Docker swarm and Kubernetes. In this thesis work both Docker swarm and Kubernetes has integrated for doing various experiment to evaluate the performance between Docker swarm and Kubernetes.

3 Requirements Analysis

In this thesis work the main goal is to integrate different orchestrator such as Docker swarm and Kubernetes with clusters of Docker containers and realize the performance of both orchestrator based on different parameter.

3.1 General Objectives

In this work, both Docker swarm and Kubernetes will be installed to manage and orchestrate various cloud native containerized applications. In both cases Docker containers will be used as applications. In the case of Docker swarm, a swarm manager will act as a scheduler and discover services for different swarm nodes. In case of Kubernetes the master node will play the manager role for various other working nodes. Each working node for both Docker swarm and Kubernetes will host multiple Docker containers. After completing the integration of Docker swarm and Kubernetes, the evaluation between Docker swarm and Kubernetes part will come. The evaluation will be done based on following parameters.

- Setting up cluster
- Networking in between cluster
- Deployment of application
- Auto scaling of application
- Load Balancing
- Container update and Rollbacks
- Live migration
- Performance analysis

3.2 Clarifying the Requirements

In this thesis topic, the key parts are Docker swarm and Kubernetes. In the below part, the details discussion will be appeared which will be follow to reach the target state of this thesis topic. **Docker Swarm** is a native clustering system for Docker. It turns a pool of Docker hosts into a single, virtual host using an API proxy system. A swarm is a cluster of Docker engines, or nodes, where people deploy services. The Docker Engine CLI and API include commands to manage swarm nodes (e.g., add or remove nodes), and deploy and orchestrate services across the swarm. To deploy the application to a swarm, it is important to submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes. Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks. In the following figure 3.1 is showing how the Docker Swarm is working. The swarm mode is communicating between Docker hub and demons. A demon should have multiple containerized application. And the Docker swarm also managing multiple demons of multiple container.

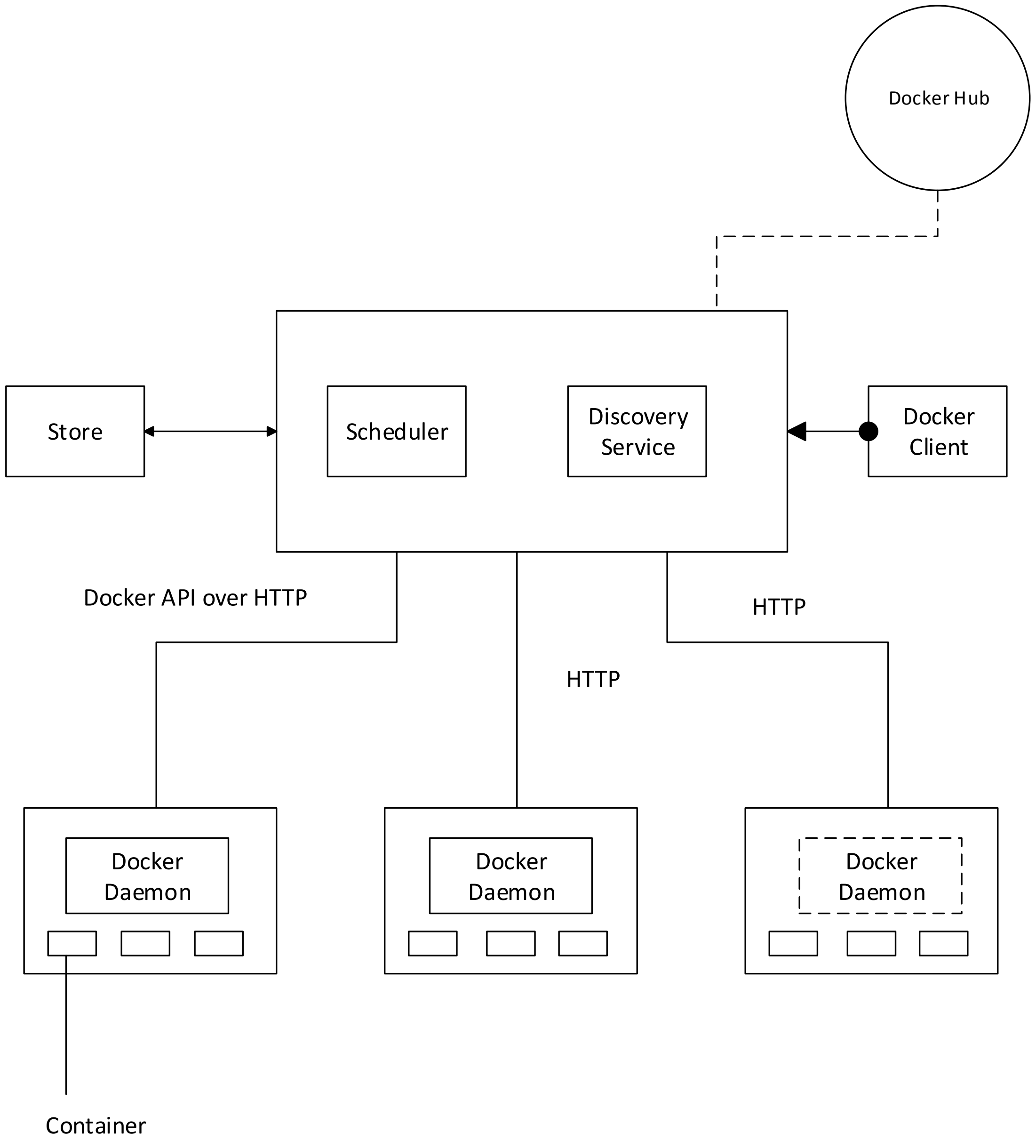


Figure 3.1: Docker Swarm architecture

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. At a minimum, Kubernetes can schedule and run application containers on clusters of physical or virtual machines. However, Kubernetes also allows developers to 'cut the cord' to physical and virtual machines, moving from a host-centric infrastructure to a container-centric infrastructure, which provides the full advantages and benefits inherent to containers. Kubernetes provides the infrastructure to build a truly container-centric development environment. The following figure 3.2 has explained the architecture of Kubernetes.

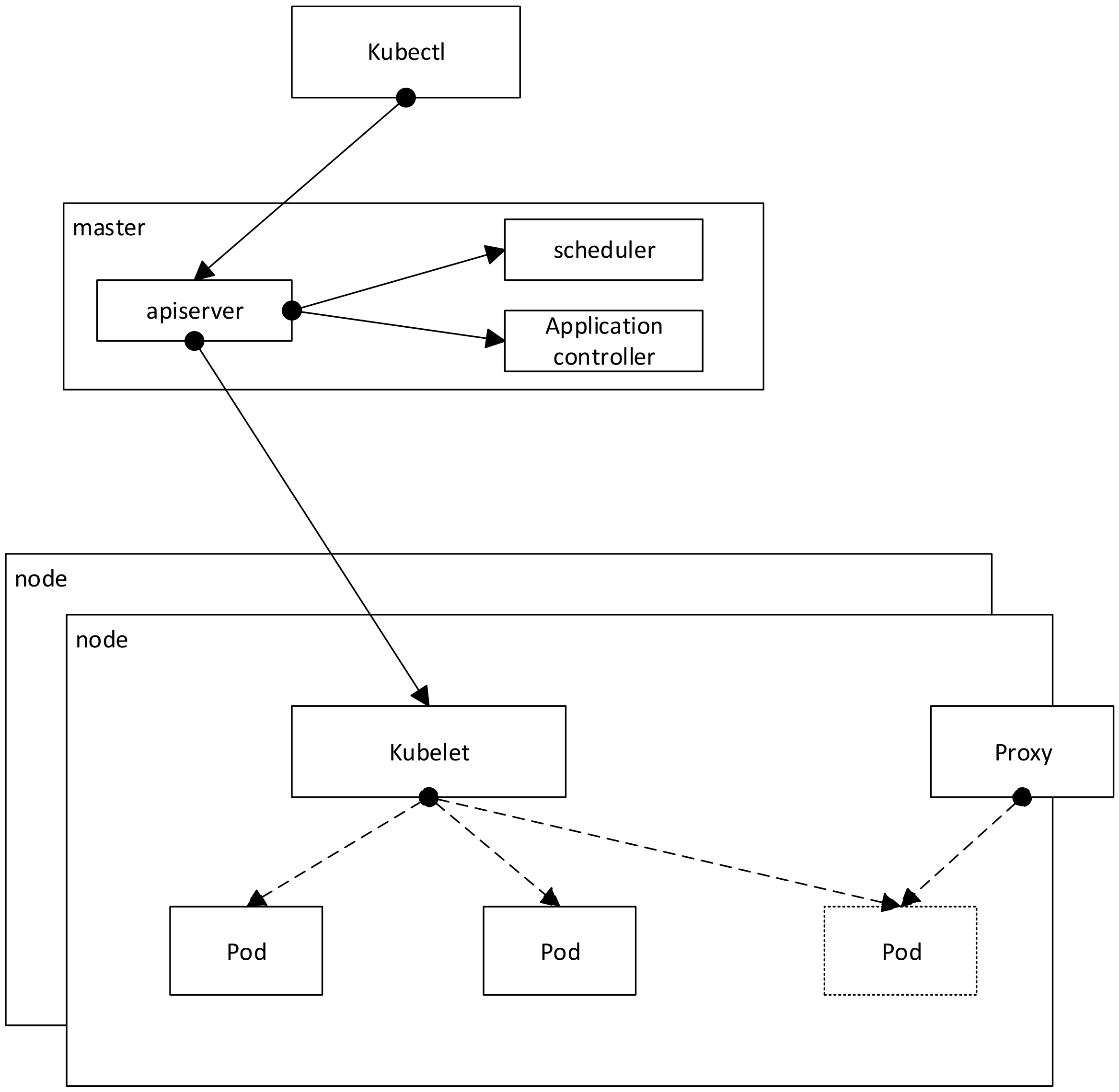


Figure 3.2: Kubernetes architecture

3.3 Time frames

Milestones:

05.04.2017 to 28.04.2017:	Analysis of the requirements
28.04.2017:	Submission of the requirements analysis
28.04.2017 to 30.04.2017:	Install necessary VM and establish internal network for Docker swarm cluster
01.05.2017 to 02.05.2017:	Install necessary VM and establish internal network for Kubernetes cluster
03.05.2017 to 12.05.2017:	Create Docker swarm cluster

13.05.2017 to 31.05.2017:	Create Kubernetes cluster
01.06.2017 to 5.06.2017	Deployment of application
5.06.2017 to 10.06.2017	Auto scaling of application
11.06.2017 to 20.06.2017	Load balancing
21.06.2017 to 30.06.2017	Container update and rollbacks
31.06.2017 to 10.07.2017	Performance analysis
11.07.2017 to 25.07.2017	Live migration
26.07.2017 to 04.09.2017	Documentation
05.09.2017:	Submission of the thesis documents

3.4 Target State

In the case of implementation, the first target is to establish two clusters of Docker containers which will be orchestrated by Kubernetes and/or Docker swarm respectively. This will be lead the work to establish the architecture what was explained earlier. To implement the idea the following application Docker containers will be used

- A Web application printing the container/pod id
- Web server based on Apache tomcat
- Database server based on MySQL
- Prime number counter which counts all prime number between 1 million

3.4.1 Docker Swarm Implementation

For the implementation of the prototype, the following figure 3.3 will state the project architecture for Docker swarm

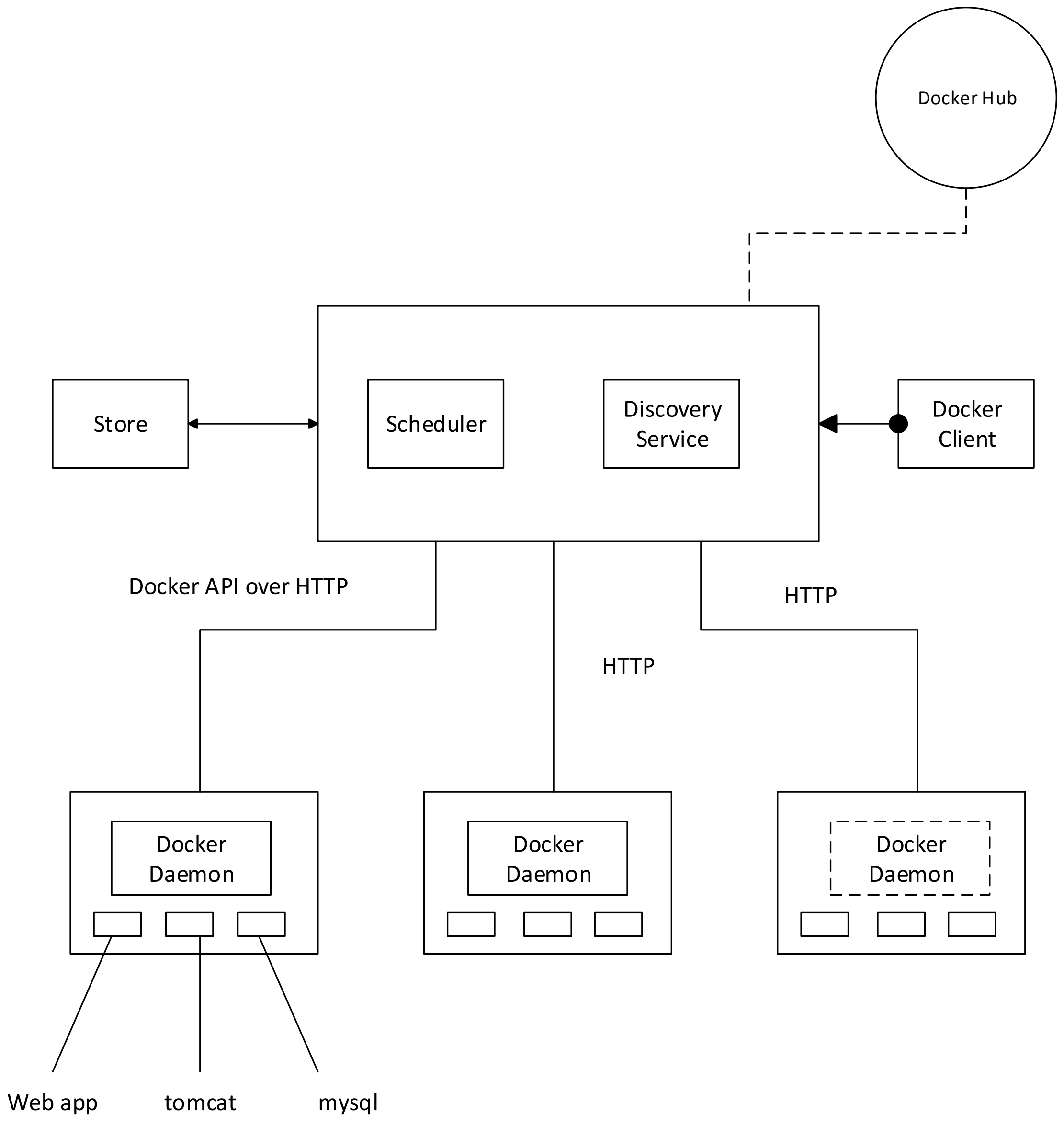


Figure 3.3: Docker swarm implementation

To achieve this goal stated in figure 3.3 the first step will be to establish swarm mode which will use all the facilities of Docker and Docker compose using Docker API. After that, make a demon with one Docker container and with three containers. Finally, make a complex Docker swarm scenario with three demons.

All the demons will act as worker node of the implemented swarm cluster and over the top of the demons there will be a master node which acts as orchestrator for that cluster.

Working environment for Docker swarm

From figure 3.3 it could be realized that in the swarm cluster there will be four Docker machines, where one machine will work as master node and the other three machines will work as worker nodes. The following figure 3.4 will explain the simplified implementation state

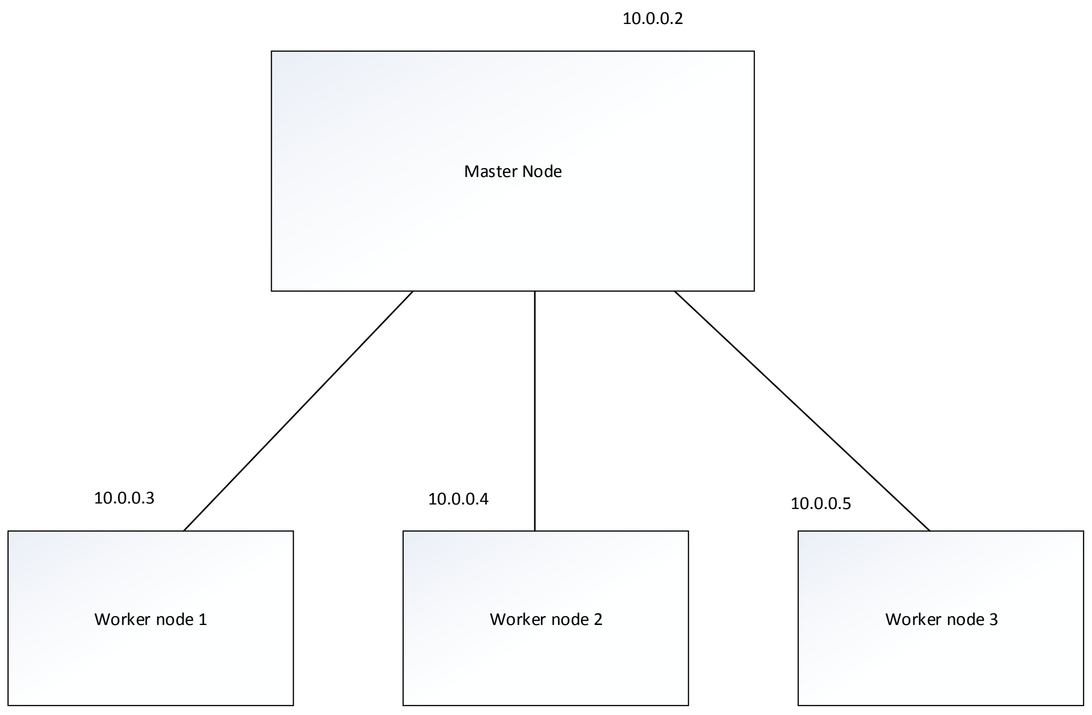


Figure 3.4: Simplified Docker swarm implementation goal

All the nodes will be implemented in separate virtual machines. The configuration of all virtual machines are in below table 3.1

Table 3.1 : Hardware plan for Docker swarm cluster

No	VM Name	Operating system	Hard drive	Random-access memory
1	Swarm_master	Ubuntu 14.04.5	10 GB	1 GB
2	node 1	Ubuntu 14.04.5	10 GB	1 GB
3	snode 2	Ubuntu 14.04.5	10 GB	1 GB
4	snode 3	Ubuntu 14.04.5	10 GB	1 GB

All the virtual machines need to communicate with each other to work in a cluster. The IP plan for this implementation is in below table 3.2.

Table 3.2 : IP plan for Docker swarm cluster

No	VM Name	IP address	Subnet mask
1	Swarm_master	10.0.0.2	255.255.255.0
2	node 1	10.0.0.3	255.255.255.0
3	snode 2	10.0.0.4	255.255.255.0
4	snode 3	10.0.0.5	255.255.255.0

Finally, the implementation should work in a way that the cloud architect can create the containers from the master node to any working node. Also, it should be possible to achieve the services like load balancing, hot migration and scalability.

3.4.2 Kubernetes Implementation

In this implementation, the main goal is to establish a master node which will orchestrate the other nodes. Each node will have multiple pods and every pod will have three applications. The following figure 3.5 will describe the Kubernetes in more details.

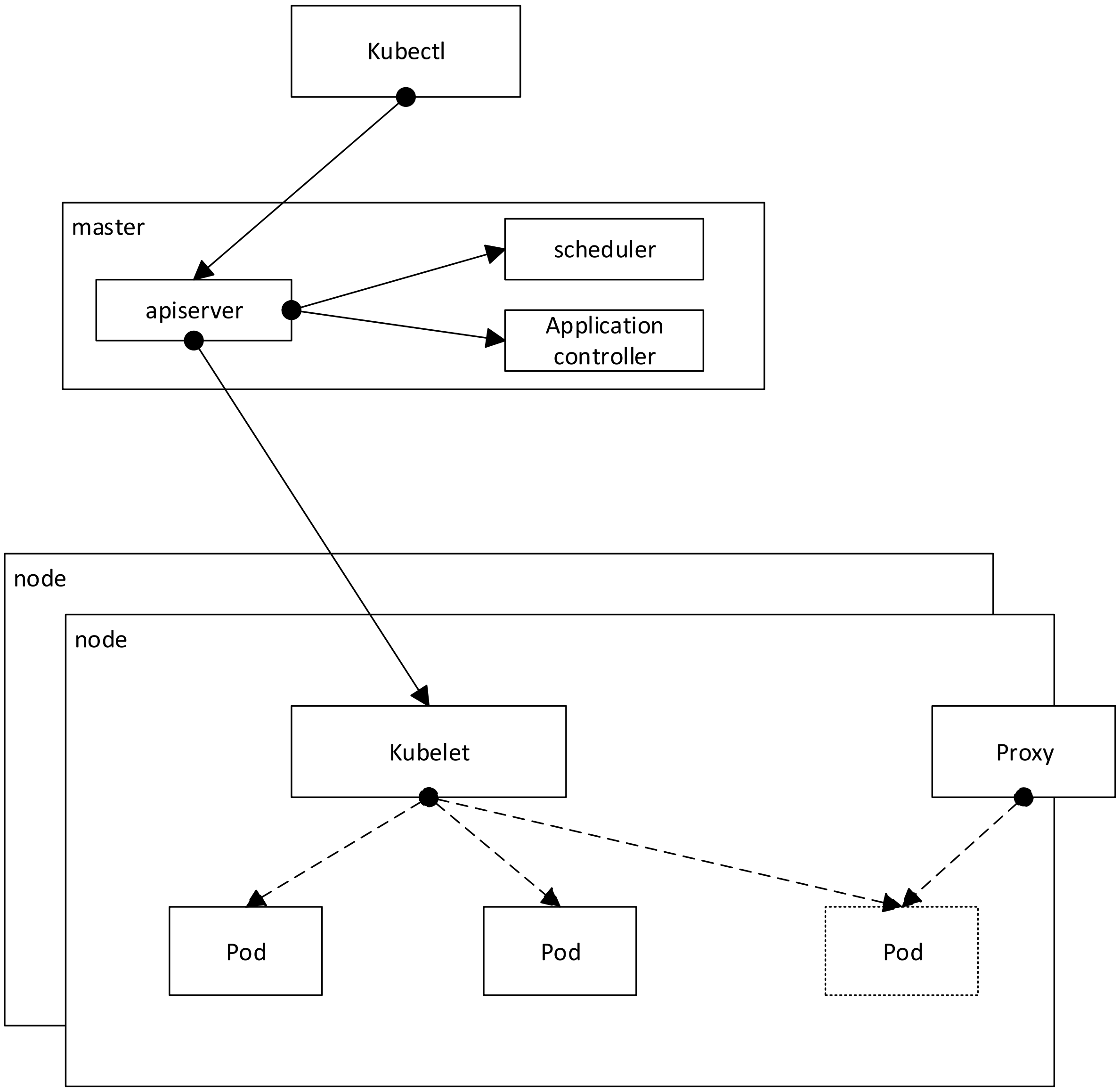


Figure 3.5: Kubernetes implementation

From this figure 3.5 it is understandable that in this cluster there will be a master node which orchestrated the other worker nodes in the same cluster. Every node will have multiple nodes which will have multiple pods. Every pod will have multiple Docker containers.

Working environment for Kubernetes

Similarly, as Docker swarm in Kubernetes there will be one master node and three working nodes. All nodes will be installed in virtual machines. The following figure 3.6 will explain the simplified implementation state of kubernetes.

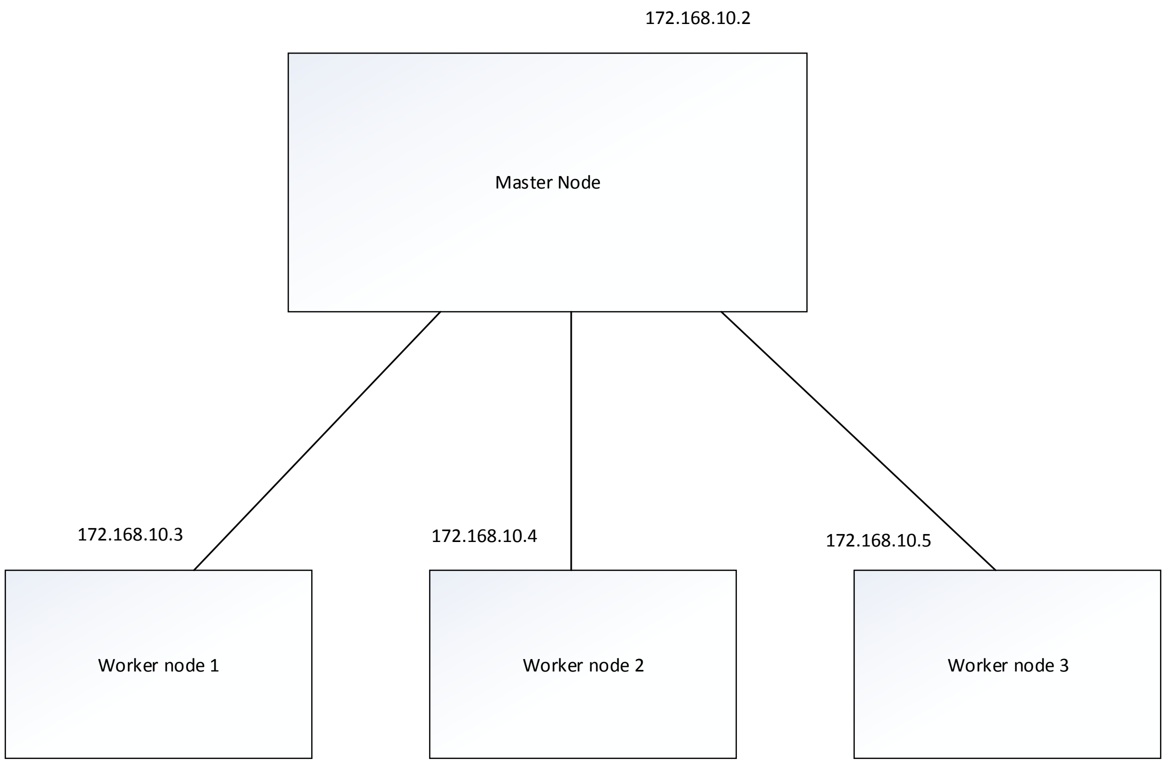


Figure 3.6: Simplified Kubernetes implementation goal

The configuration for those virtual machines are in below table 3.3.

Table 3.3 : Hardware plan for Kubernetes cluster

No	VM Name	Operating system	Hard drive	Random-access memory
1	kmaster	Ubuntu 14.04.5	10 GB	1 GB
2	knode 1	Ubuntu 14.04.5	25 GB	1 GB
3	knode 2	Ubuntu 14.04.5	25 GB	1 GB
4	knode 3	Ubuntu 14.04.5	25 GB	1 GB

The IP plan for this implementation are in below table 3.4

Table 3.4 : IP plan for Kubernetes cluster

No	VM Name	IP address	Subnet mask
1	kmaster	172.168.10.2	255.255.255.0
2	knode 1	172.168.10.3	255.255.255.0
3	knode 2	172.168.10.4	255.255.255.0
4	snode 3	172.168.10.5	255.255.255.0

This implementation also has the same purpose as Docker swarm which means creating containers from master node to all working nodes and also check all the services like load balancing, hot migration and scalability.

3.5 Use Cases for the Prototype

In this implementation, the evaluation between Kubernetes and Docker swarm will play a major role. The points for evaluation are explained below

a) Setting up cluster:

In the implementation two separate cluster will be established for both Docker swarm and Kubernetes respectively. Each cluster will have four virtual machines. One virtual machine will work as a master node and the other three will work as worker node. The architecture will be same as figure 3.4. The way of setting up clusters for both Docker swarm and Kubernetes will be evaluated.

b) Networking in between cluster:

According to figure 3.4 and 3.5 both Docker swarm cluster and Kubernetes cluster will have four machines. The goal is to make distributed network of four machines where each machine can communicate with each other. The figure 3.7 will explain the planned cluster network.

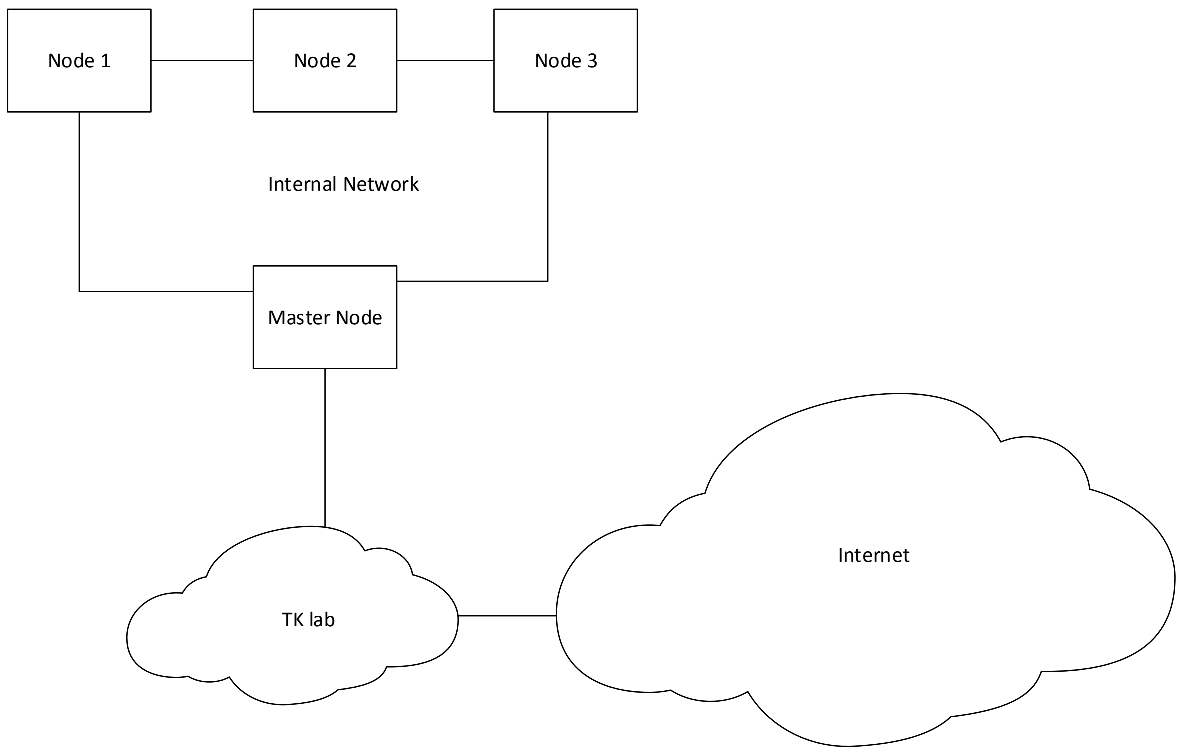


Figure 3.7: Network scenario of a cluster

The goal is to establish the network in such way that every node will communicate with each other through an internal network. The master node will communicate with the network of TK lab which provide the way of communicating with internet for that cluster. This scenario is applicable for both Docker swarm and Kubernetes. The IP plan for both Docker swarm and Kubernetes is given in Table 3.2 and 3.4 respectively.

c) Deployment of application:

It will be examined that how application can be deployed automatically by using any predefined script or technology for both Docker swarm and Kubernetes. In Docker swarm, it will use the Swarm API which provides the familiar functionally from Docker itself. On the other hand, Kubernetes uses its own client, API and YAML definition which differ of the standard Docker environment. In this implementation, three applications will be deployed as container in every node of the cluster from the master node for both Docker swarm and Kubernetes. The application will be Hello world, Web server based on Apache tomcat, Database server based on MySQL. Figure 3.8 will explain the goal of application deployment

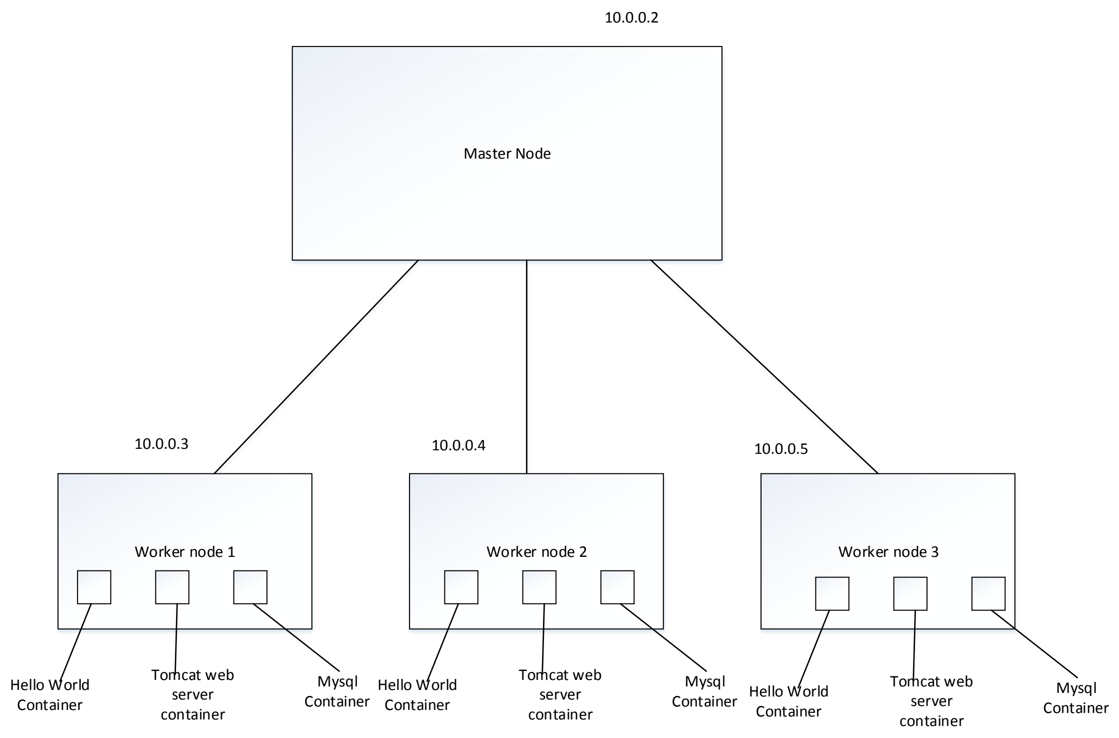


Figure 3.8: Application deployment scenario

The goal is to define a technology in the master node which deploy the containers in each working node simultaneously. Later it will be tested to deploy the applications in a specific working node from the master node (e.g. from master node to worker node 1).

d) **Auto scaling of application:**

This part will define a way which will create multiple instances of each application in every working node from the master node. In this work five instances will create for each application or containers. Figure 3.9 will explain the auto scaling function more precisely

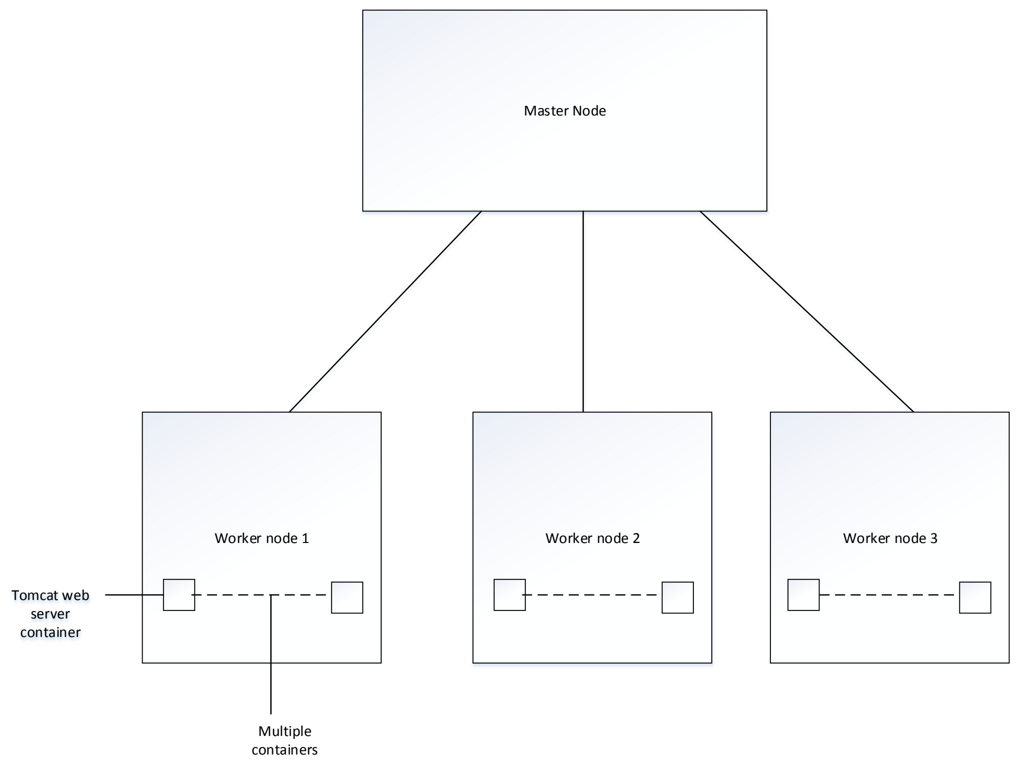


Figure 3.9: Auto scale scenario

e) Load balancing:

Compare the way of load balance of a service for both Kubernetes and Docker swarm is the goal. Docker swarm provides built in load balancing. On the other hand, enabling load balancing in Kubernetes requires manual configuration. In this implementation, the load balancer will work in master node. Figure 3.10 will explain the load balancer more precisely

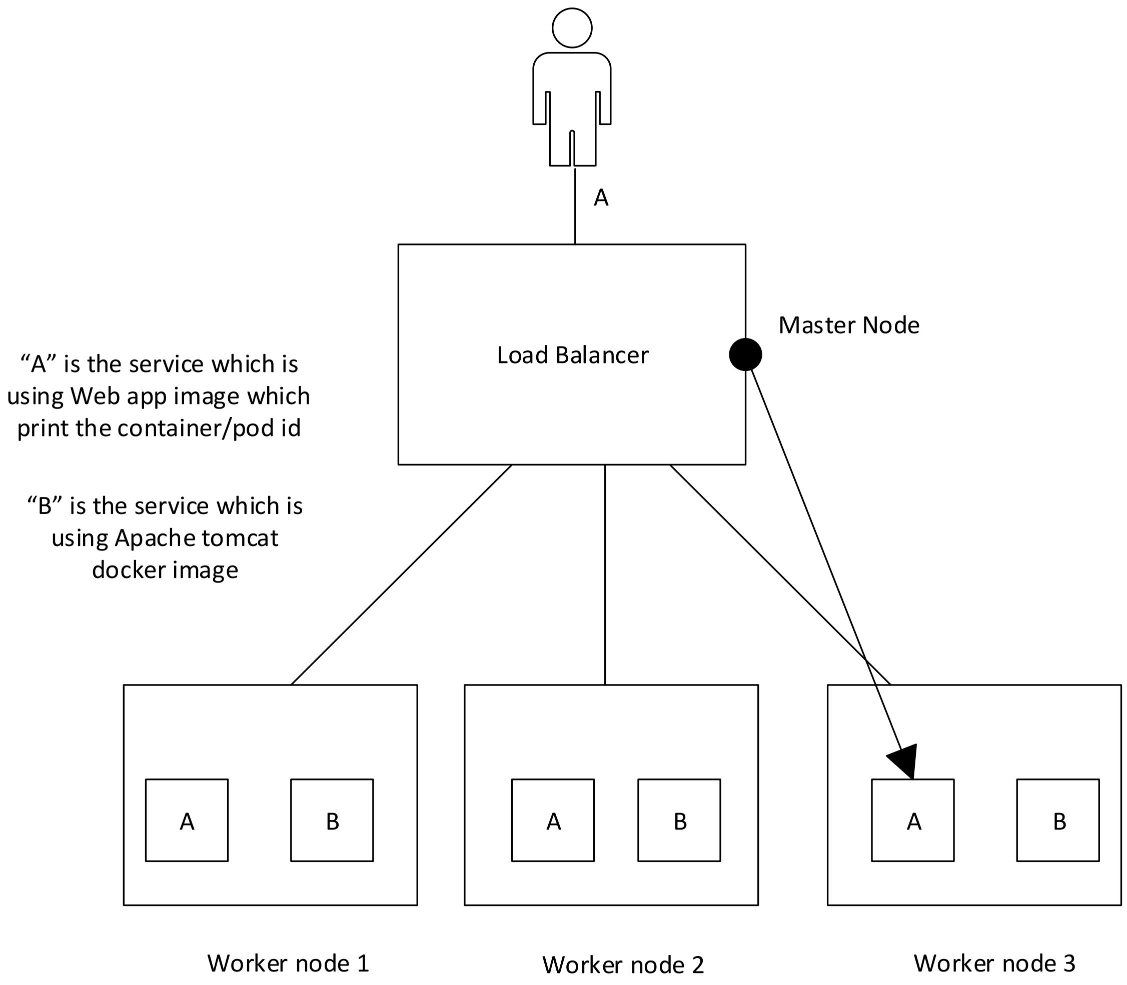

A
Load Balancer
Master Node
"A" is the service which is using Web app image which print the container/pod id
"B" is the service which is using Apache tomcat docker image
A
B
A
B
A
B
Worker node 1
Worker node 2
Worker node 3

Figure 3.10: Load balancing scenario

Each node will have A and B service. Every service will be exposed to the public network. If the user request for a service then the load balancer will decide from which node the user will get that service. For example, in the figure 3.8 the user request for service A and use the service from worker node 3.

f) Container update and rollback

In this section, the update process will be examined for both Docker swarm and Kubernetes. In the implementation, each working node will have a service based on apache tomcat image. Figure 3.11 will explain the service structure

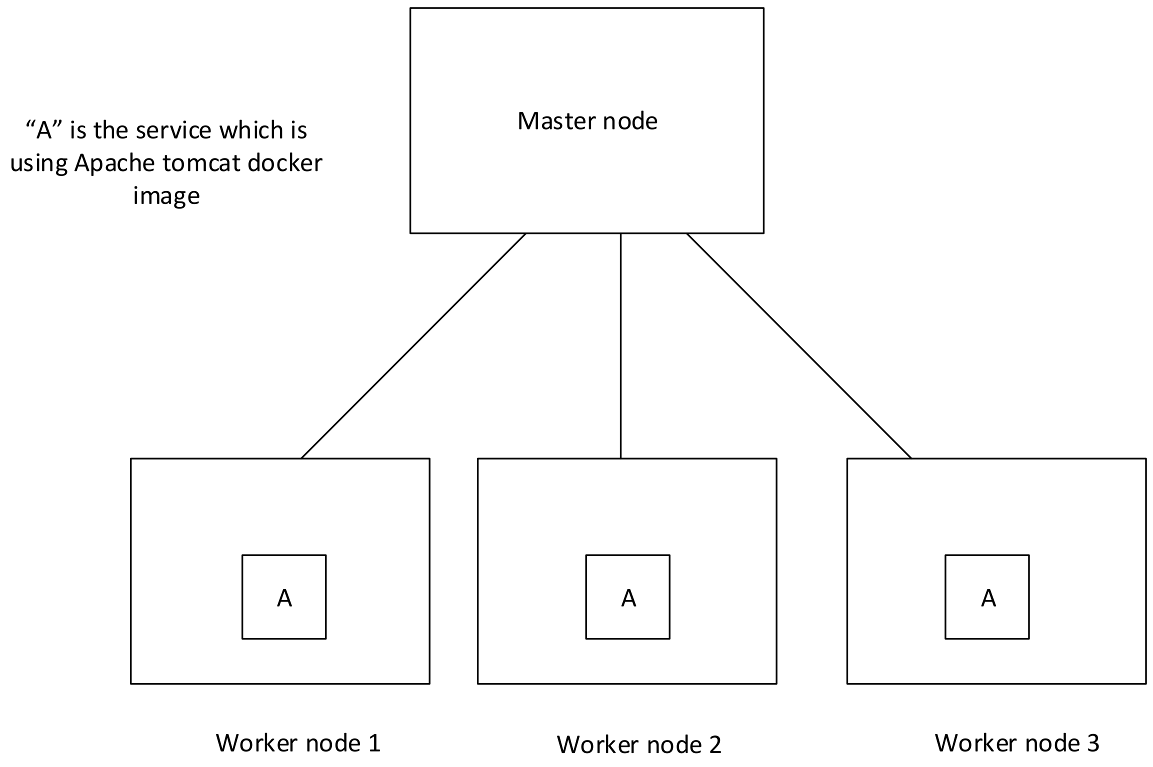

"A" is the service which is using Apache tomcat docker image
Master node
A
A
A
Worker node 1
Worker node 2
Worker node 3

Figure 3.11: Scenario for container update and rollback

Here the goal will be to update the service with new Docker image and to rollback to previous stage if anything goes wrong during update.

g) Live migration

Migration in container means to move a container from one working node to another. According to figure 3.8 each node has three containers. The migration process is explained in figure 3.12

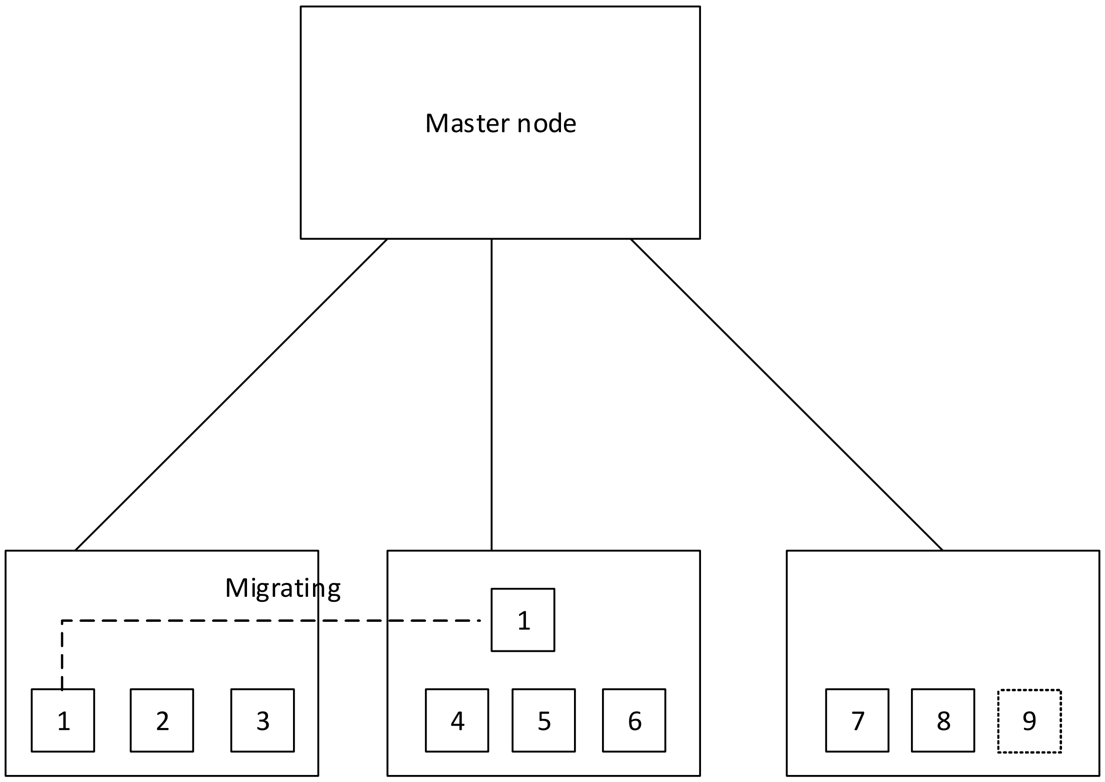


Figure 3.12: Live migration scenario

In the planned scenario, every working node will have three containers. The containers are numbered in figure 3.12 for understanding. The migration process will work in a way that the number 1 container from worker node 1 will move to worker node 2.

h) Performance analysis

For performance analysis, prime number counter in between 1 million will be used. It will be tested the condition of the node during running this application i.e the cpu usages, RAM usages etc. Also, performance will be measured for the mentioned application as a native application and as a container.

Beside this, all other points which will be found from other use cases (for example, deployment, scaling and so on) will be compared for both Docker swarm and Kubernetes.

3.6 Software Requirements

- Oracle VM VirtualBox (version 5.0.10r104061) for install virtual machine
- Ubuntu 14.04.5 server image for virtual machines
- Docker 17.03.1-ce
- Docker.io
- Docker 1.12.3
- Docker 1.13.0
- Docker Swarm
- Kubernetes v1.5.7
- Python
- Java
- YAML

4 Realization

This chapter will describe the implementation and testing of this thesis work in details. The following are the steps of the implementation of the thesis work.

- Prepare environment
- Network configuration for cluster
- Setting up Docker swarm cluster
- Setting up Kubernetes cluster
- Description of the containers
- Deployment of application in Docker swarm cluster
- Deployment of application in Kubernetes cluster
- Auto Scaling in Docker swarm
- Auto Scaling in Kubernetes
- Load balancing in Docker swarm
- Load balancing in Kubernetes
- Container update and rollback in Docker swarm
- Container update and rollback in Kubernetes
- Live migration
- Performance analysis
- Rancher

4.1 Prepare Environment

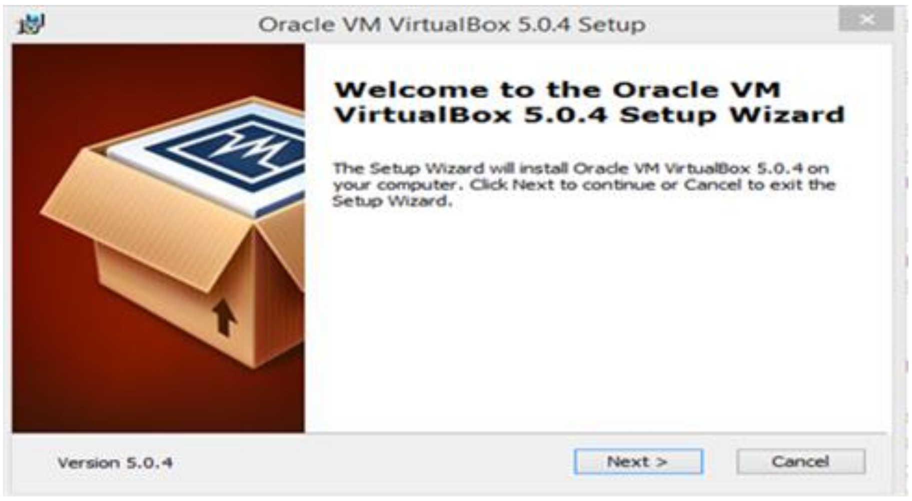
From chapter 3.4.1 and 3.4.2 it seen that in this implementation one Docker swarm cluster and one Kubernetes cluster will be used. Each cluster will have four machines where one machine will work as master and the other three will work as worker node. In this thesis work all machines will installed in Oracle virtual box. And Docker container will be used to deploy the containerized application in every cluster. To active the goal of this thesis work at first the necessary environment need to be established. To get the necessary environment the following steps has been followed

4.1.1 Software tools installation

There is certain basic software that are to be installed to start the thesis work. These tools will help to establish clusters of VMs and will be used to test use cases.

a) Installation of VirtualBox

VirtualBox is a cross-platform virtualization application. It allows to run more than one operating system at a time on the same host. Virtual Box is used as virtualization software to run many virtual machines. In this thesis work, compute nodes and SDN controller are virtual machines installed using VirtualBox. Depending on operating system, it is important to choose the right version of VirtualBox. This software can be downloaded from official website (VirtualBox, 2015). As it is to be run on a windows host machine so it important to choses latest version VirtualBox-5.*.exe file. Downloaded file is launched and installation prompt Figure 4.1 will guide the user to complete the installation process successfully.


Oracle VM VirtualBox 5.0.4 Setup

Welcome to the Oracle VM VirtualBox 5.0.4 Setup Wizard

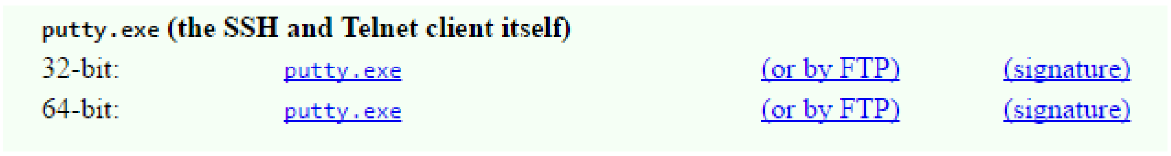
The Setup Wizard will install Oracle VM VirtualBox 5.0.4 on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Version 5.0.4

Figure 4.1: Virtual Box Installation Prompt

b) Installation of putty

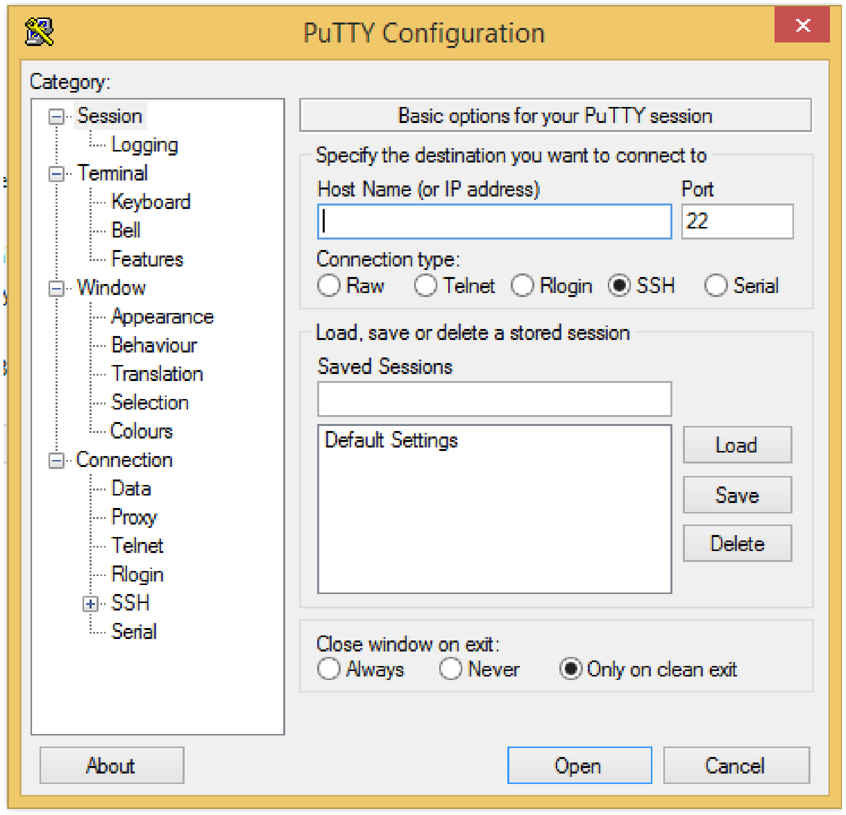
In this thesis work eight virtual machines need to be operate at the same time. So, it will be better to operate all the machines using secure shell from the host windows computer. PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers (Putty, 2017). As the host computer has 64-bit windows operating system so 64-bit putty.exe version has been used. Figure 4.2 will guide the user to download the putty successfully


putty.exe (the SSH and Telnet client itself)

32-bit:	putty.exe	(or by FTP)	(signature)
64-bit:	putty.exe	(or by FTP)	(signature)

Figure 4.2: Putty download prompt

After installing the putty, the appeared window has shown Figure 4.3



Specify the destination you want to connect to

Host Name (or IP address):

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

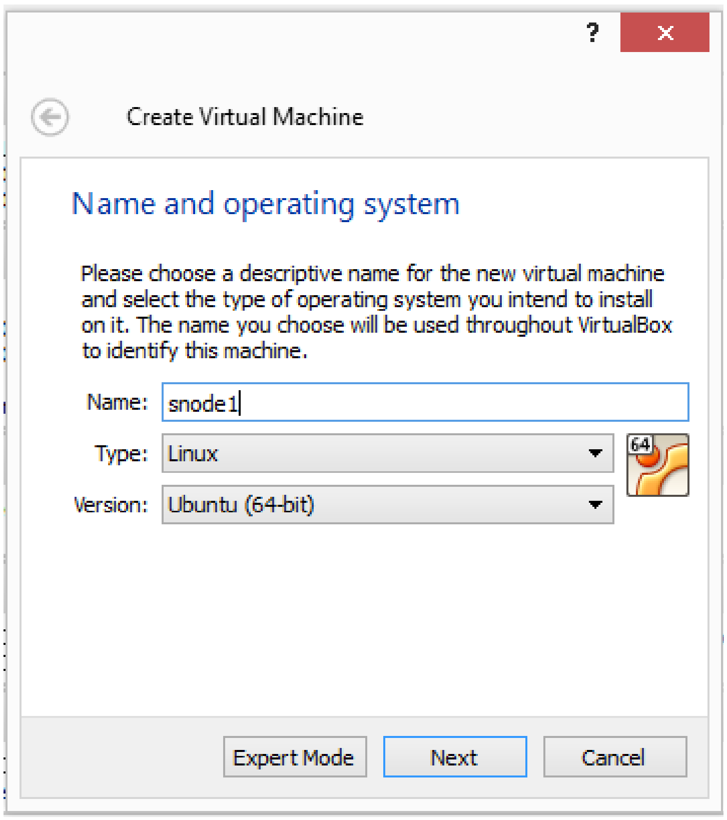
Figure 4.3: Putty window

4.1.2 Create Virtual Machine

In this thesis implementation total eight virtual machine will be used for creating both clusters. The following steps need to be followed to create a virtual machine. The steps of creating virtual machine is similar for all virtual machines.

a) Create virtual machine

To create a virtual machine at first the name must be defined and need to select the type of operating system what is going to use. The figure 4.4 explained it more precisely


Create Virtual Machine

Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: snode1

Type: Linux

Version: Ubuntu (64-bit)

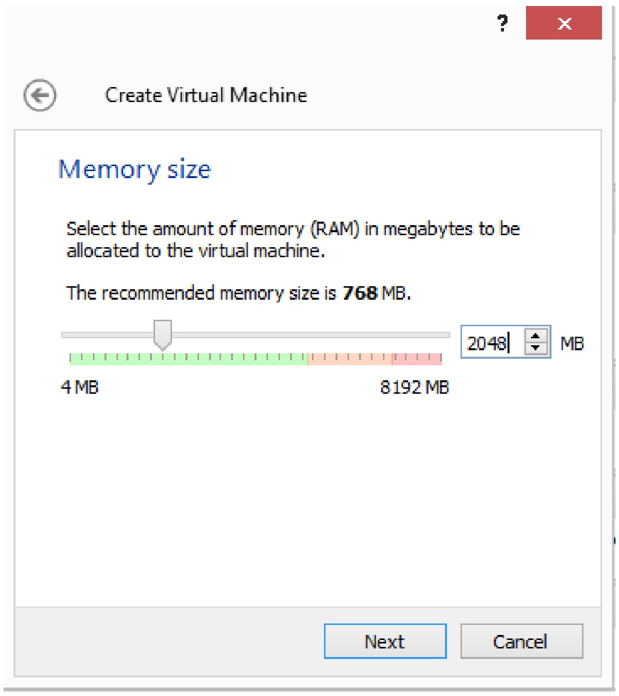
Figure 4.4: Name and operating system

All the virtual machines name, operating system, user name and password are given in below table 4.1

Table 4.1 : Necessary information of virtual machines

No	VM Name	User name	Password
1	Smaster	master	harami420
2	node1	snode1	harami420
3	snode2	snode2	harami420
4	snode3	snode3	harami420
5	kmaster	kmaster	harami420
6	knode1	knode1	harami420
7	knode2	knode2	harami420
8	knode3	knode3	harami420

It is mandatory to select the memory size (RAM) of all virtual machines. The figure 4.5 explained it more precisely



Create Virtual Machine

Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

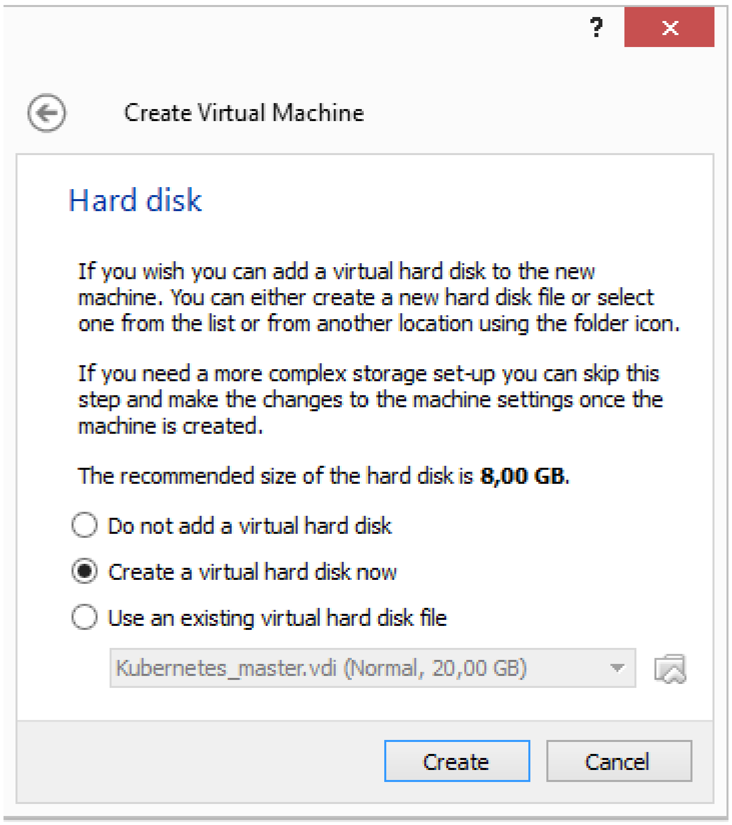
The recommended memory size is **768** MB.

2048 MB

Figure 4.5: Memory size

The memory size of all virtual machine has mentioned in table 3.1 and table 3.3

After that the virtual hard drive need to select. The below figure 4.6 explains it precisely



Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **8,00 GB.**

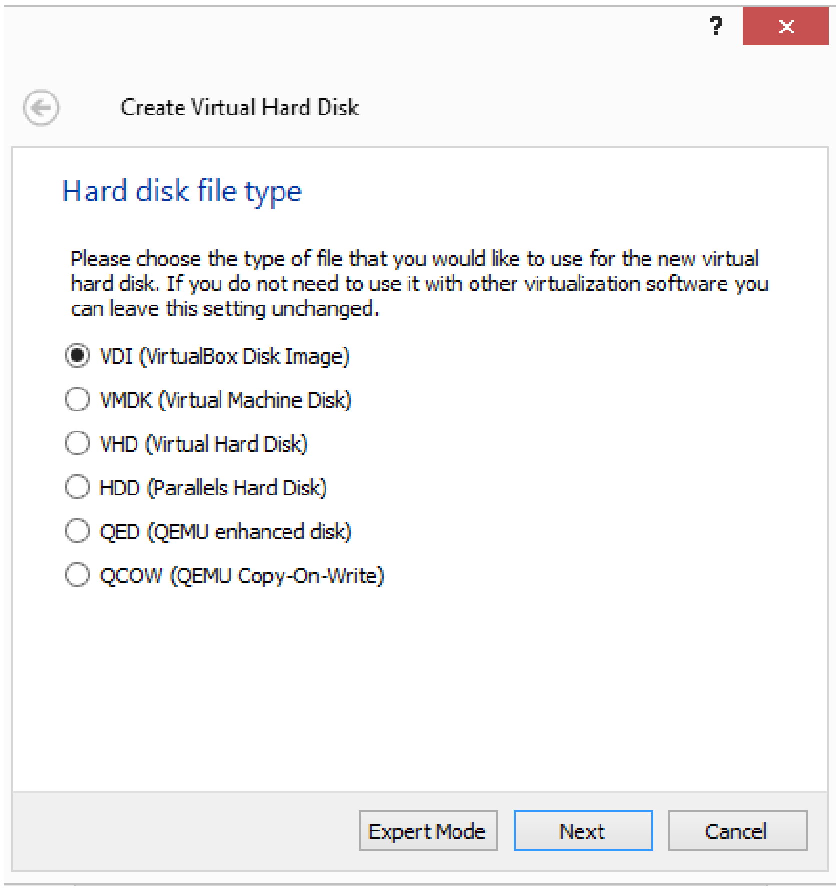
- ○ Do not add a virtual hard disk
- ● Create a virtual hard disk now
- ○ Use an existing virtual hard disk file

Kubernetes_master.vdi (Normal, 20,00 GB)

Figure 4.6: Virtual hard drive

b) Select Hard disk type

In this implementation VirtualBox Disk Image has been used, so it is important to select the image type as VirtualBox Disk Image. The figure 4.7 explained it precisely



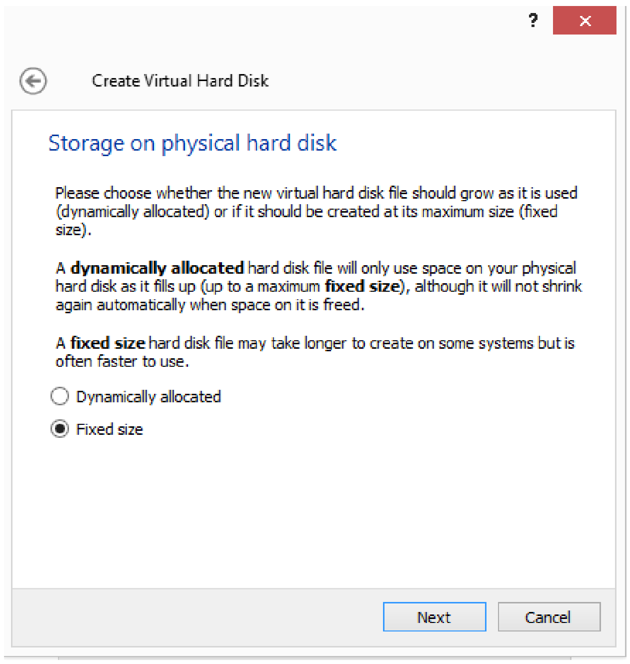
Hard disk file type

Please choose the type of file that you would like to use for the new virtual hard disk. If you do not need to use it with other virtualization software you can leave this setting unchanged.

- [✓ VDI (VirtualBox Disk Image)]
- VMDK (Virtual Machine Disk)
- VHD (Virtual Hard Disk)
- HDD (Parallels Hard Disk)
- QED (QEMU enhanced disk)
- QCOW (QEMU Copy-On-Write)

Figure 4.7: Hard disk file type

After that storage size need to be select. In this implementation fixed storage size is used. The storage size for each virtual machine is available in Table 3.1 and 3.3. Figure 4.8 explained it more precisely.



Create Virtual Hard Disk

Storage on physical hard disk

Please choose whether the new virtual hard disk file should grow as it is used (dynamically allocated) or if it should be created at its maximum size (fixed size).

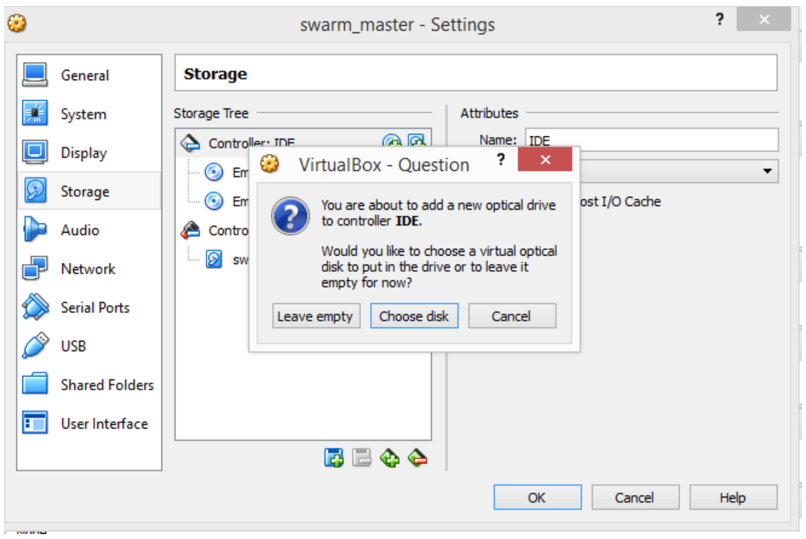
A **dynamically allocated** hard disk file will only use space on your physical hard disk as it fills up (up to a maximum **fixed size**), although it will not shrink again automatically when space on it is freed.

A **fixed size** hard disk file may take longer to create on some systems but is often faster to use.

Dynamically allocated [✓ Fixed size]

Figure 4.8: Hard disk size

Beside this there are few more setting need to be changed before starts the virtual machine. The storage option is available in settings of the virtual machine. It is important to choose the image in storage as shown in figure 4.9



swarm_master - Settings

Storage

Name: IDE

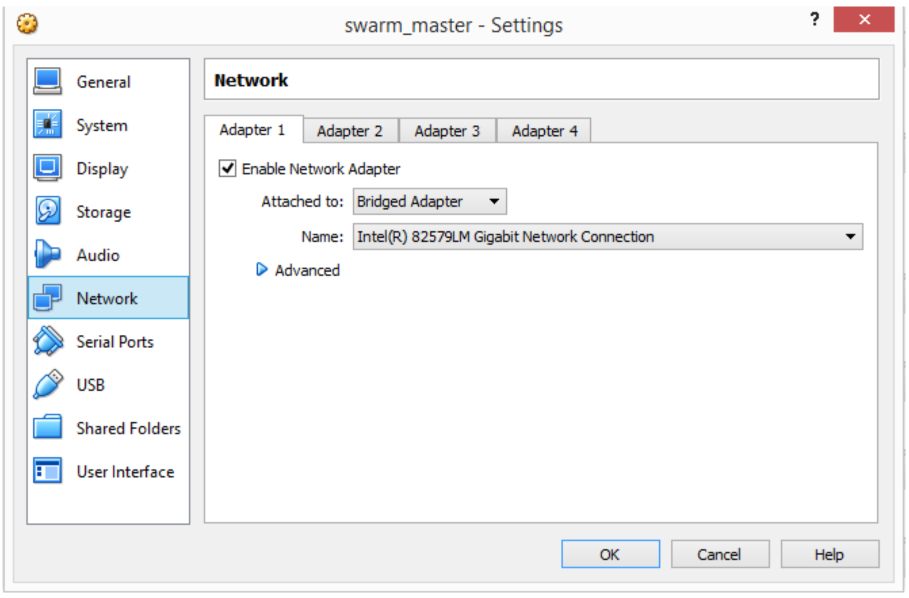
VirtualBox - Question

You are about to add a new optical drive to controller **IDE**.

Would you like to choose a virtual optical disk to put in the drive or to leave it empty for now?

Figure 4.9: Choose disk in storage

Also, as explained in Networking of cluster (section 3.5) there will be an internal network for establishing the cluster. Beside this each virtual machine will have an interface to communicate with the internet. So, each virtual machine should have two network adapters. The eth0 of each virtual machine will be a bridge interface so that the user can access the internet through this interface. On the other hand, eth1 of every virtual machine will be the part of the internal network. From the perspective of this implementation there are two internal networks, one is for Docker swarm cluster and the another is for Kubernetes cluster. Figure 4.10 and 4.11 explained it more precisely.



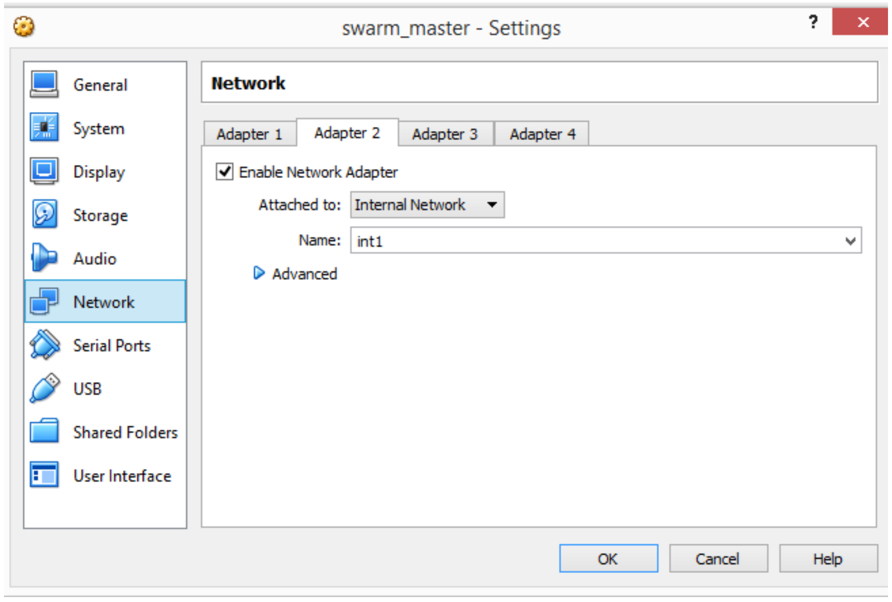
Network

☑ Enable Network Adapter

Attached to: Bridged Adapter

Name: Intel(R) 82579LM Gigabit Network Connection

Figure 4.10: Network adapter of bridge network



Network

☑ Enable Network Adapter

Attached to: Internal Network

Name: int1

Figure 4.11: Network adapter of internal network

4.2 Network Configuration for Cluster

As explained in section 3.5 and 4.1.3 it is necessary to create the internal network for establishing the cluster. To do this in every virtual machine the eth1 is needed to configure into a static IP address. To do this the following changes must do in `/etc/network/interfaces`

```
auto eth1

iface eth1 inet static
```

```
address 10.0.0.2

netmask 255.255.255.0
```

The above network configuration was done in the master node of Docker swarm cluster. Similarly, the network interfaces need to be configured in other nodes too. Necessary IP plan for both Docker swarm cluster and Kubernetes cluster is given in table 3.2 and 3.4. After finishing the network configuration in every node, it is important to check the internal network by running the following command

```
ping -I 10.0.0.2 10.0.0.3
```

Here 10.0.0.2 is the source IP address and 10.0.0.3 is the destination IP address. This same command is applicable for each node in both clusters. In this implementation, every node has a bridge adapter to access the outside world. This interface is important to use the application from outside. This is explained in below figure 4.12

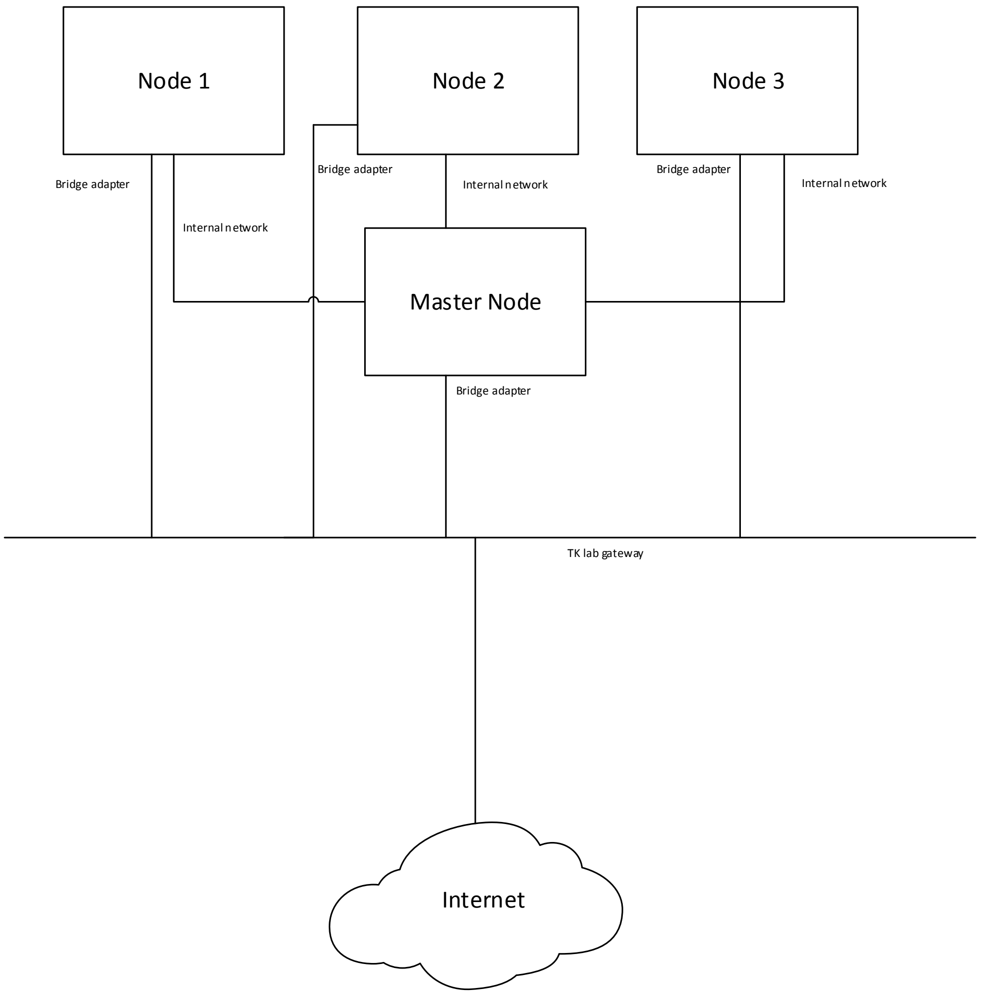


Figure 4.12: Network scenario in detail

In this implementation, there are two clusters as similar shown in figure 4.12. One is for Docker swarm and the other is for Kubernetes.

4.3 Setting up Docker swarm cluster

To setting up the Docker cluster at first Docker need to install in all node. In this thesis implementation, Docker v17.06 has used. Also, the Docker community edition has used. To install the Docker the following steps, need to follow.

a) Uninstall the old version

Older versions of Docker were called Docker or Docker-engine. If these are installed then need to uninstall those

```
$ sudo apt-get remove Docker Docker-engine
```

b) Recommended extra packages for Trusty 14.04

To install the extra package

```
$ sudo apt-get update
$ sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

c) Set up the repository

To install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

To add Docker's official GPG key:

```
$ curl -fsSL https://download.Docker.com/linux/ubuntu/gpg | sudo apt-key add
-
```

To verify that the key fingerprint is 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88

```
$ sudo apt-key fingerprint 0EBFCD88
```

To use the following command to set up the stable repository.

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.Docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

d) Install Docker

To update the apt package index.

```
$ sudo apt-get update
```

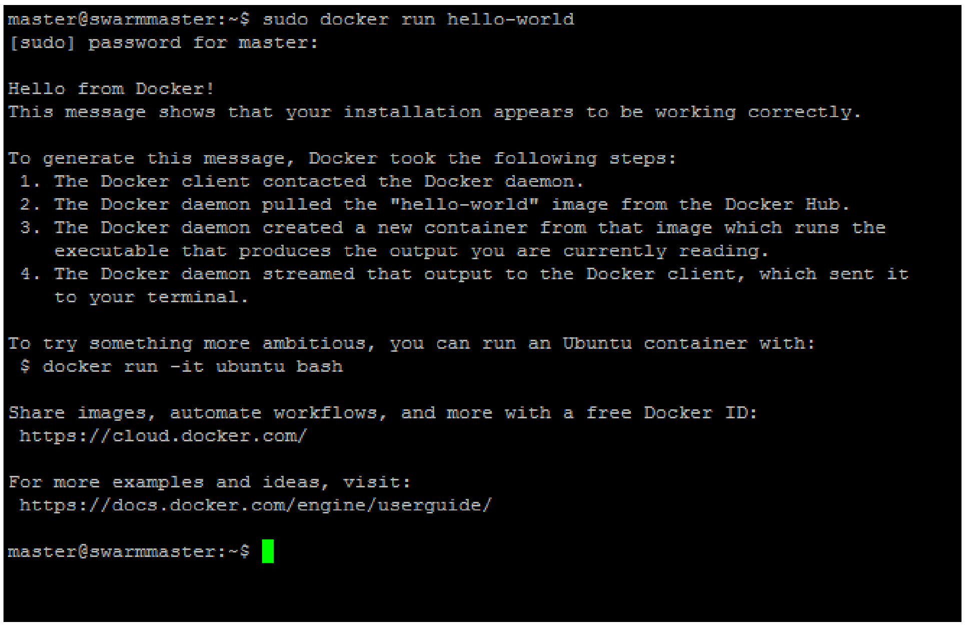
Use the following command to install the latest version of Docker

```
$ sudo apt-get install docker-ce
```

For testing the Docker by running the following command

```
$ sudo docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits. Figure 4.13 explain the result precisely



```
master@swarmmaster:~$ sudo docker run hello-world
[sudo] password for master:

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

master@swarmmaster:~$
```

Figure 4.13: Docker test

e) Create Docker swarm cluster

From figure 3.4 it is shown that in Docker swarm cluster there will be one master node and three worker nodes. All the nodes are in same network so that they can communicate with each other. Creating network has explained in section 4.2. From Table 3.2 the IP address of the master node is 10.0.0.2. It is important to keep the port 2377, 4789 and 7946 open before establishing the swarm mode. To see the port status the following command will be used

```
$ sudo netstat -ntlp | grep LISTEN
```

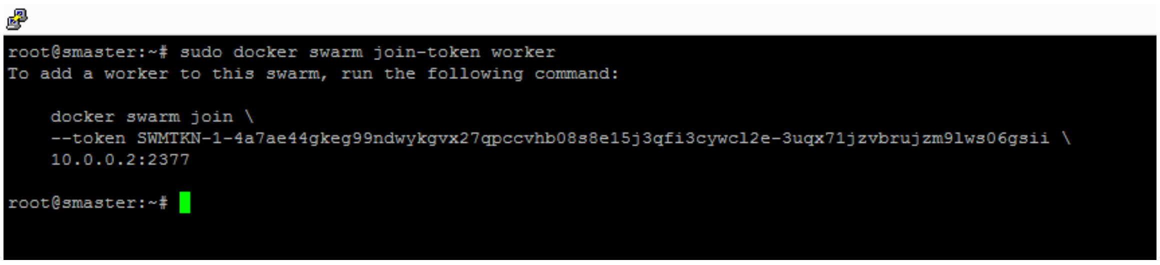
To create the swarm mode in master node the following command need to follow

```
$ sudo docker swarm init –advertise-addr 10.0.0.2 –listen-addr 10.0.0.2:2377
```

Here 10.0.0.2 is the IP of the master node. This command has created the swarm cluster and make the node a master. To add a worker node in this cluster the following command need to run in master node

```
$ sudo docker swarm join-token worker
```

This command will give a result as shown in figure 4.14



```
root@smaster:~# sudo docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-4a7ae44gkeg99ndwykgvx27qpccvhb08s8e15j3qfi3cywcl2e-3uqx71jzvbrujzm9lws06gsii \
    10.0.0.2:2377

root@smaster:~#
```

Figure 4.14: Swarm join command

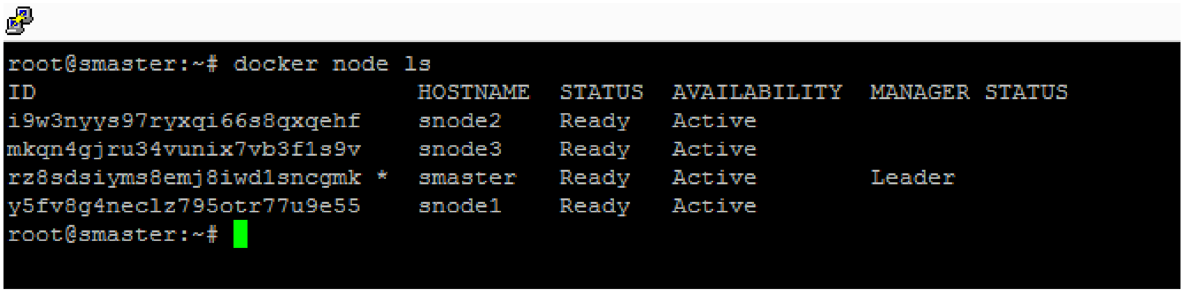
To add the worker node in this cluster it is necessary to run the following command in each worker node

```
$ docker swarm join \
    --token     SWMTKN-1-4a7ae44gkeg99ndwykgvx27qpccvhb08s8e15j3qfi3cywcl2e-3uqx71jzvbrujzm9lws06gsii \
    10.0.0.2:2377
```

In this implementation, it need to run in every three worker nodes. After that the swarm cluster has created. To check the cluster status run the following command in master node

```
$ sudo docker node ls
```

This command shows all nodes of the cluster including the ID, HOSTNAME, STATUS, AVAILABILITY and MANAGER STATUS. The cluster status has shown in Figure 4.15



```
root@smaster:~# docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
i9w3nyys97ryxqi66s8qxqehf    snode2    Ready   Active
mkqn4gjru34vunix7vb3f1s9v    snode3    Ready   Active
rz8sdsiyms8emj8iwd1sncgmk *  smaster   Ready   Active        Leader
y5fv8g4neclz795otr77u9e55    snode1    Ready   Active
root@smaster:~#
```

Figure 4.15: Swarm cluster status

4.4 Setting up Kubernetes cluster

From figure 3.4 it is seen that in Kubernetes cluster there are one master node and three worker nodes. In Kubernetes cluster worker node known as minions. In Kubernetes cluster, all nodes must be in same network. The IP plan for Kubernetes cluster is given in table 3.4. The installation procedure is described in below

- To switch in root user

  ```
  $ sudo -s
  ```

- To install pre-requisite software packages

  ```
  $ apt-get update
  $ apt-get install ssh
  ```

```
$ apt-get install Docker.io
$ apt-get install curl
```

- Password-less ssh login setup, accept all the default parameters in the prompt of the below command

```
$ ssh-keygen -t rsa
```

- Copy the ssh id_rsa key
- In case this command fails, then need to use alternative solution to add the key

```
$ cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys
```

- To validate the password-less ssh-login

```
$ ssh root@172.168.10.2
```

- To check the ssh login from master node to the other node of the cluster

```
$ ssh knode1@172.168.10.3
$ ssh knode2@172.168.10.4
$ ssh knode3@172.168.10.5
```

- To get the Kubernetes release bundle from the official github repository

```
$wget          https://github.com/GoogleCloudPlatform/Kubernetes/re-
leases/download/v1.0.1/Kubernetes.tar.gz
```

- To get the Kubernetes release bundle from the official github repository

```
$ tar -xvf Kubernetes.tar.gz
```

- To build the binaries of Kubernetes code specially for ubuntu cluster

```
$ cd Kubernetes/cluster/ubuntu
```

- To execute the shell script

```
$ ./build.sh
```

- To configure the cluster information by editing the following parameters of the file `cluster/ubuntu/config-default.sh`

```
export     nodes=${nodes:-"root@172.168.10.2     knode1@172.168.10.3
knode2@172.168.10.4 knode3@172.168.10.5"}
roles=${roles:-"a i i i"}
export NUM_MINIONS=${NUM_MINIONS:-3}
```

In roles “a” means the master and “i” means the minions.

- To start the cluster with the following command

```
$ cd Kubernetes/cluster
$ KUBERNETES_PROVIDER=ubuntu ./kube-up.sh
```

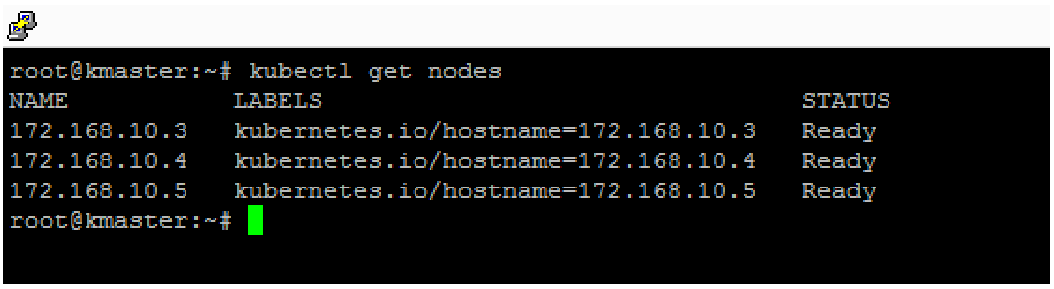
- To add kubectl binary to manage the Kubernetes cluster

```
$ export PATH=$PATH:~/Kubernetes/cluster/ubuntu/binaries
```

- To check the nodes by following command

```
kubectl get nodes
```

It will show all the nodes of the cluster as like figure 4.16



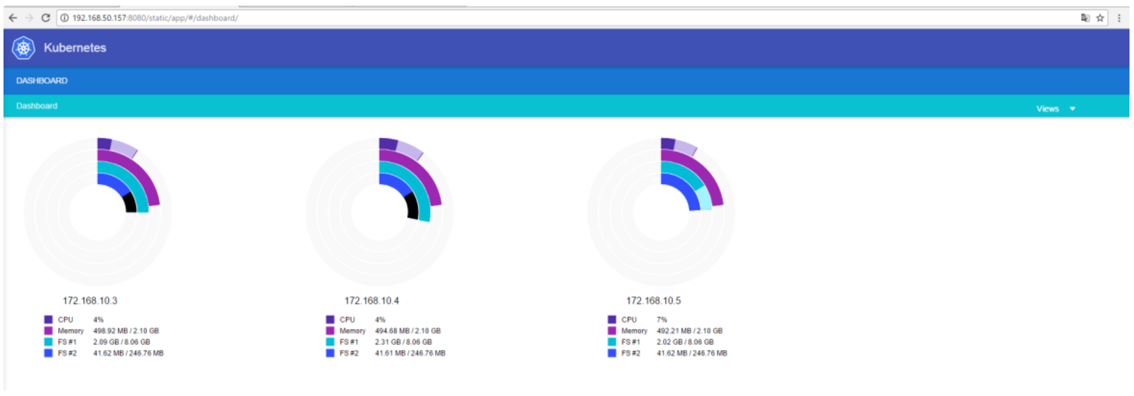
```
root@kmaster:~# kubectl get nodes
NAME           LABELS                                  STATUS
172.168.10.3   kubernetes.io/hostname=172.168.10.3     Ready
172.168.10.4   kubernetes.io/hostname=172.168.10.4     Ready
172.168.10.5   kubernetes.io/hostname=172.168.10.5     Ready
root@kmaster:~#
```

Figure 4.16: Kubernetes cluster status

To use the dashboard of the Kubernetes the following link can be used

```
192.158.50.157:8080/ui
```

The dashboard of the Kubernetes is like figure 4.17



172.168.10.3		172.168.10.4		172.168.10.5	
CPU	4%	CPU	4%	CPU	7%
Memory	498.92 MB / 2.10 GB	Memory	494.68 MB / 2.10 GB	Memory	492.21 MB / 2.10 GB
FS #1	2.09 GB / 8.06 GB	FS #1	2.31 GB / 8.06 GB	FS #1	2.02 GB / 8.06 GB
FS #2	41.62 MB / 246.76 MB	FS #2	41.61 MB / 246.76 MB	FS #2	41.62 MB / 246.76 MB

Figure 4.17: Kubernetes dashboard

4.5 Description of the Containers

In this thesis work the following four applications has used

- A Web application printing the container/pod id
- Web server based on Apache tomcat
- Database server based on MySQL
- Prime number counter which counts all prime number between 1 million

In below a description is available about this all applications

4.5.1 A web application printing the container/pod id

This application prints the container/pod id along with “Hello-World” message. To use the application as a container the container need to define with a Docker file named `Dockerfile`. The source code of the `Dockerfile` is given below

# Use an official Python runtime as a parent image

```
FROM python:2.7-slim
```

```
# Set the working directory to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
ADD . /app
# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python", "app.py"]
```

This Dockerfile refers to couple of things which has not created yet, namely `app.py` and `requirements.txt`. It is important to keep these files in the same folder where the `Dockerfile` has been. The source code `requirements.txt` file is given below

```
Flask
Redis
```

The source code app.py is given below

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket
# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2,
socket_timeout=2)
app = Flask(__name__)
@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
            "<b>Hostname:</b> {hostname}<br/>" \
            "<b>Visits:</b> {visits}"
    return      html.format(name=os.getenv("NAME",      "world"),      host-
name=socket.gethostname(), visits=visits)


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

The `app.py` will print the environment variable NAME, as well as the output of a call to `socket.gethostname()`. As the `Redis` is not running so it is expected that the attempt to use it will produce the error message. To create the Docker image

```
docker build -t friendlyhello .
```

This will create an image named `friendlyhello`. To use this image further it is important to push this image in Docker hub. To do this log in to the Docker public registry on the local machine

```
docker login
```

After that tag the image

```
docker tag image username/repository:tag
```

In this case

```
docker tag friendlyhello tanzeemzillu/tanzeemzillu:test
```

After that push the image to the Docker hub

```
docker push tanzeemzillu/tanzeemzillu:test
```

4.5.2 Web server based on Apache tomcat

Apache Tomcat (or simply Tomcat) is an open source web server and servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java Servlet and the Java Server Pages (JSP) specifications from Oracle, and provides a "pure Java" HTTP web server environment for Java code to run in. In the simplest config Tomcat runs in a single operating system process. The process runs a Java virtual machine (JVM). Every single HTTP request from a browser to Tomcat is processed in the Tomcat process in a separate thread (tomcat, 2017). In this project, the official tomcat container has been used which provided in Docker hub.

4.5.3 Database server based on MySQL

MySQL is a opensource database server. It is largely used for web based application. MySQL has become the leading database choice for web-based applications, covering the entire range from personal projects and websites, via e-commerce and information services (mysql, 2017). In this project, the official mysql container has been used which provided in Docker hub.

4.5.4 Prime number counter which counts all prime number between 1 million

This application will count all the prime numbers between 1 to 1000000. To create this application as Docker containerized application the following steps need to follow

- Create a directory named `primenumber`

```
mkdir primenumber
```

- Create a file named `primenumber.java`. The source code of this file is in below

```
import java.util.Date;
public class PrimeNumber {
      public static void main(String[] args) {
             Date d1= new Date();
           System.out.println("Printing prime number from 1 to 10000000");
            for(int number = 2; number<=10000000; number++){
                //print prime numbers only
                if(isPrime(number)){
                    System.out.println(number);
                }
            }
            Date d2=new Date();
            System.out.println("Total time to run the application: "+(d2.getTime()-d1.getTime())/1000);
            }


          /*
           * Prime number is not divisible by any number other than 1 and itself
           * @return true if number is prime
           */
          public static boolean isPrime(int number){
              for(int i=2; i<number; i++){
                 if(number%i == 0){
                     return false; //number is divisible so its not prime
                 }
              }
              return true; //number is prime now
          }
}
```

- Create the `Dockerfile` with the following code

```
FROM java:8
COPY . /var/www/java
WORKDIR /var/www/java
RUN javac Hello.java
CMD ["java", "PrimeNumber"]
```

- Build the Docker image

  ```
  sudo docker build -t primenumber .
  ```

  This command creates a Docker image named `primenumber`

To use this image further it is important to push this image in Docker hub. To do this log in to the Docker public registry on the local machine

```
docker login
```

After that tag the image

```
docker tag image username/repository:tag
```

In this case it is

```
docker tag image tanzeemzillu/tanzeemzillu:primenumber
```

After that push the image to the Docker hub

```
docker push tanzeemzillu/tanzeemzillu:primenumber
```

These four containers used in various use cases in this thesis work.

4.6 Deployment of Application in Docker Swarm Cluster

According to figure 3.7 a web container which printing container/pod id, tomcat server, MySQL server need to deploy in every node of Docker swarm cluster.

4.6.1 Deploy web application printing the container/pod id

It is possible to deploy this container in every node by creating a global service in Docker swarm cluster. Global service runs on each active node in the swarm (Docker service create, 2017). Even if an administrator will add another node into the cluster then the service automatically created if the service is in global mode. For this application, the following command need to run

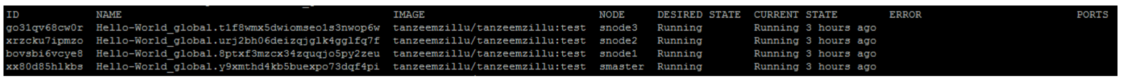
```
docker service create --name Hello-world_global -p 81:80 --mode global tanzeemzillu/tanzeemzillu:test
```

This command will deploy containers in every node of the cluster. The output can be seen by the following command

```
docker service ps Hello-world_global
```

So, a service has created named `Hello-world_global` which has one container in every node of the cluster. Also, the service exposed in port `81` for using it outside of the cluster.

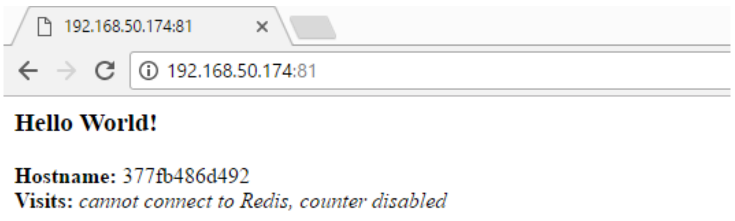
The output is like figure 4.18



```
ID            NAME                                          IMAGE                               NODE     DESIRED STATE  CURRENT STATE          ERROR                              PORTS
go31qv68cw0r  Hello-World_global.t1f8wmx5dwiomseo1s3nwop6w  tanzeemzillu/tanzeemzillu:test  snode3   Running        Running 3 hours ago
xrzcku7ipmzo  Hello-World_global.urj2bh06deizqjglk4gglfq7f  tanzeemzillu/tanzeemzillu:test  snode2   Running        Running 3 hours ago
bovsbi6vcye8  Hello-World_global.8ptxf3mzcx34zquqjo5py2zeu  tanzeemzillu/tanzeemzillu:test  snode1   Running        Running 3 hours ago
xx80d85hlkbs  Hello-World_global.y9xmthd4kb5buexpo73dqf4pi  tanzeemzillu/tanzeemzillu:test  smaster  Running        Running 3 hours ago
```

Figure 4.18: Hello-world_global service

It is seen that each node has the container and all are in running state. It is possible to access the application by using the bridge IP of any machine. For example, if user want to use from node1 then http://192.168.50.174:81 need to access. Here 192.168.50.174 is the bridge IP for node1. The output of the application is like figure 4.19



Hello World!

Hostname: 377fb486d492
Visits: *cannot connect to Redis, counter disabled*

Figure 4.19: Hello-world_global service output

Here the hostname is the container id.

4.6.2 Deploy Tomcat Container in Docker Swarm Cluster

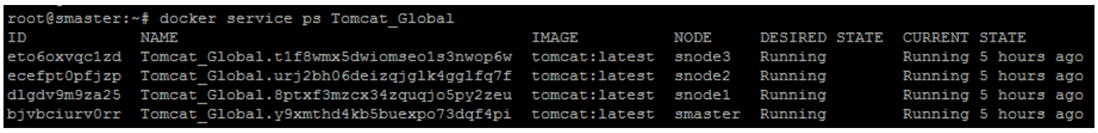
To deploy tomcat container in every node of the cluster a global service need to create. This can be done by using the following command

```
docker service create --name Tomcat_Global --publish 8082:8080 --mode glob-
al tomcat
```

To check the output of the service

```
docker service ps Tomcat_Global
```

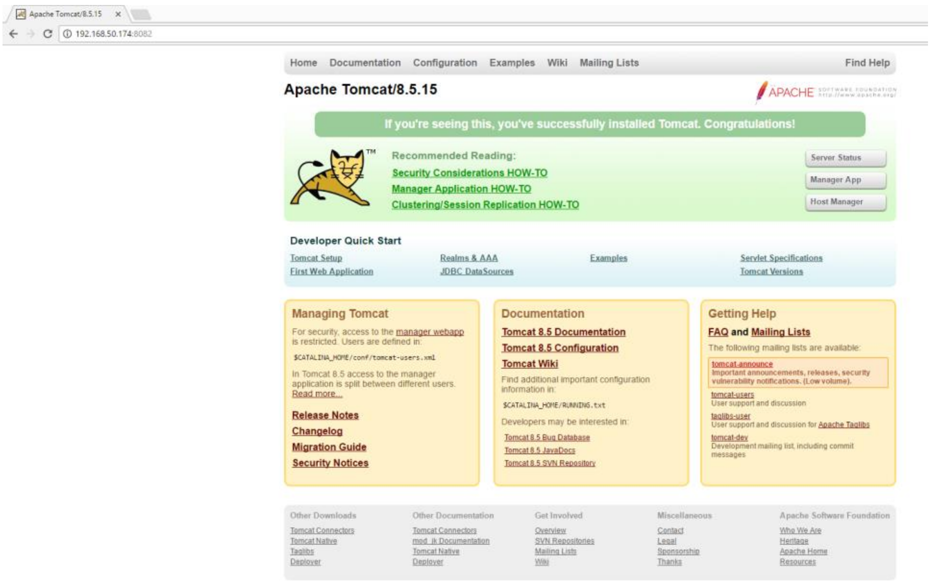
The output will be like figure 4.20



```
root@smaster:~# docker service ps Tomcat_Global
ID            NAME                                         IMAGE          NODE     DESIRED STATE  CURRENT STATE
eto6oxvqc1zd  Tomcat_Global.t1f8wmx5dwiomseo1s3nwop6w  tomcat:latest  snode3   Running        Running 5 hours ago
ecefpt0pfjzp  Tomcat_Global.urj2bh06deizqjglk4gglfq7f  tomcat:latest  snode2   Running        Running 5 hours ago
dlgdv9m9za25  Tomcat_Global.8ptxf3mzcx34zquqjo5py2zeu  tomcat:latest  snode1   Running        Running 5 hours ago
bjvbciurv0rr  Tomcat_Global.y9xmthd4kb5buexpo73dqf4pi  tomcat:latest  smaster  Running        Running 5 hours ago
```

Figure 4.20: Tomcat_Global service

As like the previous application this can be access from any node. For example, to access from node 1 the user need to browse `http://192.168.50.174:8082`. The user will see the welcome page of Apache Tomcat server as shown in figure 4.21



Apache Tomcat/8.5.15

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Developer Quick Start

Managing Tomcat

For security, access to the manager webapp is restricted. Users are defined in:

```
$CATALINA_HOME/conf/tomcat-users.xml
```

In Tomcat 8.5 access to the manager application is split between different users. Read more...

Documentation

Find additional important configuration information in:

```
$CATALINA_HOME/RUNNING.txt
```

Developers may be interested in:

Getting Help

FAQ and Mailing Lists

The following mailing lists are available:

tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).

tomcat-users
User support and discussion

taglibs-user
User support and discussion for Apache Taglibs

tomcat-dev
Development mailing list, including commit messages

Figure 4.21: Tomcat_Global service output

4.6.3 Deploy MySQL Container in Docker Swarm Cluster

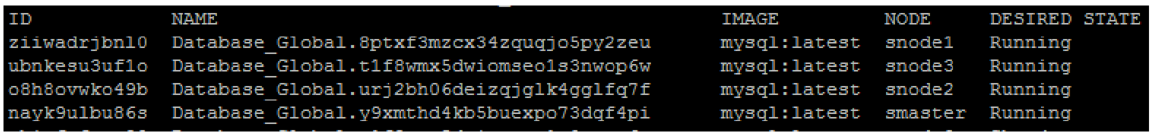
To deploy MySQL container in every node of the cluster a global service need to create. This can be done by using the following command

```
docker service create --name Database_Global --mode global -e
MYSQL_ROOT_PASSWORD=my-secret-pw mysql
```

To check the output of the service

```
docker service ps Database_Global
```

The output will be like figure 4.22



```
ID            NAME                                          IMAGE         NODE     DESIRED STATE
ziiwadrjbnl0  Database_Global.8ptxf3mzcx34zquqjo5py2zeu     mysql:latest  snode1   Running
ubnkesu3uf1o  Database_Global.t1f8wmx5dwiomseo1s3nwop6w     mysql:latest  snode3   Running
o8h8ovwko49b  Database_Global.urj2bh06deizqjglk4gglfq7f     mysql:latest  snode2   Running
nayk9ulbu86s  Database_Global.y9xmthd4kb5buexpo73dqf4pi     mysql:latest  smaster  Running
```

Figure 4.22: Database_Global service

It is seen that MySQL server is running in every node. If the user wants to use MySQL in a specific node the he/she needs to run the following command in that specific node

```
docker exec -it "container id" mysql -uroot -p
```

This command will connect the user with the console of the MySQL. In this project "`my-secret-pw`" has been used as password.

4.7 Deployment of Application in Kubernetes Cluster

According to figure 3.7 a web container which printing container/pod id, tomcat server, MySQL server need to deploy in every node of Docker swarm cluster.

4.7.1 Deploy web application printing the container/pod id

In Kubernetes, it is possible to deploy the container in a specific node. In this implementation, all containers deployed in every node based on node label. Containers are deployed based on the label node. Here nodes are labeled as `node1`, `node2` and `node3`. The node can be labeled by the following command

```
kubectl label nodes 172.168.10.3 node=node1
```

Here `172.168.10.3` is the name of first node. Second and third node is addressed as `172.68.10.4` and `172.168.10.5` respectively. The second and third node is labeled by the following command

```
kubectl label nodes 172.168.10.4 node=node2

kubectl label nodes 172.168.10.5 node=node3
```

In this implementation replication controller created and then it exposed by creating corresponding service. For deploying hello-world container in node1 `hello-world-node1.yaml` created in master node. The source code of this file is

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello-world-node1
spec:
  replicas: 1
  selector:
    app: hello-world-node1
  template:
    metadata:
      name: hello-world-node1
      labels:
        app: hello-world-node1
    spec:
      containers:
      - name: hello-world-node1
        image: tanzeemzillu/tanzeemzillu:test
        ports:
        - containerPort: 80
      nodeSelector:
        node: node1
```

To create the replication controller

```
kubectl create -f hello-world-node1.yaml
```

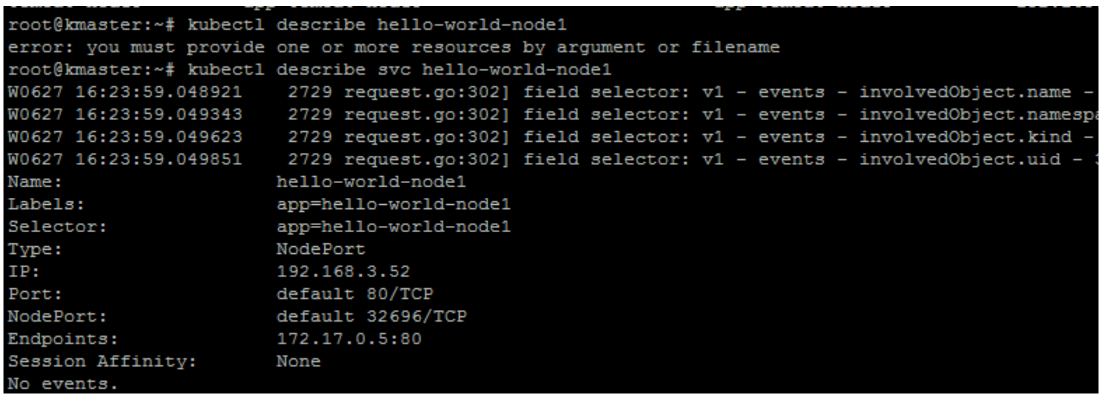
This command will create the replication controller named `hello-world-node1.yaml`. To create the service based on this replication controller

```
kubect expose rc hello-world-node1 --type=NodePort
```

To check the service

```
kubectl describe hello-world-node1
```

The output is like figure 4.20



```
root@kmaster:~# kubectl describe hello-world-node1
error: you must provide one or more resources by argument or filename
root@kmaster:~# kubectl describe svc hello-world-node1
W0627 16:23:59.048921    2729 request.go:302] field selector: v1 - events - involvedObject.name -
W0627 16:23:59.049343    2729 request.go:302] field selector: v1 - events - involvedObject.namesp
W0627 16:23:59.049623    2729 request.go:302] field selector: v1 - events - involvedObject.kind -
W0627 16:23:59.049851    2729 request.go:302] field selector: v1 - events - involvedObject.uid -
Name:                   hello-world-node1
Labels:                 app=hello-world-node1
Selector:               app=hello-world-node1
Type:                   NodePort
IP:                     192.168.3.52
Port:                   default 80/TCP
NodePort:               default 32696/TCP
Endpoints:              172.17.0.5:80
Session Affinity:       None
No events.
```

Figure 4.23: hello-world-node1 service

From figure 4.20 it is shown that this service has been exposed by `32696` port. As service type is declared as `NodePort` which eventually exposed the service in a random but not repeatable port. So to use this service user need to use `192.168.50.83:32696.` Similarly, this container can be deployed in both node2 and node3. The only change is in the respective yaml file. The `hello-world-node2.yaml` is using for the service in node2. The source code is given below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello-world-node2
spec:
  replicas: 1
  selector:
    app: hello-world-node2
  template:
    metadata:
      name: hello-world-node2
      labels:
        app: hello-world-node2
    spec:
      containers:
```

```
      - name: hello-world-node2
        image: tanzeemzillu/tanzeemzillu:test
        ports:
        - containerPort: 80
      nodeSelector:
        node: node2
```

The source code of `hello-world-node3.yaml` is given below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello-world-node3
spec:
  replicas: 1
  selector:
    app: hello-world-node3
  template:
    metadata:
      name: hello-world-node3
      labels:
        app: hello-world-node3
    spec:
      containers:
      - name: hello-world-node3
        image: tanzeemzillu/tanzeemzillu:test
        ports:
        - containerPort: 80
      nodeSelector:
        node: node3
```

The url for `hello-world-node2` and `hello-world-node3` is respectively `192.168.50.86:31356` and `192.168.50.87:30218`

4.7.2 Deploy Tomcat Container in Kubernetes Cluster

As described in section 4.7.1 replication controller created in every node to deploy the service based on tomcat container. For node1 `tomcat-node1.yaml` is used. The source code of this file is

```
apiVersion: v1
```

```
kind: ReplicationController
metadata:
  name: tomcat-node1
spec:
  replicas: 1
  selector:
    app: tomcat-node1
  template:
    metadata:
      name: tomcat-node1
      labels:
        app: tomcat-node1
    spec:
      containers:
      - name: tomcat-node1
        image: tomcat
        ports:
        - containerPort: 8080
      nodeSelector:
        node: node1
```

`tomcat-node2` is using for node2. The source code is

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: tomcat-node2
spec:
  replicas: 1
  selector:
    app: tomcat-node2
  template:
    metadata:
      name: tomcat-node2
      labels:
        app: tomcat-node2
    spec:
      containers:
      - name: tomcat-node2
        image: tomcat
        ports:
        - containerPort: 8080
      nodeSelector:
        node: node2
```

`tomcat-node3` is using for node3. The source of this file is in below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: tomcat-node3
spec:
  replicas: 1
  selector:
    app: tomcat-node3
  template:
    metadata:
      name: tomcat-node3
      labels:
        app: tomcat-node3
    spec:
      containers:
      - name: tomcat-node3
        image: tomcat
        ports:
        - containerPort: 8080
      nodeSelector:
        node: node3
```

To create the replication controller for respective node

```
kubectl create -f tomcat-node1.yaml
kubectl create -f tomcat-node2.yaml
kubectl create -f tomcat-node3.yaml
```

To create the service for respective node

```
kubectl expose rc tomcat-node1 --type=NodePort

kubectl expose rc tomcat-node2 --type=NodePort

kubectl expose rc tomcat-node3 --type=NodePort
```

To use these services the following urls need to browse for node1, node2, node3 respectively

```
http://192.168.50.83:31018

http://192.168.50.86:30181

http://192.168.50.87:30524
```

4.7.3 Deploy MySQL Container in Kubernetes Cluster

As described in section 4.4.4 and 4.4.5 replication controller created in every node to deploy the service based on MySQL container. For node1 `mysql-node1.yaml` is used. The source code of this file is in below

```
kind: ReplicationController
metadata:
  name: my-mysql-v1
  namespace: default
  labels:
    name: my-mysql
    version: v1
spec:
  replicas: 1
  selector:
    name: my-mysql
    version: v1
  template:
    metadata:
      labels:
        name: my-mysql
        version: v1
    spec:
      containers:
        - image: Docker.io/mysql
          name: my-mysql-v1
          ports:
            -  containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: root
      nodeSelector:
        node: node1
```

For node1 `mysql-node2.yaml` is used. The source code of this file is in below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-mysql-v2
  namespace: default
  labels:
    name: my-mysql
    version: v2
spec:
  replicas: 1
  selector:
    name: my-mysql
    version: v2
  template:
    metadata:
      labels:
        name: my-mysql
        version: v2
    spec:
      containers:
        - image: Docker.io/mysql
          name: my-mysql-v2
          ports:
            -  containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: root
      nodeSelector:
        node: node2
```

For node1 `mysql-node3.yaml` is used. The source code of this file is in below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-mysql-v3
  namespace: default
  labels:
    name: my-mysql
    version: v3
spec:
  replicas: 1
  selector:
    name: my-mysql
    version: v3
  template:
    metadata:
      labels:
        name: my-mysql
        version: v3
    spec:
      containers:
        - image: Docker.io/mysql
          name: my-mysql-v3
          ports:
            -  containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: root
      nodeSelector:
        node: node3
```

To create the replication controller for respective node

```
kubectl create -f mysql-node1.yaml
kubectl create -f mysql-node2.yaml
kubectl create -f mysql-node3.yaml
```

To create the service for respective node

```
kubectl expose rc mysql-node1 --type=NodePort

kubectl expose rc mysql-node2 --type=NodePort

kubectl expose rc mysql-node3 --type=NodePort
```

If the user wants to use MySQL in a specific node the he/she needs to run the following command in that specific node

```
docker exec -it "container id" mysql -uroot -p
```

In this case `"root"` is using as password.

4.8 Auto Scaling in Docker Swarm Cluster

In this implementation, a service Tomcat is created which scale the tomcat container 8 times. To do this the following command has used

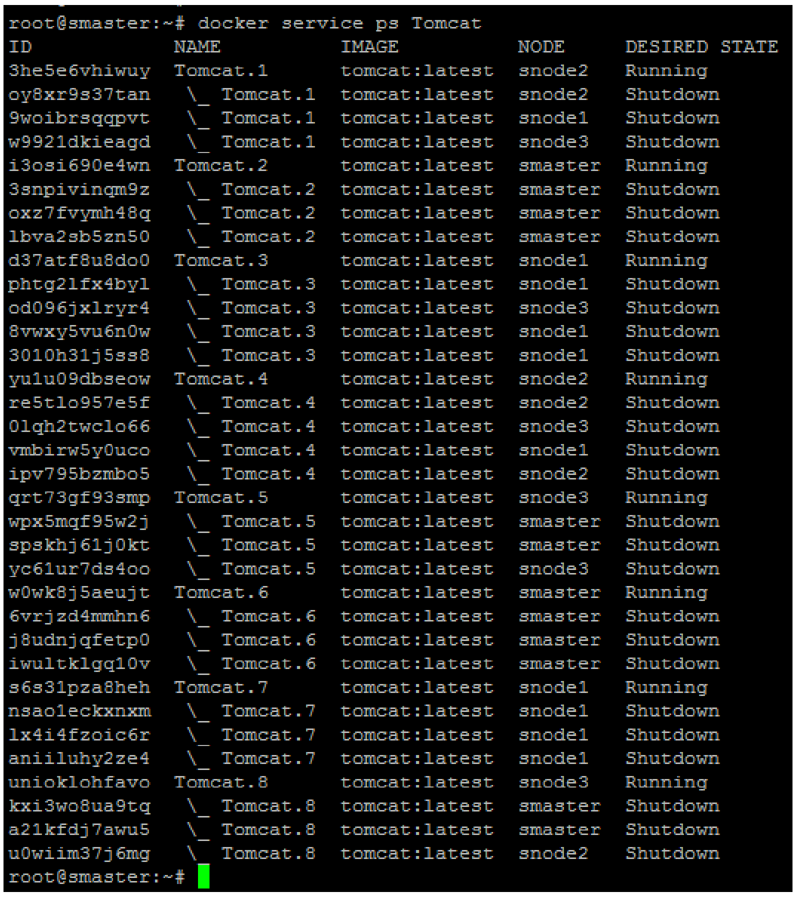
```
docker service create --name Tomcat --publish 8081:8080 --replicas 8 tomcat
```

Please note that, in Docker swarm it is not possible to scale the global service. In this situation, replicated service must be used.

To see the output

```
docker service ps Tomcat
```

The output will like figure 4.21



```
root@smaster:~# docker service ps Tomcat
ID            NAME          IMAGE          NODE     DESIRED STATE
3he5e6vhiwuy  Tomcat.1      tomcat:latest  snode2   Running
oy8xr9s37tan   \_ Tomcat.1  tomcat:latest  snode2   Shutdown
9woibrsqqpvt   \_ Tomcat.1  tomcat:latest  snode1   Shutdown
w9921dkieagd   \_ Tomcat.1  tomcat:latest  snode3   Shutdown
i3osi690e4wn  Tomcat.2      tomcat:latest  smaster  Running
3snpivinqm9z   \_ Tomcat.2  tomcat:latest  smaster  Shutdown
oxz7fvymh48q   \_ Tomcat.2  tomcat:latest  smaster  Shutdown
lbva2sb5zn50   \_ Tomcat.2  tomcat:latest  smaster  Shutdown
d37atf8u8do0  Tomcat.3      tomcat:latest  snode1   Running
phtg2lfx4byl   \_ Tomcat.3  tomcat:latest  snode1   Shutdown
od096jxlryr4   \_ Tomcat.3  tomcat:latest  snode3   Shutdown
8vwxy5vu6n0w   \_ Tomcat.3  tomcat:latest  snode1   Shutdown
3010h31j5ss8   \_ Tomcat.3  tomcat:latest  snode1   Shutdown
yu1u09dbseow  Tomcat.4      tomcat:latest  snode2   Running
re5tlo957e5f   \_ Tomcat.4  tomcat:latest  snode2   Shutdown
01qh2twclo66   \_ Tomcat.4  tomcat:latest  snode3   Shutdown
vmbirw5y0uco   \_ Tomcat.4  tomcat:latest  snode1   Shutdown
ipv795bzmbo5   \_ Tomcat.4  tomcat:latest  snode2   Shutdown
qrt73gf93smp  Tomcat.5      tomcat:latest  snode3   Running
wpx5mqf95w2j   \_ Tomcat.5  tomcat:latest  smaster  Shutdown
spskhj61j0kt   \_ Tomcat.5  tomcat:latest  smaster  Shutdown
yc61ur7ds4oo   \_ Tomcat.5  tomcat:latest  snode3   Shutdown
w0wk8j5aeujt  Tomcat.6      tomcat:latest  smaster  Running
6vrjzd4mmhn6   \_ Tomcat.6  tomcat:latest  smaster  Shutdown
j8udnjqfetp0   \_ Tomcat.6  tomcat:latest  smaster  Shutdown
iwultklgq10v   \_ Tomcat.6  tomcat:latest  smaster  Shutdown
s6s31pza8heh  Tomcat.7      tomcat:latest  snode1   Running
nsaoleckxnxm   \_ Tomcat.7  tomcat:latest  snode1   Shutdown
lx4i4fzoic6r   \_ Tomcat.7  tomcat:latest  snode1   Shutdown
aniiluhy2ze4   \_ Tomcat.7  tomcat:latest  snode1   Shutdown
unioklohfavo  Tomcat.8      tomcat:latest  snode3   Running
kxi3wo8ua9tq   \_ Tomcat.8  tomcat:latest  smaster  Shutdown
a21kfdj7awu5   \_ Tomcat.8  tomcat:latest  smaster  Shutdown
u0wiim37j6mg   \_ Tomcat.8  tomcat:latest  snode2   Shutdown
root@smaster:~#
```

Figure 4.24: Tomcat service state after scale

From figure 4.24 it is seen that 8 tomcat containers created and it is distributed equally among all nodes i.e. 2 containers per node.

Beside this to scale a running service the following command need to run

```
docker service scale Tomcat=8
```

4.9 Auto Scaling in Kubernetes Cluster

In Kubernetes, it is possible to scale container in a specific node. In this implementation, tomcat container scale in node1. To create the replication controller `tomcat-node1-scale.yaml` file is using. The source code of this file is in below

```
apiVersion: v1

kind: ReplicationController

metadata:

  name: tomcat-node1

spec:

  replicas: 2
  selector:
    app: tomcat-node1
  template:
    metadata:
      name: tomcat-node1
      labels:
        app: tomcat-node1
    spec:
      containers:
      - name: tomcat-node1
        image: tomcat
        ports:
        - containerPort: 8080
      nodeSelector:
        node: node1
```

In this code, it is seen that the `replicas` are 2. So, it creates 2 pods in `node1` which are using tomcat container. To create the replication controller

```
kubectl create -f tomcat-node1-scale.yaml
```

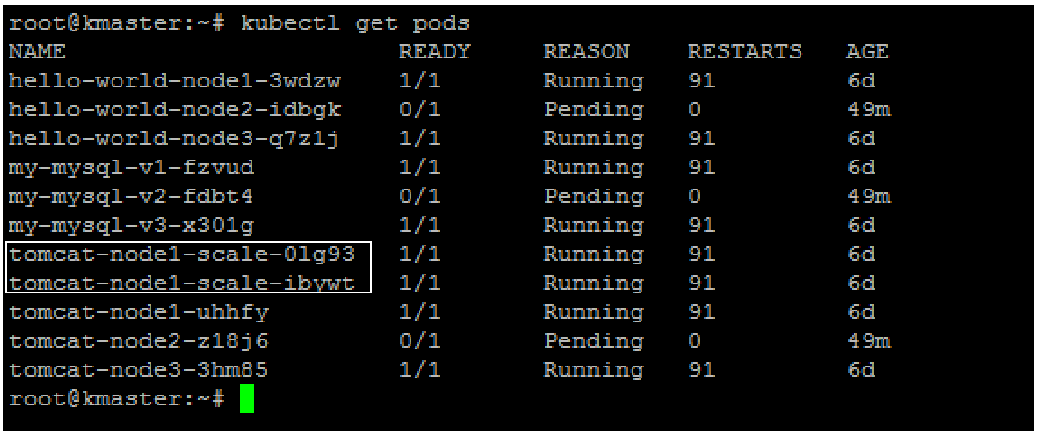
To create the service

```
kubectl expose rc tomcat-node1-scale --type=NodePort
```

To check the pods status

```
kubectl get pods
```

The output will be like below figure 4.25



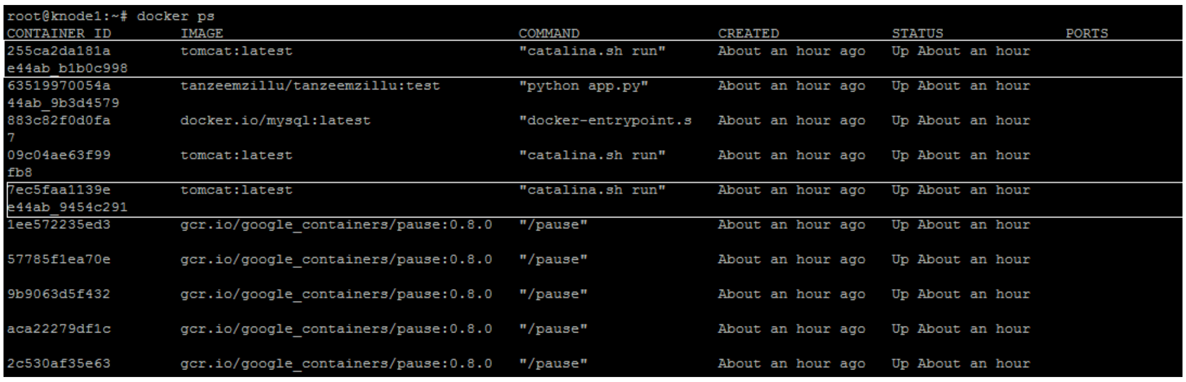
```
root@kmaster:~# kubectl get pods
NAME                        READY     REASON    RESTARTS   AGE
hello-world-node1-3wdzw     1/1       Running   91         6d
hello-world-node2-idbgk     0/1       Pending   0          49m
hello-world-node3-q7z1j     1/1       Running   91         6d
my-mysql-v1-fzvud           1/1       Running   91         6d
my-mysql-v2-fdbt4           0/1       Pending   0          49m
my-mysql-v3-x301g           1/1       Running   91         6d
tomcat-node1-scale-01g93    1/1       Running   91         6d
tomcat-node1-scale-ibywt    1/1       Running   91         6d
tomcat-node1-uhhfy          1/1       Running   91         6d
tomcat-node2-z18j6          0/1       Pending   0          49m
tomcat-node3-3hm85          1/1       Running   91         6d
root@kmaster:~#
```

Figure 4.25: Pod status

From figure 4.25 it is seen that 2 pods created for this service (indicated by white box). User also can check in node1 by the following command

```
Docker ps
```

The output will be like below figure 4.26



```
root@knode1:~# docker ps
CONTAINER ID        IMAGE                                  COMMAND               CREATED             STATUS              PORTS
255ca2da181a        tomcat:latest                          "catalina.sh run"     About an hour ago   Up About an hour
e44ab_b1b0c998
63519970054a        tanzeemzillu/tanzeemzillu:test         "python app.py"       About an hour ago   Up About an hour
44ab_9b3d4579
883c82f0d0fa        docker.io/mysql:latest                 "docker-entrypoint.s  About an hour ago   Up About an hour
7
09c04ae63f99        tomcat:latest                          "catalina.sh run"     About an hour ago   Up About an hour
fb8
7ec5faa1139e        tomcat:latest                          "catalina.sh run"     About an hour ago   Up About an hour
e44ab_9454c291
1ee572235ed3        gcr.io/google_containers/pause:0.8.0   "/pause"              About an hour ago   Up About an hour

57785f1ea70e        gcr.io/google_containers/pause:0.8.0   "/pause"              About an hour ago   Up About an hour

9b9063d5f432        gcr.io/google_containers/pause:0.8.0   "/pause"              About an hour ago   Up About an hour

aca22279df1c        gcr.io/google_containers/pause:0.8.0   "/pause"              About an hour ago   Up About an hour

2c530af35e63        gcr.io/google_containers/pause:0.8.0   "/pause"              About an hour ago   Up About an hour
```

Figure 4.26: Container status in node 1

It is seen that there are two container running corresponding `tomcat-node1-scale` service. (indicated by white box). Similarly, it is possible to scale in other nodes also.

It is also possible to scale a running service. For example, in this case the following command will scale the pods

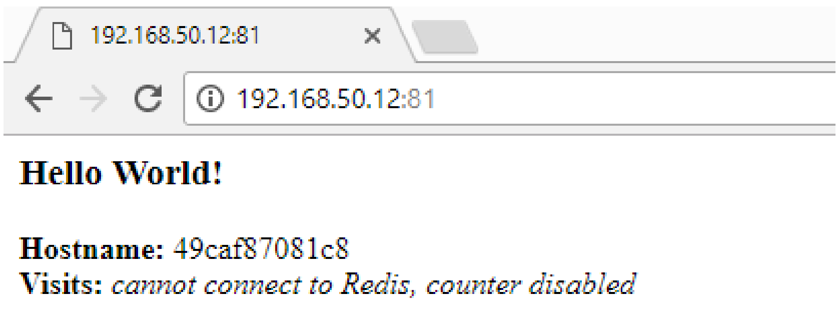
```
kubectl scale --replicas=2 rc tomcat-node1-scale
```

4.10 Load Balancing in Docker Swarm

According to figure 3.10 it is seen that a container is scaled in multiple copies throughout the cluster. The main target of load balancing is when user request for a web service then the request will distribute among the cluster and the user get response from any of the container. It is very useful to balance the load and to maintain the service.

The swarm manager uses ingress load balancing to expose the services user want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort (range 30000-32767) or user can configure a PublishedPort for the service. External components, such as cloud load balancers, can access the service on the PublishedPort of any node in the cluster whether the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance. Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service (Services and tasks, 2017).

In this implementation hello-world container has been used. In section 4.5.1 it is shown that how the user can create a global service of hello-world container and exposed it in port 81. So now this container has deployed in every node in the cluster. This service already using the internal load balancing facilities of Docker swarm. To test this user can browse the following address http://192.168.50.12:81. The response will be like below figure 4.27

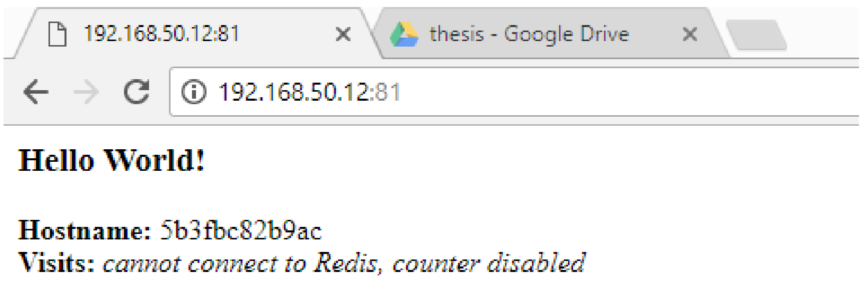


Hello World!

Hostname: 49caf87081c8
Visits: *cannot connect to Redis, counter disabled*

Figure 4.27: Response of Hello-world_global service

Now if the user refreshes the page then he will get the response from a different container. Please see the out in figure 4.28



Hello World!

Hostname: 5b3fbc82b9ac
Visits: *cannot connect to Redis, counter disabled*

Figure 4.28: Response of Hello-world_global service after refresh

4.11 Load Balancing in Kubernetes

As like Docker swarm Kubernetes also has built in load-balancing. So, in Kubernetes a user can request for a service which will contain distributed pods to maintain the load of the service. As like 4.10 here also hello-world container will use in the implementation. To achieve this the hello-world-loadbalancer.yaml used for creation replication controller. The source code of hello-world-loadbalancer.yaml are in below

```
apiVersion: v1

kind: ReplicationController

metadata:

  name: hello-world-node1

spec:

  replicas: 3

  selector:

    app: hello-world-node1

  template:

    metadata:
      name: hello-world-node1
      labels:
        app: hello-world-node1
    spec:
      containers:
      - name: hello-world-node1
        image: tanzeemzillu/tanzeemzillu:test
        ports:
        - containerPort: 80
```

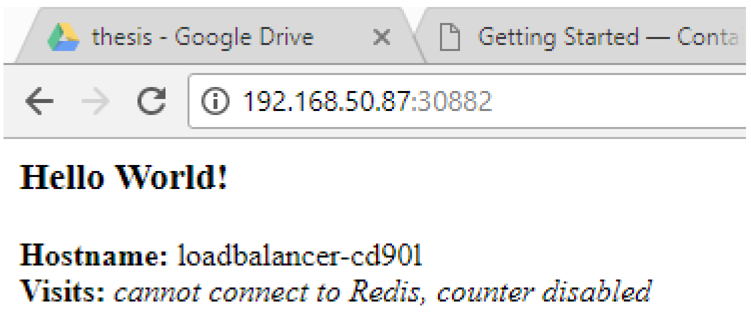
To create the replication controller

```
kubectl create -f hello-world-loadbalancer.yaml
```

To use this the replication controller need to expose as a service

```
kubectl expose hello-world-loadbalancer --type=NodePort
```

To use the service user can browse in `192.168.50.87:30882`. The output will be like figure 4.29

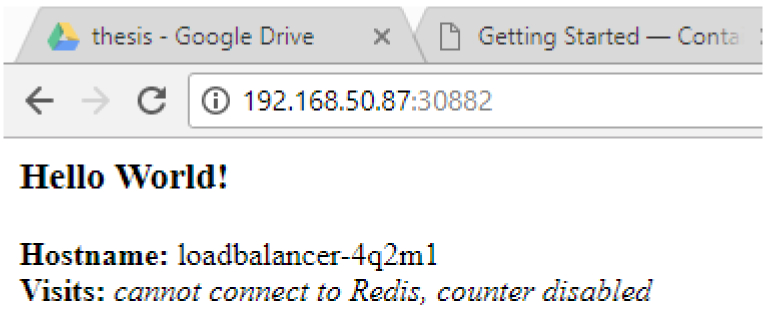


Hello World!

Hostname: loadbalancer-cd901
Visits: *cannot connect to Redis, counter disabled*

Figure 4.29: Response of Hello-world service in Kubernetes

If the user will try to use the same service once again then the output is like below figure 4.30



Hello World!

Hostname: loadbalancer-4q2m1
Visits: *cannot connect to Redis, counter disabled*

Figure 4.30: Response of Hello-world service in Kubernetes after refresh

In this case, the type of the service is NodePort and every node allocated with that port. This is explained in below figure 4.31

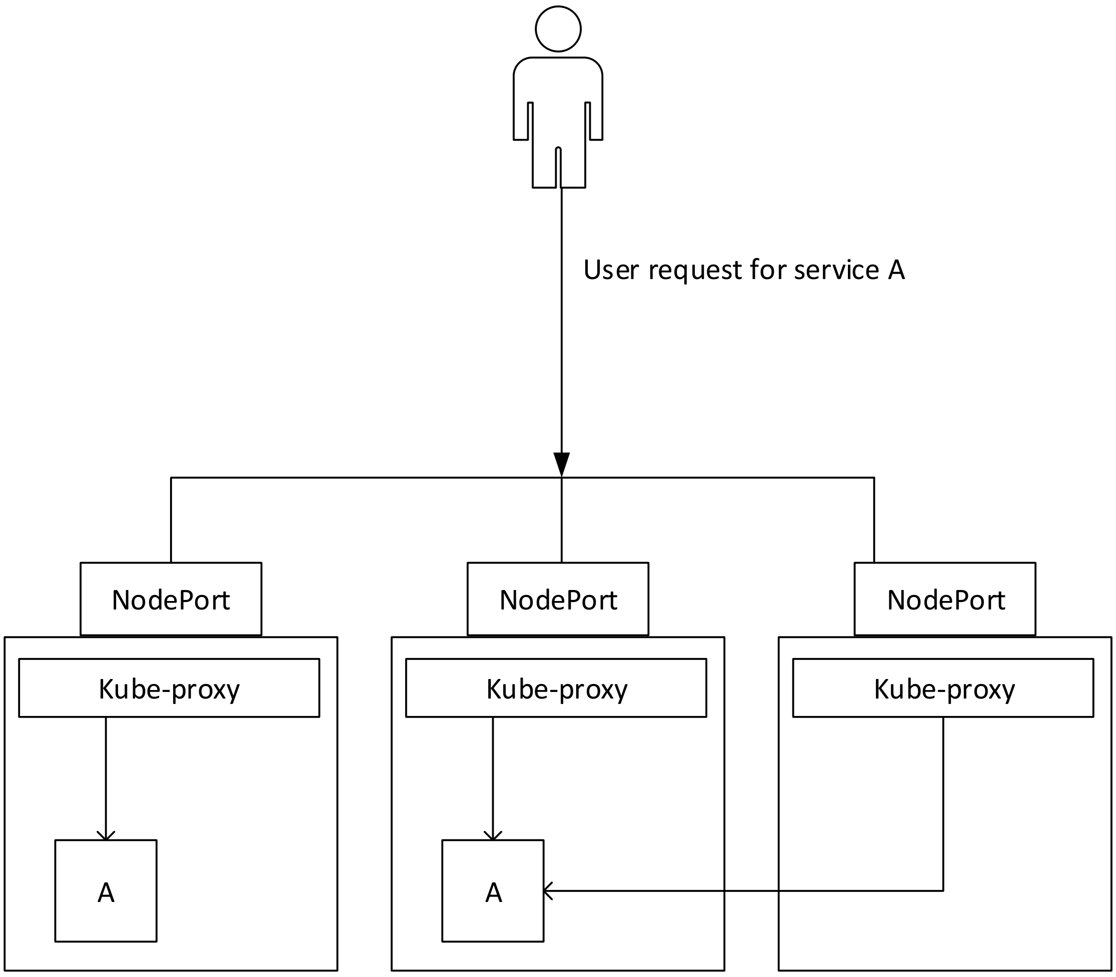


Figure 4.31: Kubernetes built-in load balance

All the NodePort are connected by kube-proxy and internally load balanced. So, when a user request to use the service A, he/she get the facilities of load balancing as shown in the implementation.

Please note that, both Docker swarm and Kubernetes offer the layer 4 load balancer. If a user wants to use more advanced load balancing facilities such as HTTPS then he/she need to use an external load balancer.

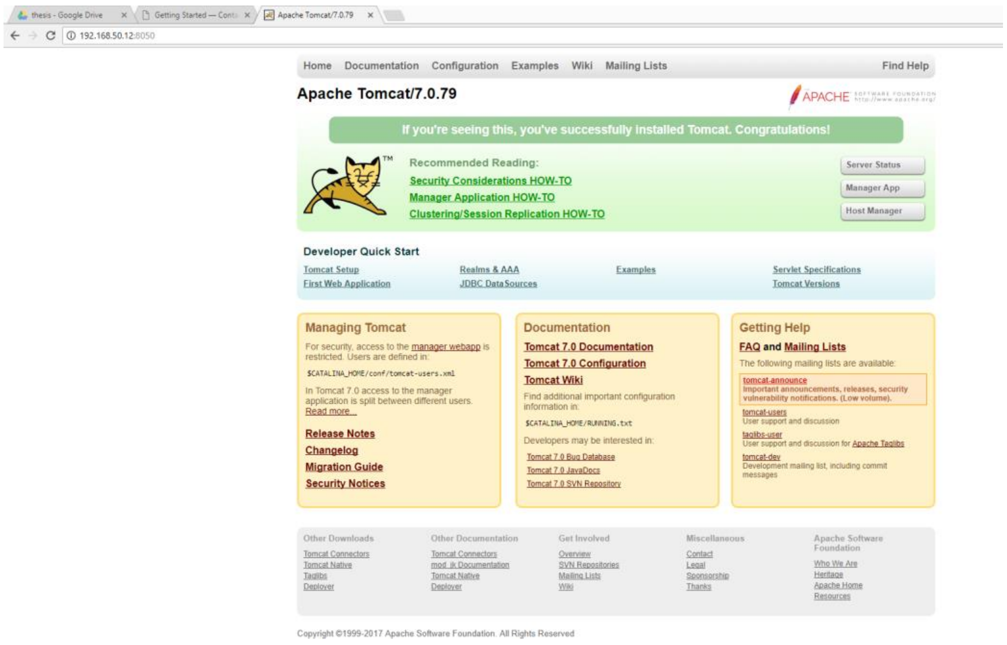
4.12 Container Update and Rollback in Docker Swarm

According to figure 3.11 the main target is to update the container version. In Docker swarm Apache Tomcat container will be updated from version 7.0 to 8.0

To create a service which is using tomcat:7.0 version

```
docker service create --replicas 1 --name Tomcat_Update --update-delay 10s -
-publish 8050:8080 tomcat:7.0
```

This command will create a service named `Tomcat_Update` which is using tomcat version 7.0. As the update delay is mentioned 10 seconds so the update process will take 10 seconds. Now if the user browses `192.168.50.12:8050` then the user will see the following result as figure 4.29



Apache Tomcat/7.0.79

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Developer Quick Start
Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat
For security, access to the manager webapp is restricted. Users are defined in:
```
$CATALINA_HOME/conf/tomcat-users.xml
```
In Tomcat 7.0 access to the manager application is split between different users. Read more...

Release Notes
Changelog
Migration Guide
Security Notices

Documentation
Tomcat 7.0 Documentation
Tomcat 7.0 Configuration
Tomcat Wiki
Find additional important configuration information in:
```
$CATALINA_HOME/RUNNING.txt
```
Developers may be interested in:
Tomcat 7.0 Bug Database
Tomcat 7.0 JavaDocs
Tomcat 7.0 SVN Repository

Getting Help
FAQ and Mailing Lists
The following mailing lists are available:
tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).
tomcat-users
User support and discussion
taglibs-user
User support and discussion for Apache Taglibs
tomcat-dev
Development mailing list, including commit messages

Other Downloads: Tomcat Connectors, Tomcat Native, Taglibs, Deployer
Other Documentation: Tomcat Connectors, mod_jk Documentation, Tomcat Native, Deployer
Get Involved: Overview, SVN Repositories, Mailing Lists, Wiki
Miscellaneous: Contact, Legal, Sponsorship, Thanks
Apache Software Foundation: Who We Are, Heritage, Apache Home, Resources

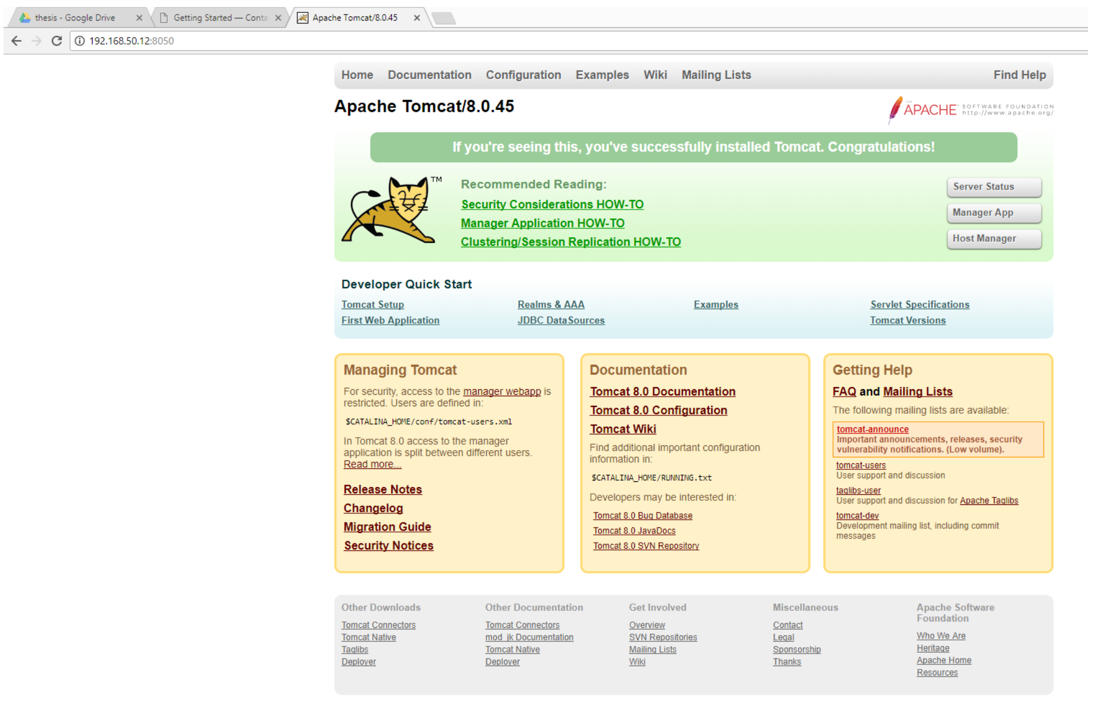
Copyright ©1999-2017 Apache Software Foundation. All Rights Reserved

Figure 4.32: Tomcat version 7.0.79 before update in Docker swarm cluster

From figure 4.26 it is seen that the Tomcat version is 7.0.79. To update the Tomcat version in 8.0

```
docker service update –image tomcat:8.0 Tomcat_Update
```

This command updates the container version from 7.0 to 8.0. To see the output user can access `192.168.50.12:8050`. The output will be like below figure 4.33



thesis - Google Drive | Getting Started — Conta | Apache Tomcat/8.0.45

192.168.50.12:8050

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/8.0.45

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Server Status
Manager App
Host Manager

Developer Quick Start

Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat

For security, access to the manager webapp is restricted. Users are defined in:

`$CATALINA_HOME/conf/tomcat-users.xml`

In Tomcat 8.0 access to the manager application is split between different users. Read more...

Release Notes
Changelog
Migration Guide
Security Notices

Documentation

Tomcat 8.0 Documentation
Tomcat 8.0 Configuration
Tomcat Wiki

Find additional important configuration information in:

`$CATALINA_HOME/RUNNING.txt`

Developers may be interested in:

Tomcat 8.0 Bug Database
Tomcat 8.0 JavaDocs
Tomcat 8.0 SVN Repository

Getting Help

FAQ and Mailing Lists

The following mailing lists are available:

tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).

tomcat-users
User support and discussion

taglibs-user
User support and discussion for Apache Taglibs

tomcat-dev
Development mailing list, including commit messages

Other Downloads: Tomcat Connectors, Tomcat Native, Taglibs, Deployer

Other Documentation: Tomcat Connectors, mod_jk Documentation, Tomcat Native, Deployer

Get Involved: Overview, SVN Repositories, Mailing Lists, Wiki

Miscellaneous: Contact, Legal, Sponsorship, Thanks

Apache Software Foundation: Who We Are, Heritage, Apache Home, Resources

Figure 4.33: Tomcat version 8.0.45 after update in Docker swarm cluster

From the figure 4.32 it is seen that after update the Tomcat version is 8.0.45.

4.13 Container Update and Rollback in Kubernetes

As like section 4.12 in Kubernetes cluster the tomcat container updated from version 7 to version 8. To achieve this a replication controller `update.yaml` file is used. The source code is in below

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-tomcat
spec:
  replicas: 2
  template:
    metadata:
      name: tomcat
      labels:
        app: tomcat
    spec:
      containers:
```

```
      - name: tomcat
        image: tomcat:7.0
        ports:
        - containerPort: 8080
```

To create the replication controller

```
kubectl create -f update.yaml
```

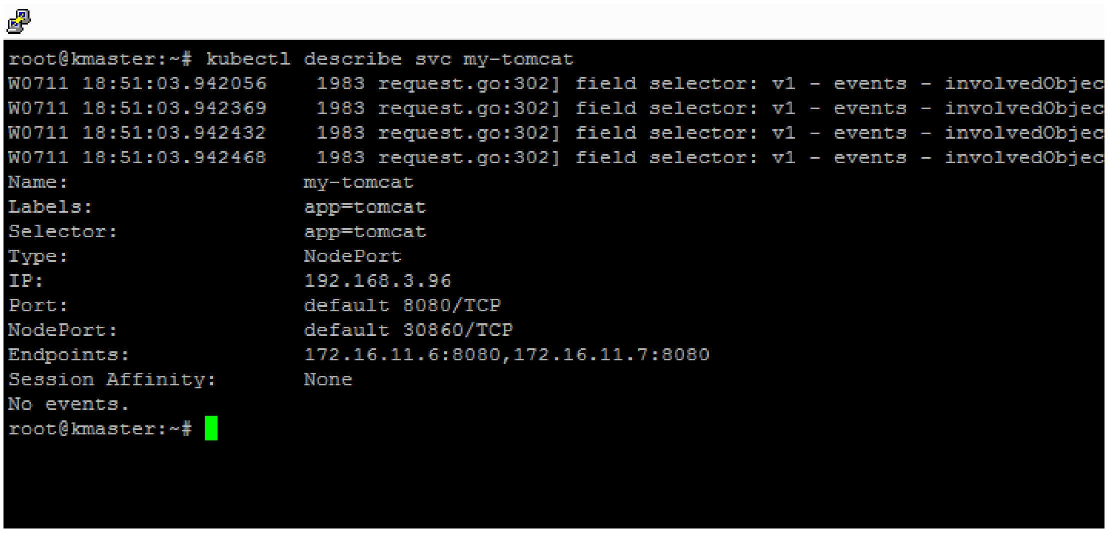
To create the service

```
kubectl expose rc my-tomcat --type=NodePort
```

Check the service

```
kubectl describe svc my-tomcat
```

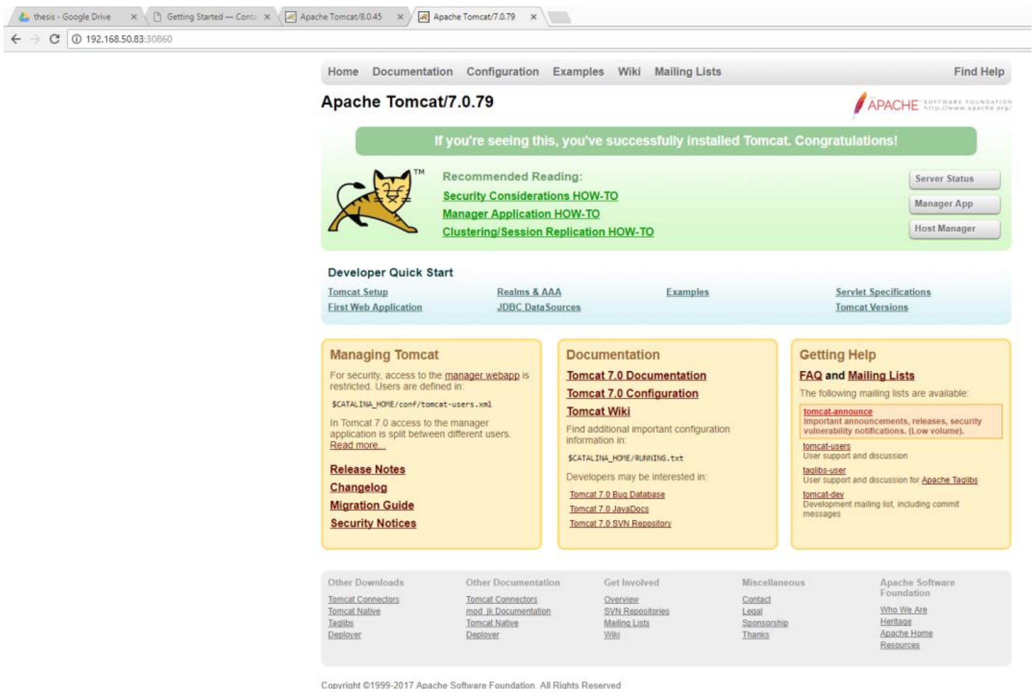
The output is like figure 4.33



```
root@kmaster:~# kubectl describe svc my-tomcat
W0711 18:51:03.942056    1983 request.go:302] field selector: v1 - events - involvedObjec
W0711 18:51:03.942369    1983 request.go:302] field selector: v1 - events - involvedObjec
W0711 18:51:03.942432    1983 request.go:302] field selector: v1 - events - involvedObjec
W0711 18:51:03.942468    1983 request.go:302] field selector: v1 - events - involvedObjec
Name:                   my-tomcat
Labels:                 app=tomcat
Selector:               app=tomcat
Type:                   NodePort
IP:                     192.168.3.96
Port:                   default 8080/TCP
NodePort:               default 30860/TCP
Endpoints:              172.16.11.6:8080,172.16.11.7:8080
Session Affinity:       None
No events.
root@kmaster:~#
```

Figure 4.34: Service Status

To access the output of the service user should access the following address 192.168.50.83:30860. The output will like below figure 4.35



Apache Tomcat/7.0.79

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Developer Quick Start
Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat
For security, access to the manager webapp is restricted. Users are defined in:
`$CATALINA_HOME/conf/tomcat-users.xml`
In Tomcat 7.0 access to the manager application is split between different users. Read more...
Release Notes
Changelog
Migration Guide
Security Notices

Documentation
Tomcat 7.0 Documentation
Tomcat 7.0 Configuration
Tomcat Wiki
Find additional important configuration information in:
`$CATALINA_HOME/RUNNING.txt`
Developers may be interested in:
Tomcat 7.0 Bug Database
Tomcat 7.0 JavaDocs
Tomcat 7.0 SVN Repository

Getting Help
FAQ and Mailing Lists
The following mailing lists are available:
tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).
tomcat-users
User support and discussion
taglibs-user
User support and discussion for Apache Taglibs
tomcat-dev
Development mailing list, including commit messages

Other Downloads: Tomcat Connectors, Tomcat Native, Taglibs, Deployer
Other Documentation: Tomcat Connectors, mod_jk Documentation, Tomcat Native, Deployer
Get Involved: Overview, SVN Repositories, Mailing Lists, Wiki
Miscellaneous: Contact, Legal, Sponsorship, Thanks
Apache Software Foundation: Who We Are, Heritage, Apache Home, Resources

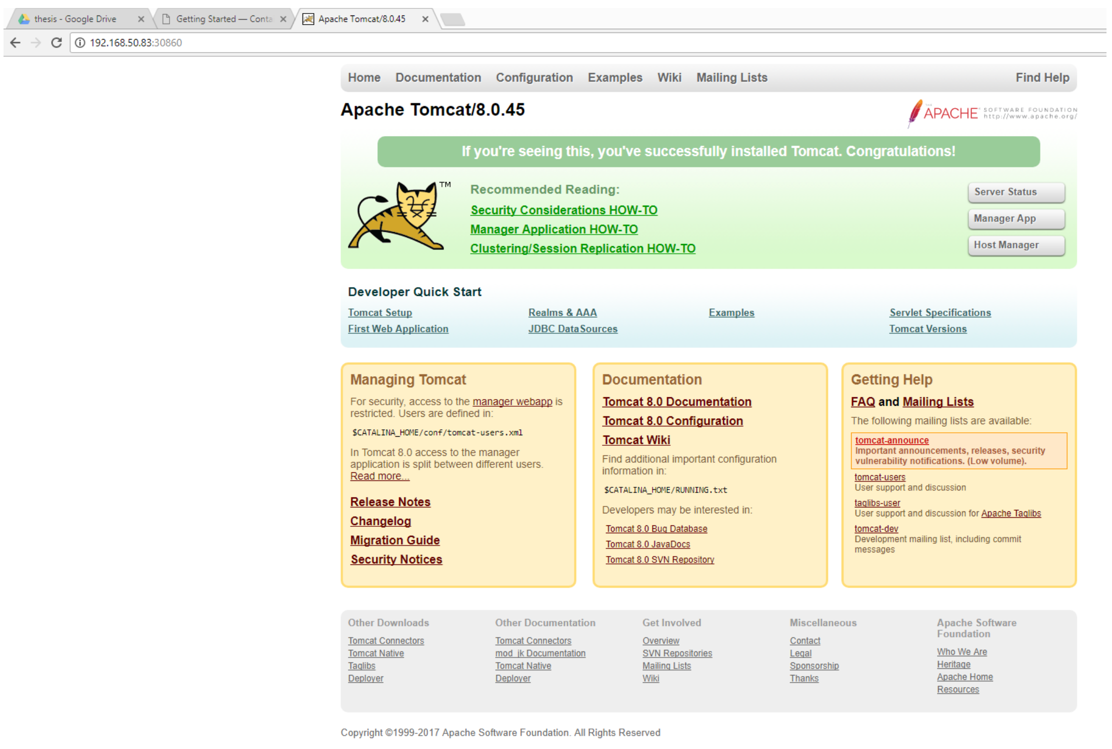
Copyright ©1999-2017 Apache Software Foundation. All Rights Reserved

Figure 4.35: Tomcat version 7.0.79 before update in Kubernetes cluster

To update the version

```
kubectl rolling-update my-tomcat --image=tomcat:8.0
```

To access the output of the service user should access the following address `192.168.50.83:30860`. The output will like below figure 4.36


thesis - Google Drive
Getting Started — Conta
Apache Tomcat/8.0.45

192.168.50.83:30860

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/8.0.45

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Server Status
Manager App
Host Manager

Developer Quick Start
Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat
For security, access to the manager webapp is restricted. Users are defined in:
`$CATALINA_HOME/conf/tomcat-users.xml`
In Tomcat 8.0 access to the manager application is split between different users. Read more...
Release Notes
Changelog
Migration Guide
Security Notices

Documentation
Tomcat 8.0 Documentation
Tomcat 8.0 Configuration
Tomcat Wiki
Find additional important configuration information in:
`$CATALINA_HOME/RUNNING.txt`
Developers may be interested in:
Tomcat 8.0 Bug Database
Tomcat 8.0 JavaDocs
Tomcat 8.0 SVN Repository

Getting Help
FAQ and Mailing Lists
The following mailing lists are available:
tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).
tomcat-users
User support and discussion
taglibs-user
User support and discussion for Apache Taglibs
tomcat-dev
Development mailing list, including commit messages

Other Downloads
Tomcat Connectors
Tomcat Native
Taglibs
Deployer

Other Documentation
Tomcat Connectors
mod_jk Documentation
Tomcat Native
Deployer

Get Involved
Overview
SVN Repositories
Mailing Lists
Wiki

Miscellaneous
Contact
Legal
Sponsorship
Thanks

Apache Software Foundation
Who We Are
Heritage
Apache Home
Resources

Copyright ©1999-2017 Apache Software Foundation. All Rights Reserved

Figure 4.36: Tomcat version 8.0.45 before update in Kubernetes cluster

4.14 Live Migration

In figure 3.12 the target state of live migration has been explained. Unfortunately, both Docker swarm and Kubernetes does not support it right now. It is possible to deploy the container in a node where user wants to migrate the container (as explained in section 4.7.1, 4.7.2 and 4.7.3) and then delete the old container.

4.15 Performance Analysis

To analyze the performance, it has examined that how a container utilizes the resources in a node. For this a prime number counter application has been used which counts all the prime numbers between 1 million. Section 4.5.4 describes this application in detail. As this application is written in java, so the performances have analyzed by running it as a java native application and a containerized application. The performance compared based on few points which are average run time in second, cpu usages in percentage and ram usages in percentage. The table 4.2 and figure 4.37 showed the comparison of average run time

Table 4.2 : Average runtime comparison

Event no	Application run time-Docker	Application run time-nativ
1	169	186
2	172	187
3	179	186
4	168	186

5	198	186
6	193	187
7	199	186
8	169	186
9	201	186
10	193	186
11	172	187
12	180	187
13	181	186
14	206	186
15	208	188
16	191	186
17	191	186
18	191	189
19	191	187
20	169	186
21	170	186
22	185	186
23	198	186
24	197	186
25	200	186
26	191	187
27	191	187
28	216	186
29	169	186
30	191	186
31	191	186
32	191	187
33	168	186
34	191	187
35	191	187
36	191	186
37	196	187
38	198	186
39	194	187
40	191	186
41	191	185
42	170	186
43	204	185
44	191	186
45	191	185

46	191	186
47	191	185
48	191	186
49	191	185
50	191	186

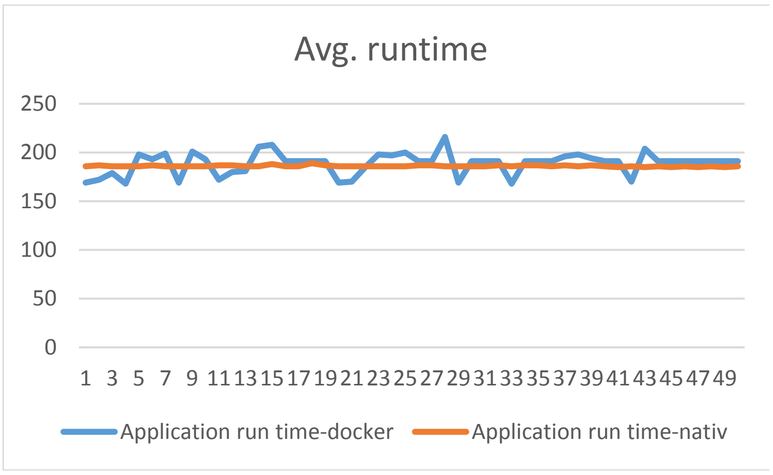


Figure 4.37: Average runtime comparison chart

The table 4.3 and figure 4.38 shows the CPU uses comparison

Table 4.3 : CPU uses comparison

Event no	Cpu usages-Docker	Cpu usages-nativ
1	98,60	92,70
2	97,00	94,10
3	97,1	92,50
4	98,3	92,7
5	87	91,8
6	88,7	92,6
7	89,6	91,7
8	98	92,4
9	97,1	92,4
10	88,6	92,1
11	96,7	92,8
12	97,1	92,5
13	95,7	94,1
14	90,6	92,6
15	88,3	93,8
16	87,8	93,4
17	89,4	93,4
18	89	91,6

19	89,4	91,3
20	98,1	91,1
21	96,7	93,7
22	89,8	93,8
23	97,1	93,4
24	92	92,9
25	88,7	92
26	88	94,1
27	89,1	92,4
28	96	94,2
29	94,3	92,4
30	88,6	93,4
31	88,4	94,5
32	88,3	94
33	92,5	92,5
34	88,6	93,8
35	88	92,8
36	88	93,5
37	84	94,1
38	89,1	92,4
39	87,7	91,2
40	85,7	92,4
41	88,3	92,7
42	97	94,2
43	86,4	93,2
44	88	94,4
45	86,3	95,1
46	87	95,4
47	86,2	93,7
48	86,7	94,1
49	85,7	92,4
50	91,1	95,1

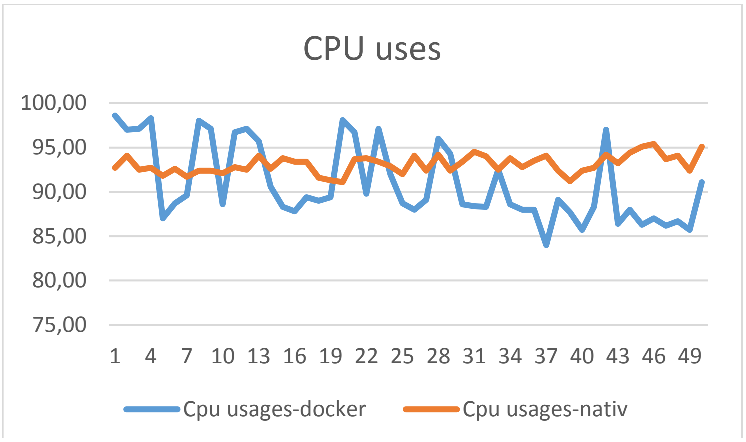


Figure 4.38: CPU uses comparison chart

The table 4.4 and figure 4.39 shows the RAM uses comparison

Table 4.4 : RAM uses comparison

Event no	Used RAM-Docker	Used RAM-native
1	912	924
2	923	925
3	881	926
4	921	918
5	682	918
6	921	919
7	911	923
8	918	923
9	924	923
10	921	923
11	931	924
12	886	924
13	910	924
14	929	925
15	918	926
16	909	926
17	909	926
18	908	927
19	908	912
20	912	912
21	927	910
22	887	913
23	913	912
24	929	913
25	890	913
]

26	895	914
27	908	914
28	613	915
29	925	914
30	914	915
31	912	915
32	906	915
33	917	915
34	932	914
35	923	914
36	914	916
37	610	916
38	911	917
39	927	917
40	922	918
41	910	924
42	906	924
43	916	924
44	914	902
45	909	902
46	912	906
47	919	906
48	915	906
49	918	906
50	921	907

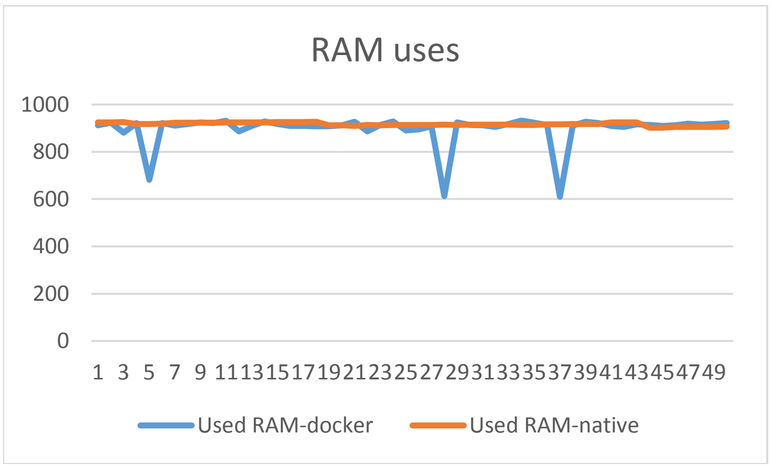


Figure 4.39: RAM uses comparison chart

From the above all comparison it is seen that there is not much difference between the native application and the containerized application.

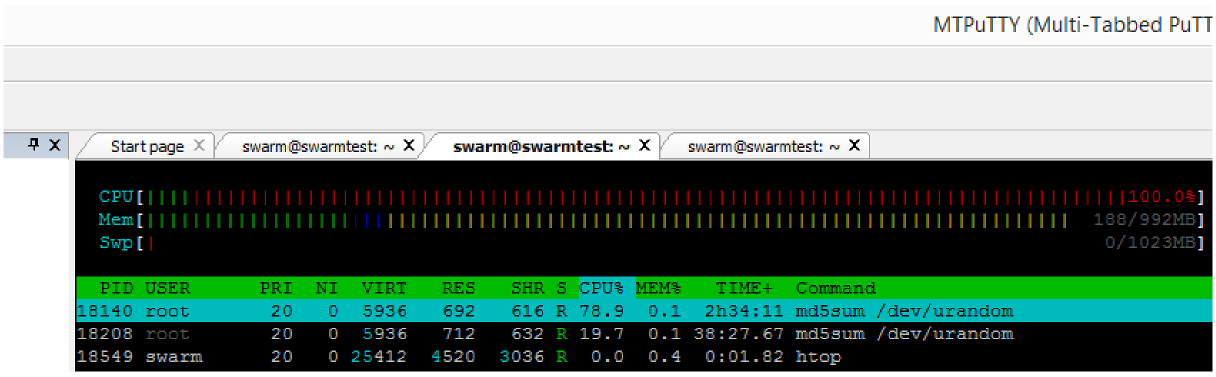
When a user runs lots of container in a node, then it is very important to allocate the resources such as CPU for every containerized application which is essential to maintain the performance of that node. This is possible to achieve by using control group (Cgroup). Cgroups allows user to allocate resources — such as CPU time, system memory, network bandwidth, or combinations of these resources — among user-defined groups of tasks (processes) running on a system (Introduction to control group, 2017).

To test Cgroup, two containers created which are using the primenumber image (as explained in sec 4.5.2) and allocate the resource 80 percent and 20 percent respectively.

```
docker run -d --name='high_prio_prime' --cpuset-cpus=0 --cpu-shares=80 tan-
zeemzillu/tanzeemzillu:primenumber md5sum /dev/urandom
```

```
docker run -d --name='low_prio_prime' --cpuset-cpus=0 --cpu-shares=20 tan-
zeemzillu/tanzeemzillu:primenumber md5sum /dev/urandom
```

After that the CPU usages are same as shown in below figure 4.40



```
  CPU[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||    188/992MB]
  Swp[|                                                                      0/1023MB]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
18140 root       20   0  5936   692   616 R 78.9  0.1  2h34:11 md5sum /dev/urandom
18208 root       20   0  5936   712   632 R 19.7  0.1 38:27.67 md5sum /dev/urandom
18549 swarm      20   0 25412  4520  3036 R  0.0  0.4  0:01.82 htop
```

Figure 4.40: CPU uses after using Cgroup

From figure 4.39 it is seen that the CPU uses of `high prio prime` is limited between 80 percent and the `low prio prime` is limited between 20 percent. So, this is an effective way to maintain the uses of resources.

Beside this, it is possible to compare the performance of Docker swarm and Kubernetes based on the earlier use cases which are deployment of application, scaling, load balancing and container update and rollback. Considering deployment of application, in Docker swarm global application creation is possible whereas Kubernetes does not support global application. But on the other hand, it is easy to deploy an application in a specific node in Kubernetes. Considering scaling, both Docker swarm and Kubernetes supports that. How many containers a user can scale is depends on the hardware capacity of the cluster. Both Docker swarm and Kubernetes supports layer 4 build in load balancing and both supports container update and rollback.

4.16 Rancher

From the previous discussion of this thesis it is clear that, both Docker swarm and Kubernetes offers many facilities which may help an administrator to maintain the cloud in an efficient way. But beside this, a user might need to use both Kubernetes and Docker swarm for different purpose. But it is not easy as these both are follow different ways to orchestrate the cluster. In this situation, a user may use Rancher (Simple, easy-to-use container management, 2017) as rancher offers to use both Docker swarm and Kubernetes.

Rancher is an open source software platform that enables organizations to run and manage Docker and Kubernetes in production. With Rancher, organizations no longer have to build a container services platform from scratch using a distinct set of open source technologies. Rancher supplies the entire software stack needed to manage containers in production (Overview of Rancher, 2017). Below figure 4.41 shows the higher-level architecture of Rancher

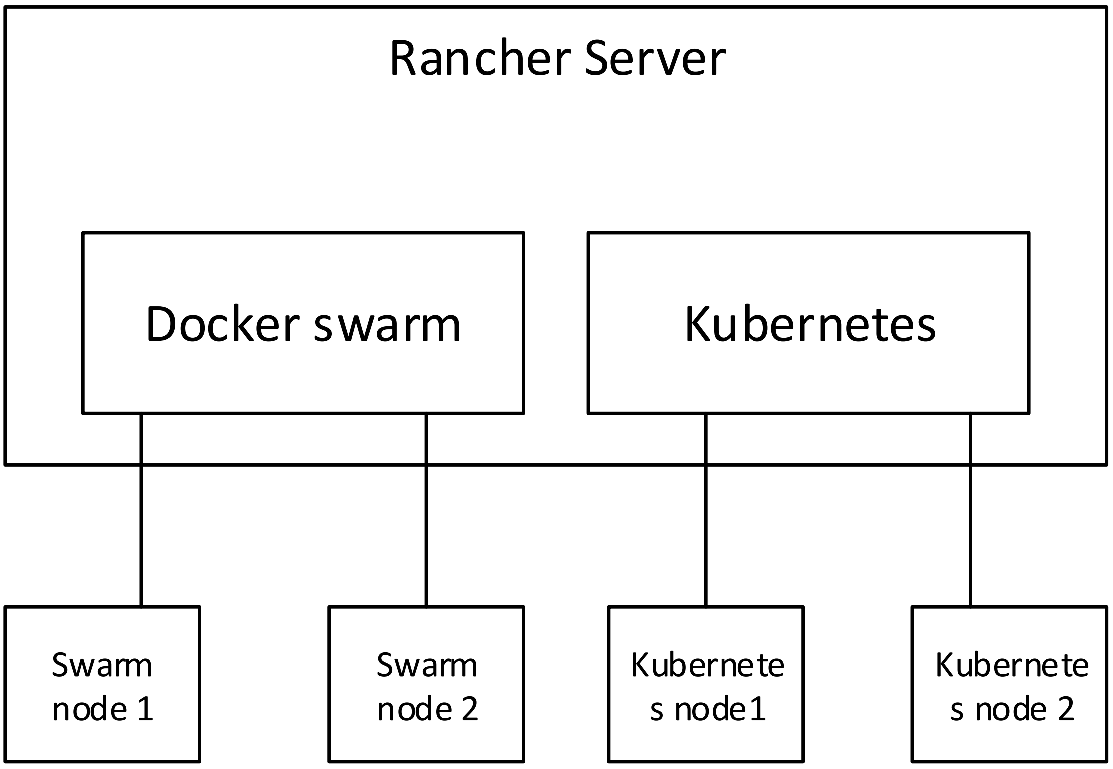


Figure 4.41: Rancher higher level architecture

In this thesis work two clusters created based on rancher where one is using Docker swarm infrastructure and the other cluster is using Kubernetes infrastructure.

4.16.1 Docker Swarm in Rancher

To establish a cluster based on rancher server three VM used. One VM is working as a rancher server and the two other VM is working as swarm host. The architecture of this implementation is shown in below figure 4.42

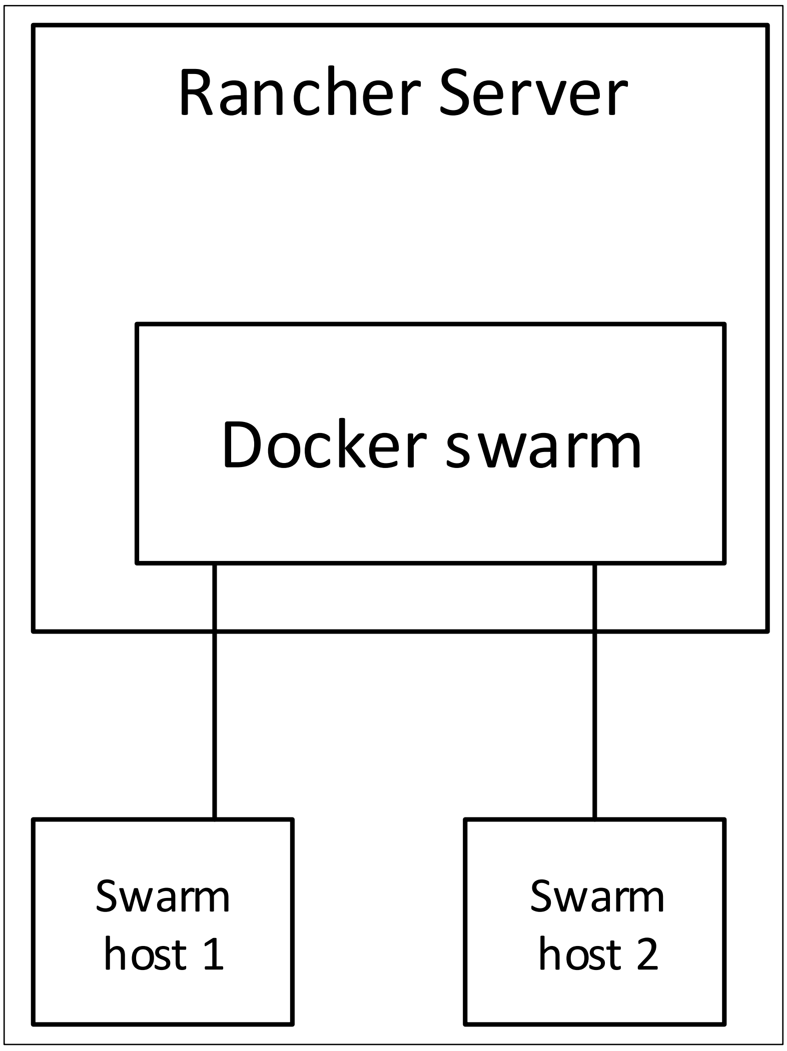


Figure 4.42: Docker swarm cluster on rancher

The specification of all VM are given in below table 4.5

Table 4.5 : VM specification for Docker swarm in Rancher

No	VM Name	User name	Password	RAM	Hard drive	Operating system
1	Rserver	rserver	harami420	2 GB	15 GB	Ubuntu-16.04.2-server-amd64
2	rs1	rs1	harami420	1 GB	10 GB	Ubuntu-16.04.2-server-amd64
3	rs2	rs2	harami420	1 GB	10 GB	Ubuntu-16.04.2-server-amd64

The IP plan for this cluster are given below table 4.6

Table 4.6 : IP plan for Docker swarm in Rancher

No	VM Name	IP address	Subnet mask
1	Rserver	10.2.0.2	255.255.255.0

2	rs1	10.2.0.3	255.255.255.0
3	rs2	10.2.0.4	255.255.255.0

For Rserver Docker version 1.12.3 and for rs1 and rs2 1.13.0 has used (Getting Started with Hosts, 2017). The procedure of install Docker version 1.12.3 are given below

- To update the apt sources

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

- To add the new GPG key

```
$ sudo apt-key adv \
              --keyserver hkp://ha.pool.sks-keyservers.net:80 \
              --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

- To substituting the entry for the operating system for the placeholder

```
$ echo "deb https://apt.Dockerproject.org/repo ubuntu-xenial main" |
sudo tee /etc/apt/sources.list.d/Docker.list
```

- To update the APT package index

```
$ sudo apt-get update
```

- To install the recommended package

```
$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-
virtual
```

- To install the specific Docker version

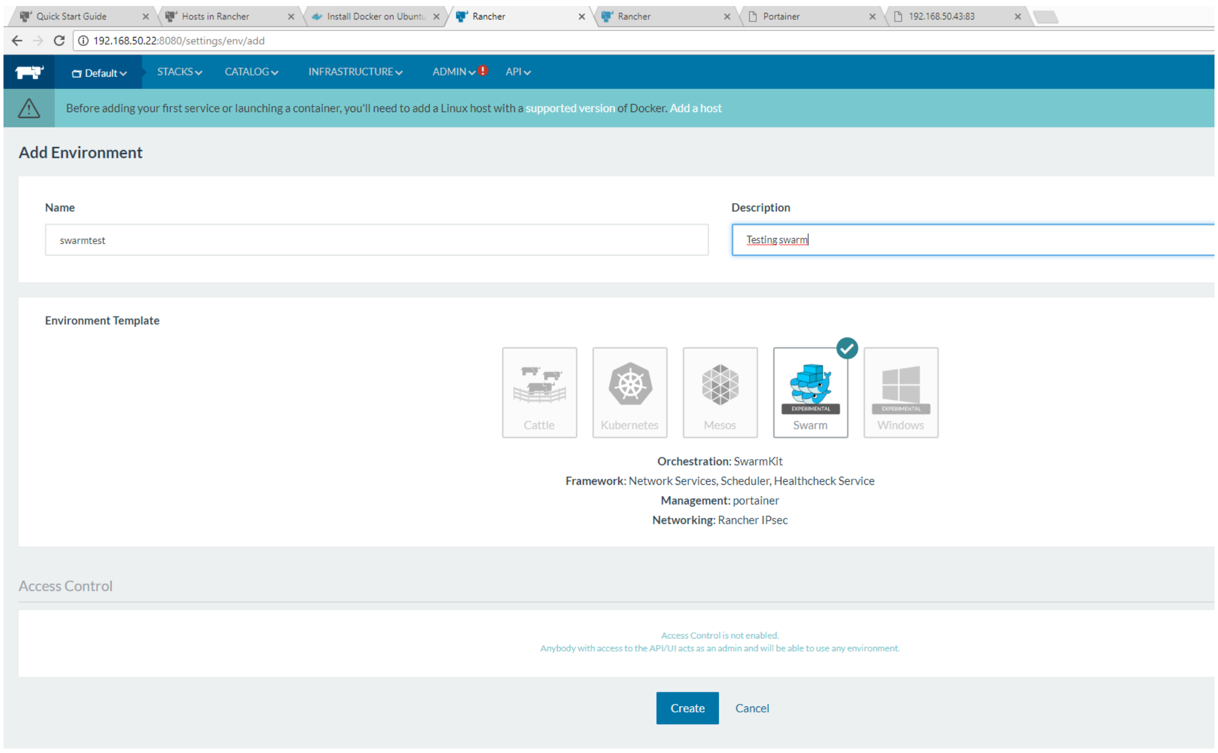
```
$ sudo apt-get install Docker-engine= 1.12.3-0~xenial 500
```

Similarly, Docker version 1.13.0 need to install in rs1 and rs2. Now to run the rancher server in Rserver the following command need to run

```
$ sudo Docker run -d --restart=unless-stopped -p 8080:8080
rancher/server:stable
```

The user interface of rancher can be accessed by `http://192.168.50.22:8080`

After launching the Rancher server an environment need to create which is using Docker swarm infrastructure. This is shown in below figure 4.43



Before adding your first service or launching a container, you'll need to add a Linux host with a supported version of Docker. Add a host

Add Environment

Name: swarmtest

Description: Testing swarm

Environment Template

Orchestration: SwarmKit
Framework: Network Services, Scheduler, Healthcheck Service
Management: portainer
Networking: Rancher IPsec

Access Control

Access Control is not enabled.
Anybody with access to the API/UI acts as an admin and will be able to use any environment.

Figure 4.43: Add Docker swarm environment

After that both Docker swarm host added which shown in below figure 4.44

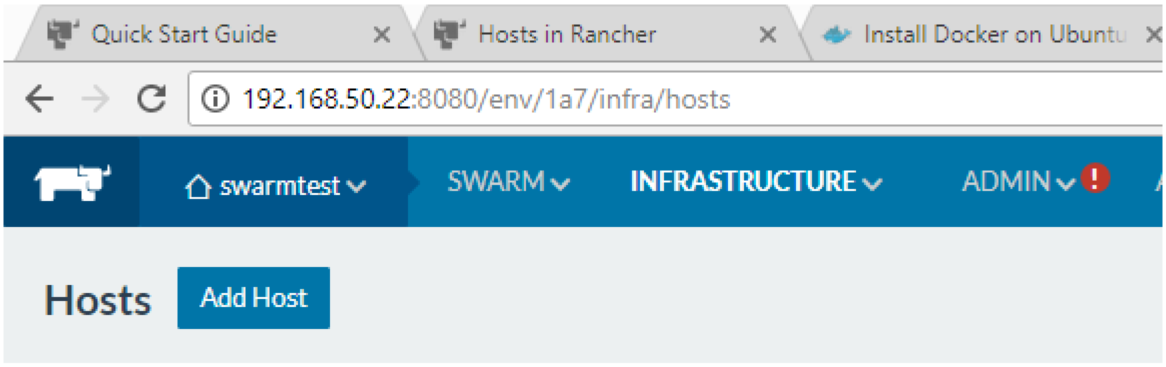
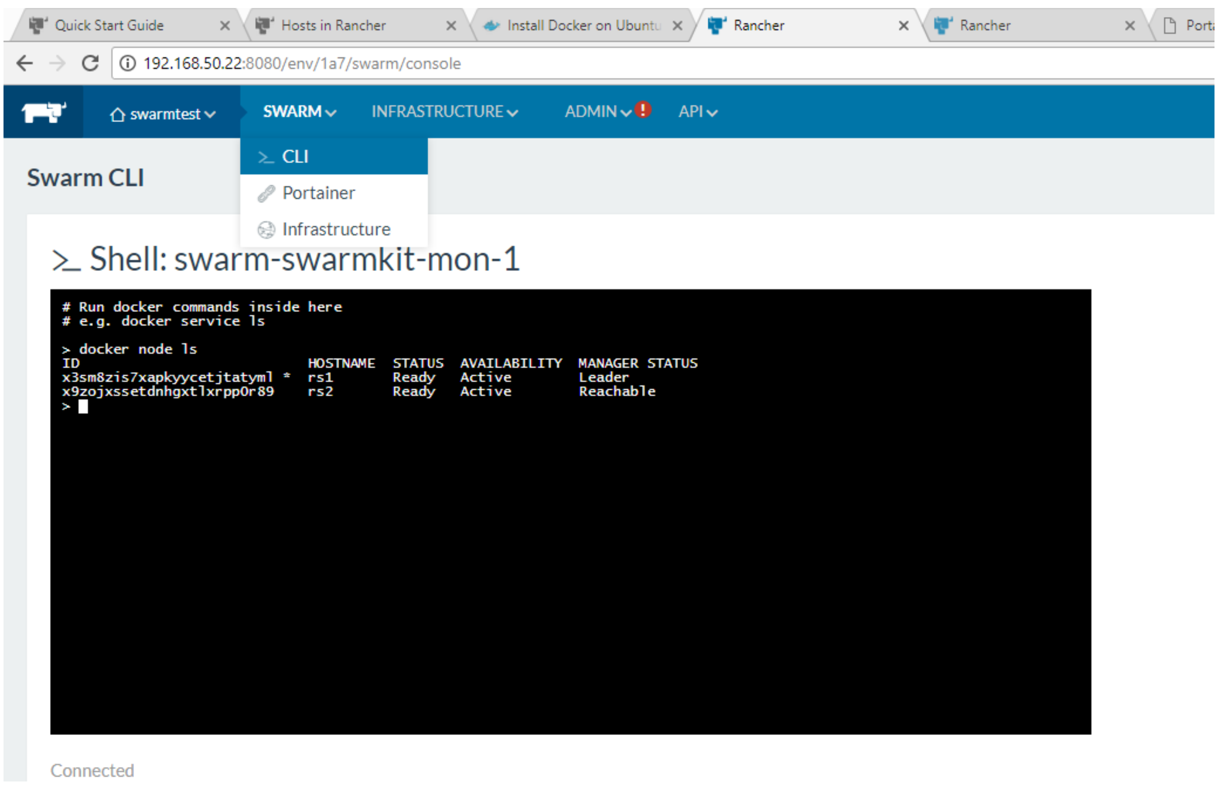


Figure 4.44: Swarm host add

The Docker swarm cluster is ready now. It can be checked by the CLI interface provided by rancher shown in figure 4.45



```
# Run docker commands inside here
# e.g. docker service ls

> docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
x3sm8zis7xapkyycetjtatyml *  rs1       Ready   Active        Leader
x9zojxssetdnhgxtlxrpp0r89    rs2       Ready   Active        Reachable
>
```

Figure 4.45: Swarm cluster status

Beside this, rancher provide a graphical interface to use Docker swarm which shown in below figure 4.45

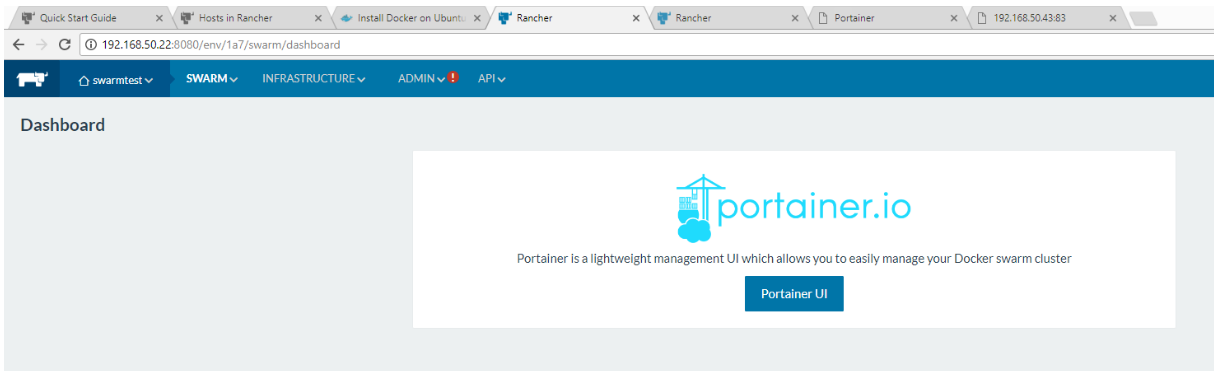
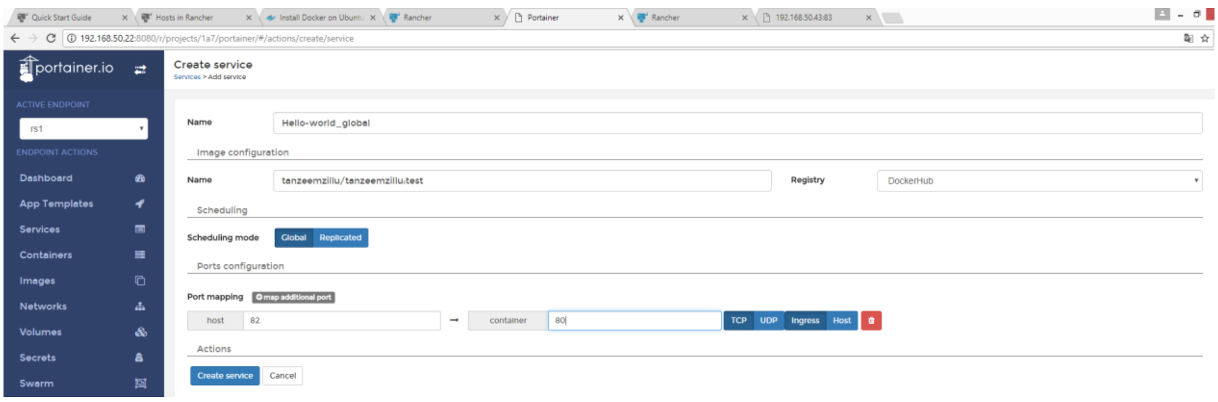


Figure 4.46: Swarm graphical interface

User can create both global and replicated service using this graphical interface. In this case a web application which prints the container/pod id (see 4.5.1) is used. Figure 4.47 and 4.48 show the global service and the replicated service respectively



Create service
Services > Add service

Name: Hello-world_global

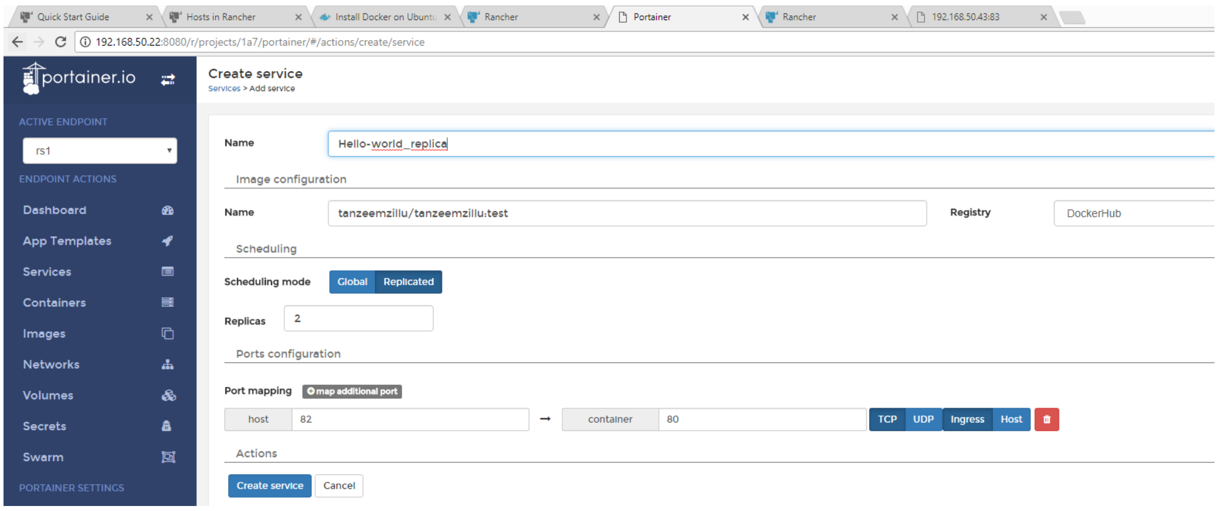
Image configuration
Name: tanzeemzillu/tanzeemzillu:test
Registry: DockerHub

Scheduling
Scheduling mode: Global | Replicated

Ports configuration
Port mapping
host 82 → container 80 — TCP | UDP | Ingress | Host

Actions
Create service | Cancel

Figure 4.47: Hello-world_global service



Create service
Services > Add service

Name: Hello-world_replica

Image configuration
Name: tanzeemzillu/tanzeemzillu:test
Registry: DockerHub

Scheduling
Scheduling mode: Global | Replicated
Replicas: 2

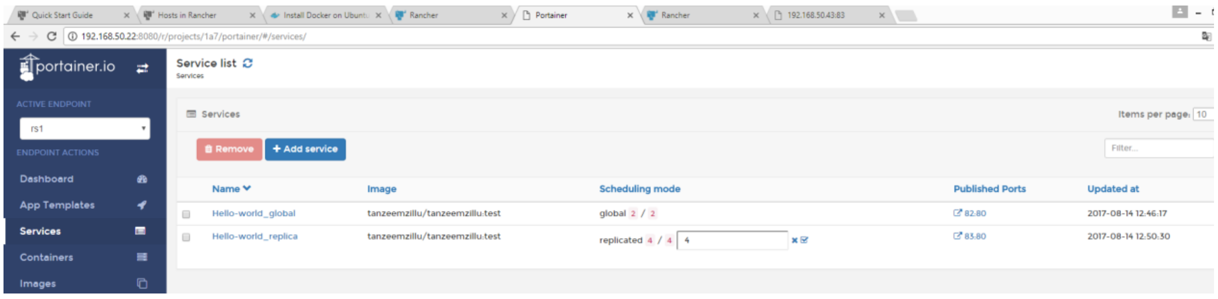
Ports configuration
Port mapping
host 82 → container 80 — TCP | UDP | Ingress | Host

Actions
Create service | Cancel

Figure 4.48: Hello-world_replica service

In figure 4.47 it is seen that a global service has created named Hello-world_global and the service has exposed in port 82. In figure 4.48 it is seen that a replicated service has created named Hello-world-replica and the service is exposed in port 82.

The replicated service is possible to scale in a very easy way as shown in figure 4.49

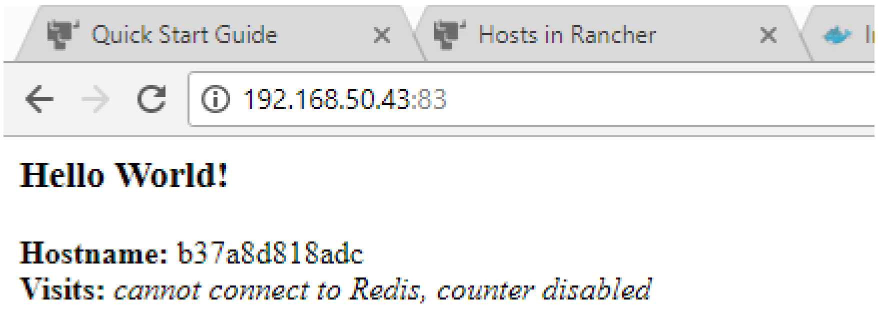


Service list
Services

	Name	Image	Scheduling mode	Published Ports	Updated at
☐	Hello-world_global	tanzeemzillu/tanzeemzillu:test	global 2 / 2	82:80	2017-08-14 12:46:17
☐	Hello-world_replica	tanzeemzillu/tanzeemzillu:test	replicated 4 / 4 — 4	83:80	2017-08-14 12:50:30

Figure 4.49: Scaling in Docker swarm in Rancher

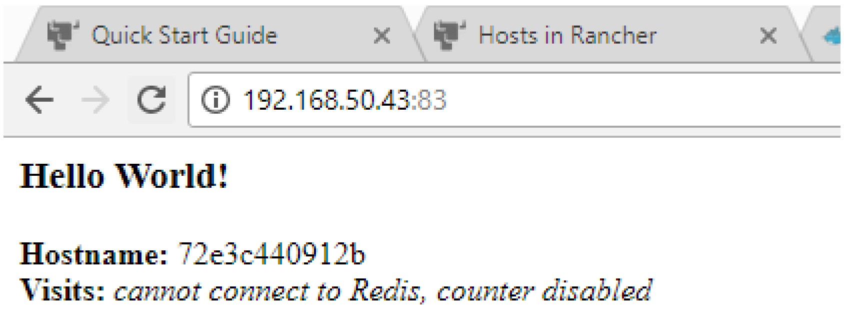
The built-in load balancing for Docker swarm is also working fine. In this experiment when a user request for a service through `http://192.168.50.43:83`, then he/she get response from different container in different time. This is shown in figure 4.50 and 4.51



Hello World!

Hostname: b37a8d818adc
Visits: *cannot connect to Redis, counter disabled*

Figure 4.50: Internal load balancing in Docker swarm in Rancher (1)



Hello World!

Hostname: 72e3c440912b
Visits: *cannot connect to Redis, counter disabled*

Figure 4.51: Internal load balancing in Docker swarm in Rancher (2)

4.16.2 Kubernetes in Rancher

In rancher, a Kubernetes cluster established by using three VM. Where one VM is working as Rancher server the other two VM is working as Kubernetes host. The architecture of this implementation is shown in below figure 4.52

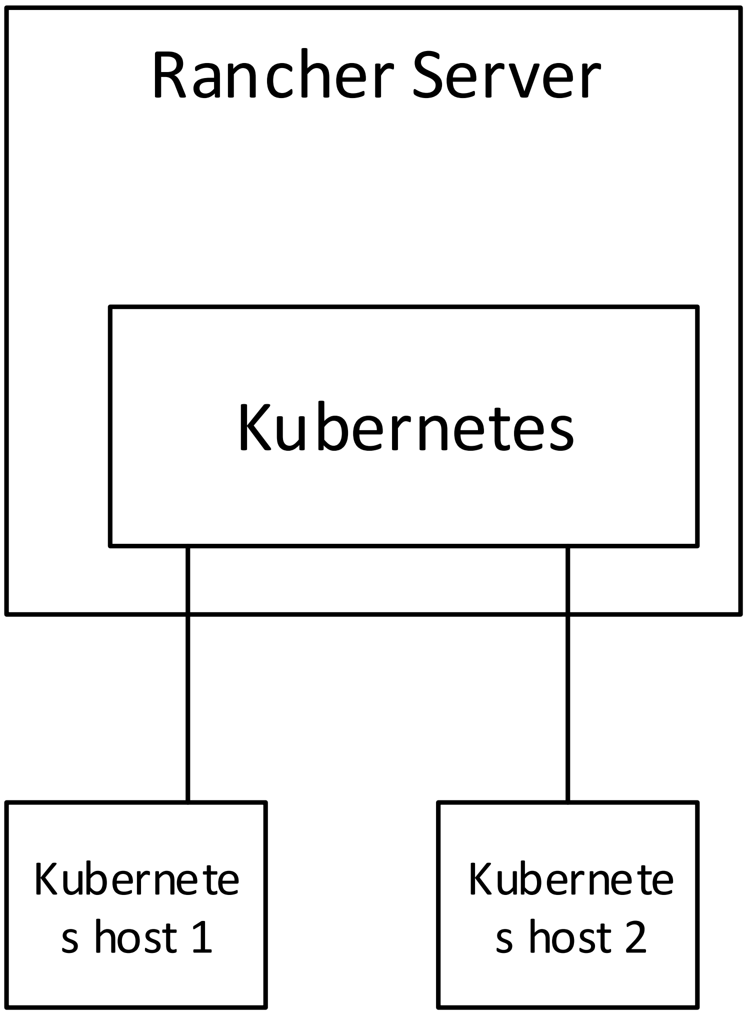


Figure 4.52: Kubernetes cluster on rancher

The specification of all VM are given in below table 4.7

Table 4.7 : VM specification for Kubernetes in Rancher

No	VM Name	User name	Password	RAM	Hard drive	Operating system
1	Rserver2	rserver2	harami420	4 GB	20 GB	Ubuntu-16.04.2-server-amd64
2	rk1	rk1	harami420	1 GB	10 GB	Ubuntu-16.04.2-server-amd64
3	rk2	rk2	harami420	1 GB	10 GB	Ubuntu-16.04.2-server-amd64

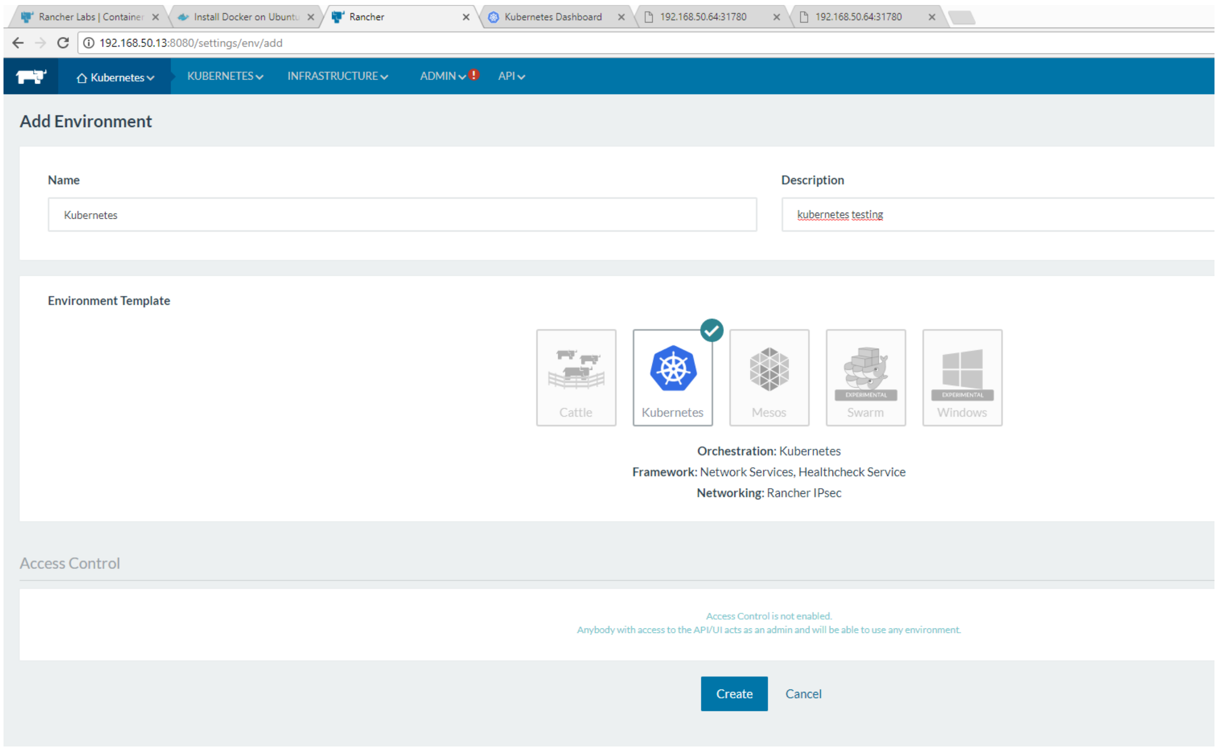
The IP plan for this cluster are given below table 4.8

Table 4.8 : IP plan for Kubernetes in Rancher

No	VM Name	IP address	Subnet mask
1	Rserver2	10.3.0.2	255.255.255.0
2	rk1	10.3.0.3	255.255.255.0
3	rk2	10.3.0.4	255.255.255.0

For Rserver2, rk1 and rk2 Docker version 1.12.3 has used [34].

As like Docker swarm a Kubernetes based environment added in rancher as shown in below figure 4.53



Add Environment

Name: Kubernetes

Description: kubernetes testing

Environment Template

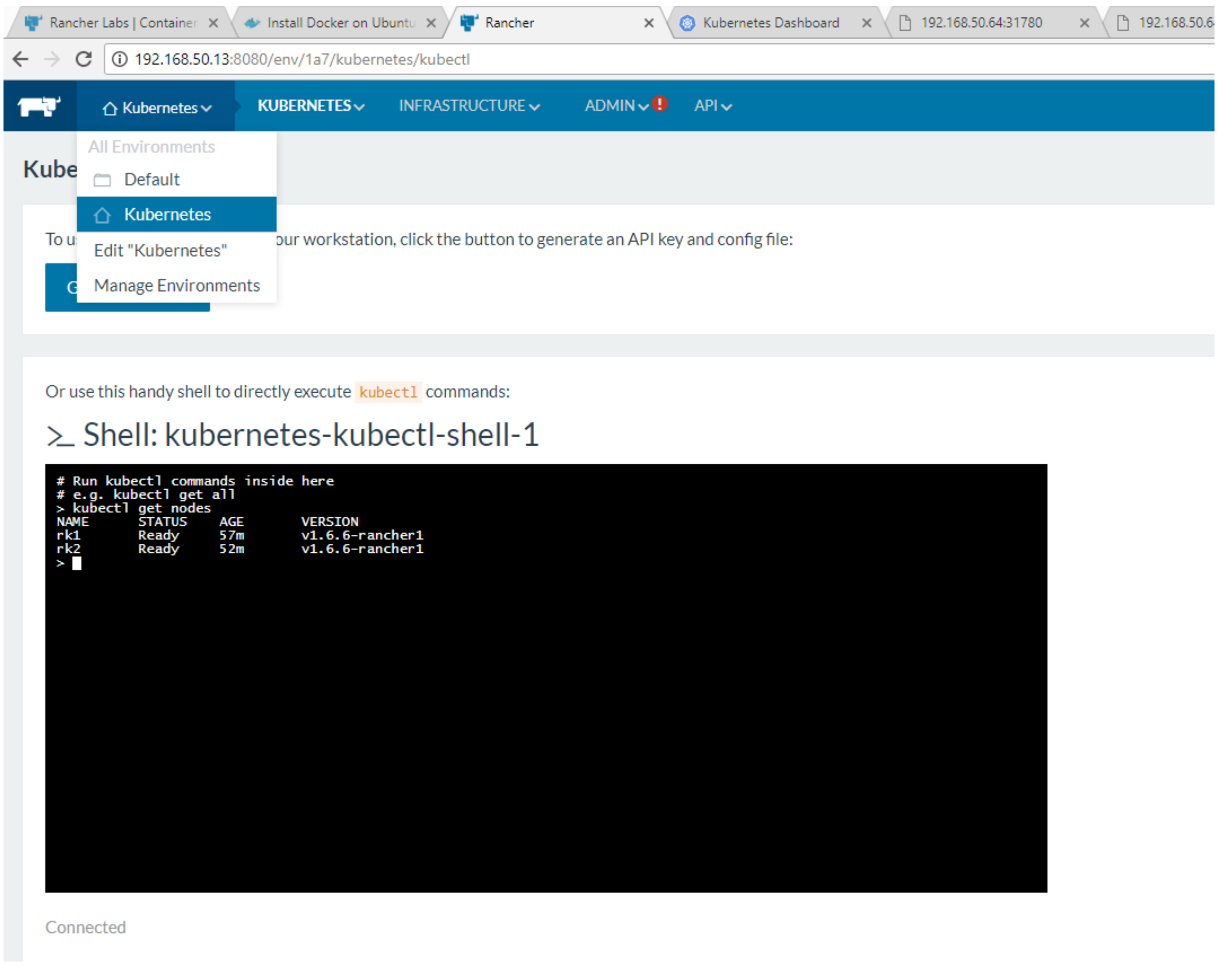
Orchestration: Kubernetes
Framework: Network Services, Healthcheck Service
Networking: Rancher IPsec

Access Control

Access Control is not enabled.
Anybody with access to the API/UI acts as an admin and will be able to use any environment.

Figure 4.53: Kubernetes environment add in rancher

After adding the environment Kubernetes host can be added. It is possible to check the cluster condition from CLI shown in below figure 4.54



To u… our workstation, click the button to generate an API key and config file:

Or use this handy shell to directly execute `kubectl` commands:

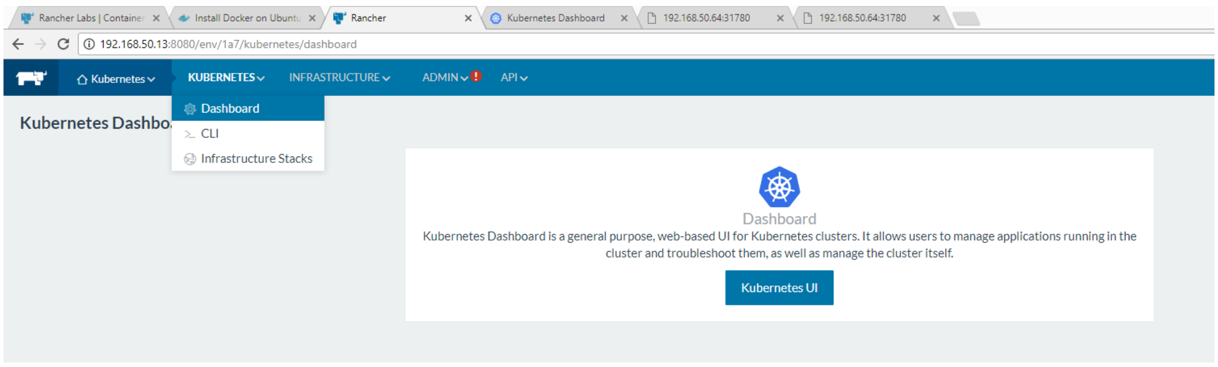
>_ Shell: kubernetes-kubectl-shell-1

```
# Run kubectl commands inside here
# e.g. kubectl get all
> kubectl get nodes
NAME      STATUS    AGE       VERSION
rk1       Ready     57m       v1.6.6-rancher1
rk2       Ready     52m       v1.6.6-rancher1
> 
```

Connected

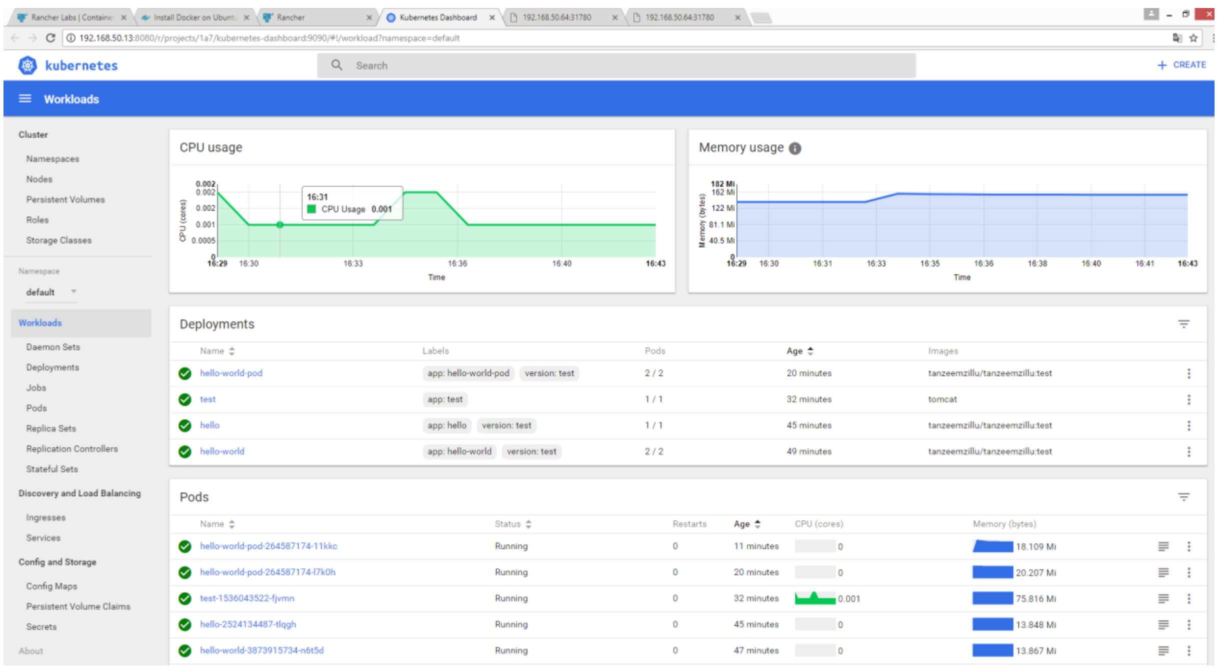
Figure 4.54: Kubernetes cluster condition in rancher

As like Docker swarm Kubernetes dashboard also available in rancher. This is shown in below figure 4.55 and 4.66



Kubernetes Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage applications running in the cluster and troubleshoot them, as well as manage the cluster itself.

Figure 4.55: Kubernetes dashboard



Deployments

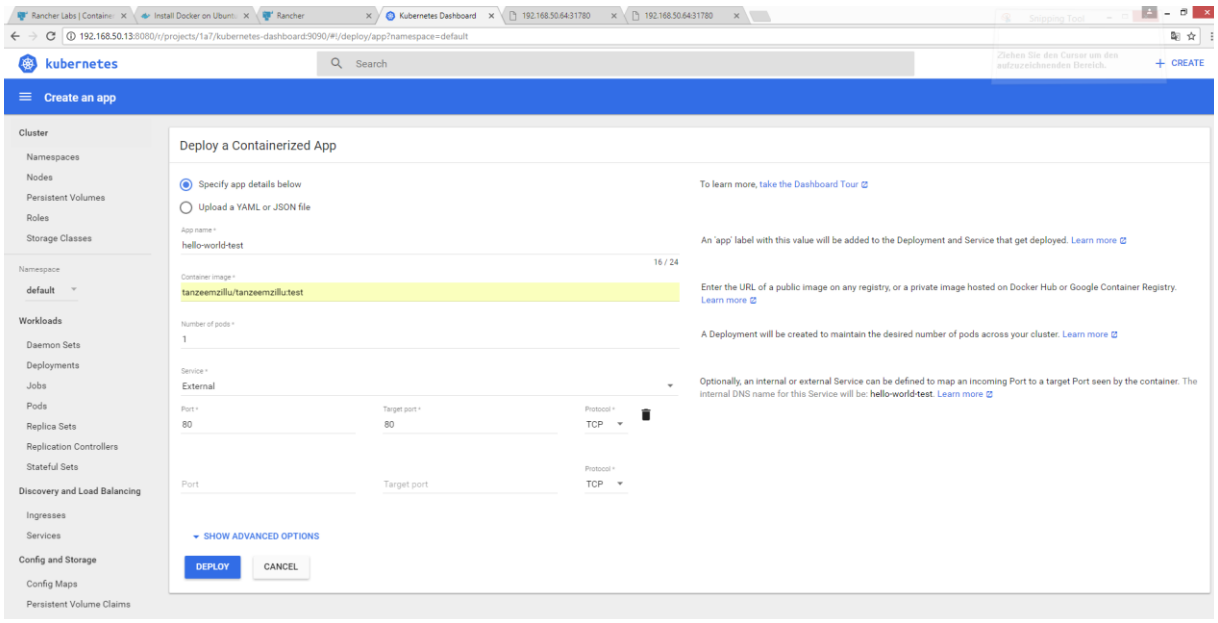
Name	Labels	Pods	Age	Images
hello-world-pod	app: hello-world-pod version: test	2 / 2	20 minutes	tanzeemzillu/tanzeemzillu:test
test	app: test	1 / 1	32 minutes	tomcat
hello	app: hello version: test	1 / 1	45 minutes	tanzeemzillu/tanzeemzillu:test
hello-world	app: hello-world version: test	2 / 2	49 minutes	tanzeemzillu/tanzeemzillu:test

Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
hello-world-pod-264587174-11kkc	Running	0	11 minutes	0	18.109 Mi
hello-world-pod-264587174-l7k0h	Running	0	20 minutes	0	20.207 Mi
test-1536043522-fjvmn	Running	0	32 minutes	0.001	75.816 Mi
hello-2524134487-tlqgh	Running	0	45 minutes	0	13.848 Mi
hello-world-3873915734-n6t5d	Running	0	47 minutes	0	13.867 Mi

Figure 4.56: Kubernetes dashboard (2)

A service can be deployed in a easy way through this dashboard shown in below figure 4.57



Deploy a Containerized App

(● Specify app details below) ○ Upload a YAML or JSON file

To learn more, take the Dashboard Tour

App name *: hello-world-test (16 / 24)

An 'app' label with this value will be added to the Deployment and Service that get deployed. Learn more

Container image *: tanzeemzillu/tanzeemzillu:test

Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. Learn more

Number of pods *: 1

A Deployment will be created to maintain the desired number of pods across your cluster. Learn more

Service *: External

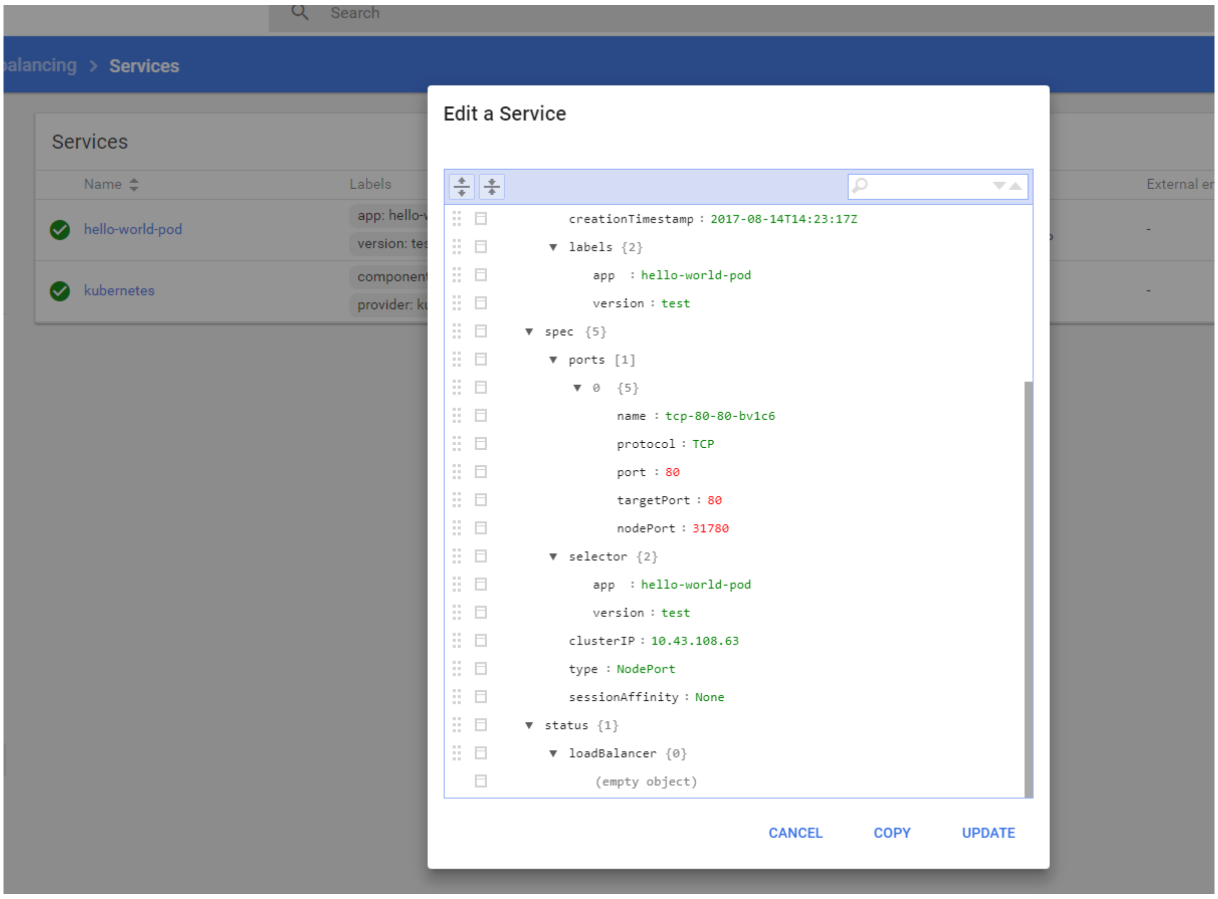
Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. The internal DNS name for this Service will be: hello-world-test. Learn more

Port *: 80 Target port *: 80 Protocol *: TCP

Port Target port Protocol *: TCP

Figure 4.57: service deployment

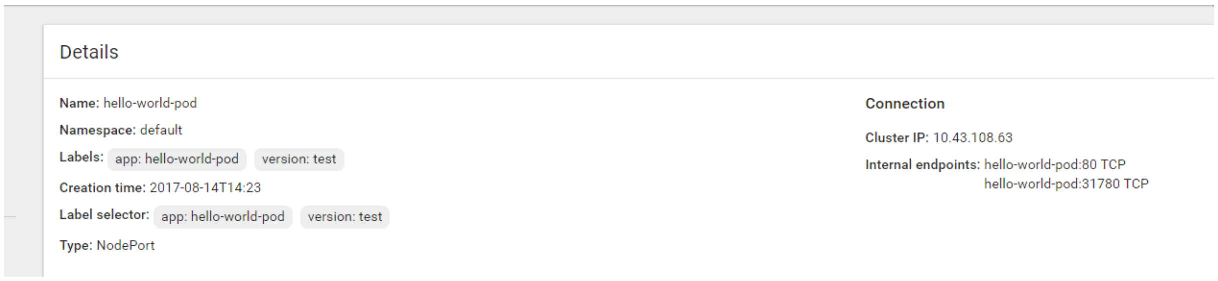
In this case a service is going to creating which is using a web application which prints the container/pod id (see 4.5.1). And the service type is `NodePort`. But here in default the service type is `LoadBalancer`. So to run the service perfectly is it must to edit the service and change the type in `NodePort` as shown in below figure 4.58



```
    creationTimestamp : 2017-08-14T14:23:17Z
  ▼ labels {2}
      app   : hello-world-pod
      version : test
▼ spec {5}
  ▼ ports [1]
    ▼ 0  {5}
        name : tcp-80-80-bv1c6
        protocol : TCP
        port : 80
        targetPort : 80
        nodePort : 31780
  ▼ selector {2}
      app   : hello-world-pod
      version : test
    clusterIP : 10.43.108.63
    type : NodePort
    sessionAffinity : None
▼ status {1}
  ▼ loadBalancer {0}
      (empty object)
```

Figure 4.58: Changing service type

After that a port is available to expose the service which shown in below figure 4.59



Details

Name: hello-world-pod
Namespace: default
Labels: app: hello-world-pod version: test
Creation time: 2017-08-14T14:23
Label selector: app: hello-world-pod version: test
Type: NodePort

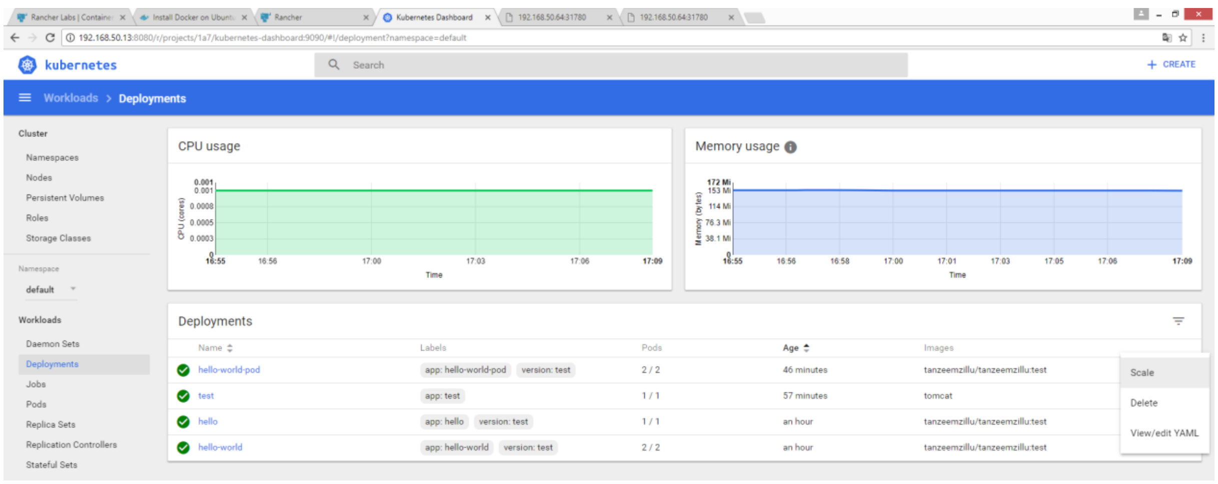
Connection

Cluster IP: 10.43.108.63
Internal endpoints: hello-world-pod:80 TCP
hello-world-pod:31780 TCP

Figure 4.59: Port to access the service

In this case the port is 31780.

Also, it is easy to scale the pod in a service. This is shown in below figure 4.60

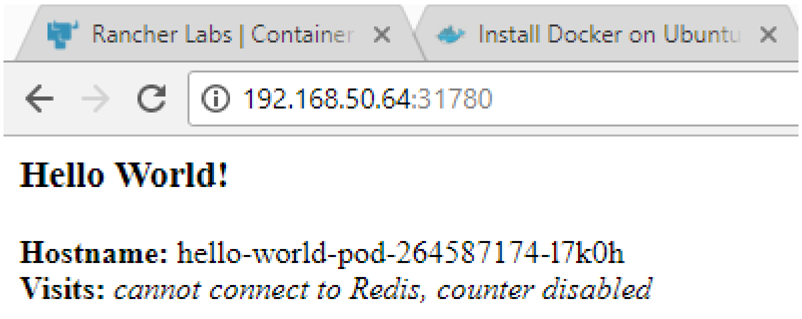


Deployments

Name	Labels	Pods	Age	Images
hello-world-pod	app: hello-world-pod version: test	2 / 2	46 minutes	tanzeemzillu/tanzeemzillu:test
test	app: test	1 / 1	57 minutes	tomcat
hello	app: hello version: test	1 / 1	an hour	tanzeemzillu/tanzeemzillu:test
hello-world	app: hello-world version: test	2 / 2	an hour	tanzeemzillu/tanzeemzillu:test

Figure 4.60: Scaling in Kubernetes in rancher

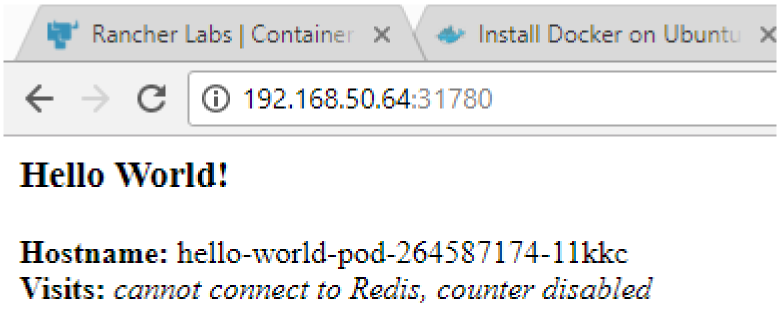
Also, the built-in load balancing is working fine for Kubernetes as shown in figure 4.61 and 4.62



Hello World!

Hostname: hello-world-pod-264587174-17k0h
Visits: *cannot connect to Redis, counter disabled*

Figure 4.61: Load balancing in Kubernetes in rancher (1)



Hello World!

Hostname: hello-world-pod-264587174-11kkc
Visits: *cannot connect to Redis, counter disabled*

Figure 4.62: Load balancing in Kubernetes in rancher (2)

In figure 4.61 and 4.62 it is seen that when a user request for a service through `http:192.168.50.64:31780` then he/she is getting response from different container in different time.

5 Summary and Perspectives

In this thesis work a prototype has been developed to evaluate the facilities of both Docker swarm and Kubernetes based on few use cases.

5.1 Summary

For both Docker swarm and Kubernetes, two separate clusters have established and then tested different scenario based on few use cases. The use cases are in below

- Setting up cluster
- Networking in between cluster
- Deployment of application
- Auto scaling of application
- Load Balancing
- Container update and Rollbacks
- Live migration
- Performance analysis

Both cluster have four VM where one VM is working as a manager and the other three is working as worker nodes.

After testing all these use cases a result has found which evaluate few points of both Docker swarm and Kubernetes. This is shown in below table 5.1

Table 5.1 : Evaluation between Docker swarm and Kubernetes

Evaluation point	Docker swarm	Kubernetes
Docker Version	Docker version 1.12 and later supports Docker swarm	Kubernetes supports Docker version 1.12.3 and all the earlier version
Deployment of application	In Docker swarm, global and replicated service is possible to deploy	In Kubernetes, it is possible to deploy the application in a specific node
Scaling	Capable	Capable
Load balancing	Docker swarm has a built-in load balancing facility	Kubernetes has a built-in load balancing facility
Live migration	Not supported yet	Not supported yet

Beside this, the node performance was checked during running the application and cgroup was used to allocate resources such as CPU against every container.

This thesis work examined many perspective of Docker swarm and kubernetes. Though both are container orchestrator and dedicated to work with docker container but there are some differences in between these two technologies.

It is already shown that, kubernetes does not support docker version which is greater than 1.12. On the other hand, from docker version 1.12 docker swarm comes along with docker by default. So, it is not possible to use any latest docker version with kubernetes. Docker is adding many features quite regularly and those are available with the latest version of docker. It is important to keep using the updated version which is not possible with kubernetes.

Setting up docker swarm cluster is easy. If a user use docker version greater than 1.12 then he/she gets all the facilities of swarm mode along with docker. Also adding the node in docker swarm cluster is very easy. On the other hand, kubernetes needs manual configuration. After comparing between both it is seen that setting up kubernetes cluster is relatively complex.

Application deployment is plain and simple in both cluster. In docker swarm it is possible to deploy global service and replicated service. But docker swarm is not the best option to deploy a service in a specific node. Regarding deployment an application in a specific node kubernetes is a really good solution. Though kubernetes does not support global service.

Both orchestrator have built-in load balancing feature. Docker swarm is maintaining the load by using ingress network between the service. Built in load balancing in kubernetes works based on node port and kube-proxy. Also, kubernetes have another load balancing which is ingress load balancer. It is available on google compute engine. Both docker swarm and kubernetes also supports external load balancing.

5.2 Future Scope and Conclusion

From the above discussion, it is seen that both orchestrator has some advantages and disadvantages. It depends on the user which one he/she prefers. But in many cases, it is useful to have both the facilities. So, in this situation Rancher can be handy. It gives user the scope to use both Docker swarm and Kubernetes. It eases the establishment of the cluster and gives graphical interface to use Docker swarm and Kubernetes. It is not necessary to build the cluster for Docker swarm or Kubernetes from the scratch. Moreover, it gives other options such as health check and so on.

This thesis work creates some scopes which can be done in further research work. There are scopes to analyses the performance between docker swarm and kubernetes based on some other points such as how many containers can be scaled in a cluster and how many nodes can be supported by one master node in both docker swarm and kubernetes. Also, building the container health checking system from master could be done in further work. In this work built in load balancing has examined. It is also possible to do more complex load balancing scenario by using external load-balancer.

The target of this thesis work was to show and explain the architecture and the features of docker swarm and kubernetes. In the theoretical part, the background and technical aspects of these technologies has been discussed. In the realization, some experiment has done based on defined use case scenario which eventually helps to evaluate the performance between docker swarm and kubernetes. In the conclusion, there is a discussion about some advantages and disadvantages of these technologies. Also, some further work scope has been defined which may examined in future.

6 Abbreviations

A	
API	Application program interface
C	
CLI	Command-line interface
CPU	Central processing unit
D	
DDC	Docker Datacenter
DTR	Docker Trusted Registry
H	
HTTP	Hyper Text Transfer Protocol
DTR	Docker Trusted Registry
I	
IPC	InterProcess Communication
IaaS	Infrastructure as a Service
J	
JSON	JavaScript Object Notation
L	
LXC	Linux Container
L	
MNT	Mount
P	
PID	Process ID
PaaS	Platform as a Service
R	
REST	Representational State Transfer
REHL	Red Hat Enterprise Linux
S	
SaaS	Software as a Service

U

Unix	Uniplexed Information and Computing Service
UTS	Unix Timesharing System
UnionFS	Union file system

V

VM	Virtual machine
Vserver	Unix Timesharing System

V

VM	Virtual machine

7 References

1. Docker (2017): *What is a Container,* https://www.Docker.com/what-container [accessed 26 July 2017]
2. Dan Kelly (2016): A History of Container Technology, https://blog.containership.io/a-history-of-container-technology [accessed 26 July 2017]
3. Docker (2017): *Docker overview,* https://docs.Docker.com/engine/Docker-overview [accessed 28 July 2017]
4. Docker (2017): *Use the AUFS storage driver,* https://docs.Docker.com/engine/userguide/storagedriver/aufs-driver [accessed 28 July 2017]
5. Docker (2017): *Use the BTRFS storage driver,* https://docs.Docker.com/engine/userguide/storagedriver/btrfs-driver [accessed 28 July 2017
6. Docker (2017): *Use the Device Mapper storage driver,* https://docs.Docker.com/engine/userguide/storagedriver/device-mapper-driver [accessed 28 July 2017]
7. Azhagappan, D. (2015): *Deploy and Test Open Platform for Network Function Virtualization (OPNFV) within the laboratory environment,* Frankfurt: Frankfurt University of Applied Sciences.
8. Menychtas A., Gatzioura A. and Varvarigou T., (2011): A Business Resolution Engine for Cloud Marketplaces, *Cloud Computing Technology and Science (CloudCom)*, 29th November 2011, [online] Available at http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=6133177 [accessed 30 July 2017]
9. Christopher Tozzi (2017): *What is a Container Orchestrator, Anyway?,* https://containerjournal.com/2017/05/29/container-orchestrator-anyway [accessed 30 July 2017]
10. Janakiram MSV (2016): *From containers to container orchestration,* https://thenewstack.io/containers-container-orchestration [accessed 30 July 2017]
11. Docker (2017): *Swarm mode key concepts,* https://docs.Docker.com/engine/swarm/key-concepts/ [accessed 31 July 2017]
12. Rajdeep Dua (2017): *Docker Swarm,* https://www.slideshare.net/rajdeep/Docker-swarm-introduction [accessed 31 July 2017]
13. Docker (2017): *How nodes work,* https://docs.Docker.com/engine/swarm/how-swarm-mode-works/nodes [accessed 31 July 2017]
14. Ongaro D. and Ousterhout J.: In Search of an Understandable Consensus Algorithm (Extended Version), *Stanford University*, [online] Available at https://raft.github.io/raft.pdf [accessed 31 July 2017]
15. Docker (2017): *How services work,* https://docs.Docker.com/engine/swarm/how-swarm-mode-works/services [accessed 31 July 2017]

16. Kubernetes (2017): *Production-Grade Container Orchestration,* https://Kubernetes.io [accessed 31 July 2017]

17. Kubernetes (2017): *Concepts,* https://Kubernetes.io/docs/concepts [accessed 31 July 2017]

18. Kubernetes (2017): *Concepts,* https://Kubernetes.io/docs/concepts/architecture/nodes [accessed 1 September 2017]

19. Kubernetes (2017): *Concepts,* https://Kubernetes.io/docs/concepts/architecture/master-node-communication [accessed 1 September 2017]

20. Kubernetes (2017): *Concepts,* https://Kubernetes.io/docs/concepts/workloads/pods/pod-overview [accessed 1 September 2017]

21. Kubernetes (2017): *Concepts,* https://Kubernetes.io/docs/concepts/workloads/controllers/replicaset [accessed 2 September 2017]

22. Kubernetes (2017): *Concepts*, https://Kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller [accessed 2 September 2017]

23. Kubernetes (2017): *Concepts*, https://Kubernetes.io/docs/concepts/services-networking/service [accessed 2 September 2017]

24. Kubernetes (2017): *Concepts*, https://Kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors [accessed 2 September 2017]

25. Docker (2017): *Overview of Docker Compose,* https://docs.Docker.com/compose/overview [accessed 3 September 2017]

26. Putty (2017): *Putty,* http://www.putty.org [accessed 4 September 2017]

27. Docker hub (2017): *tomcat,* https://hub.Docker.com/_/tomcat [accessed 4 September 2017]

28. Docker hub (2017): *mysql,* https://hub.Docker.com/_/mysql [accessed 4 September 2017]

29. Docker (2017): *Docker service create,* https://docs.Docker.com/engine/reference/commandline/service_create [accessed 10 September 2017]

30. Docker (2017): *Services and tasks,* https://docs.Docker.com/engine/swarm/key-concepts/#services-and-tasks [accessed 10 September 2017]

31. Redhat (2017): *Introduction to control group,* https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html [accessed 12 September 2017]

32. Rancher Labs (2017): *Simple, easy-to-use container management,* http://rancher.com [accessed 13 September 2017]

33. Rancher Labs (2017): *Overview of Rancher,* http://rancher.com/docs/rancher/v1.6/en [accessed 13 September 2017]

34. Rancher Labs (2017): *Getting Started with Hosts,* http://rancher.com/docs/rancher/v1.6/en/hosts/#supported-Docker-versions [ac-cessed 13 September 2017]

8 Appendix

The attachment of current research work is the CD, which contains following

- Thesis document in .pdf format
- Thesis document in .doc format

Attached CD/DVD content / Inhalt der CD/DVD