Elastic Deployment and Management of Big Data Platforms using Linux Containers

Undergraduate Thesis

Submitted in partial fulfillment of the requirements for the award of the Degree of

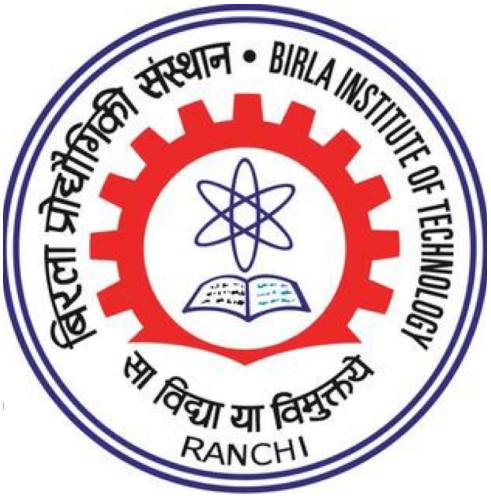
BACHELOR OF ENGINEERING
in
INFORMATION TECHNOLOGY

By

Sarthak Sharma
Roll No. BE/10657/2013

Under the supervision of:

Prof. Yogesh Simmhan
&
Prof. D. Mustafi



*During work as a Research Intern at DREAM:Lab, IISc Bangalore
April 2017

Declaration of Authorship

I, Sarthak Sharma, declare that this Undergraduate Thesis titled, 'Elastic Deployment and Management of Big Data Platforms using Linux Containers' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Certificate

This is to certify that the thesis entitled, "*Elastic Deployment and Management of Big Data Platforms using Linux Containers*" and submitted by Sarthak Sharma Roll No. BE/10657/2013 in fulfillment of the requirements of IT8020 Report embodies the work done by him under my supervision.

Supervisor
Prof. Yogesh Simmhan
Asst. Professor,
IISc, Bangalore
Date:

Co-Supervisor
Prof. D. Mustafi
Associate Lecturer,
BIT Mesra
Date:

"When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong. "

– R. Buckminster Fuller

BIRLA INSTITUTE OF TECHNOLOGY , MESRA

Abstract

Bachelor of Engineering

Elastic Deployment and Management of Big Data Platforms using Linux Containers

by Sarthak Sharma

Distributed Big Data platforms like Hadoop and Giraph provide a simplified programming interface to solve various real-world problems. These are often run on commodity clusters or Cloud Virtual Machines (VMs), and typically with a fixed configuration or image. One major challenge is to reconfigure the same cluster of machines for diverse Big Data platforms, based on the current need, rather than a static setup. This would provide an easy and efficient way to deploy the complex setups across various machines. This can also prove beneficial when the setup has to be scaled up or down, so as to not succumb to the increasing load or waste resources. While VMs with custom images can be spun up, their startup time is on the order of minutes and not agile.

In this thesis, we leverage containers to enable easier management of Big Data platforms, on-demand. Specifically, we aim to provide a quick deployment and scale-up which will allow the user to setup a complex clustered distributed platform with the provision to add new nodes in the minimum amount of time. Later, this can be used for efficient and dynamic balancing strategies by schedulers. Not only will it help in making our system cost efficient but also enable us to scale according to the dynamic requirements of the system.

This study focuses on providing the scalability and manageability at the infrastructure level using Linux containers, and targets distributed graph processing platforms such as Apache Giraph, Apache Hama, and GoFFish. We explore the functionalities of Docker as it provides a relatively easy and quick solution in the form of its containerised application setup. Its version control system and a global image repository makes the setup and management of a service in a system relatively easy. The outcome of this project allows the rapid and easy deployment of the graph processing platforms on a cluster of VMs or commodity nodes and reduces the time and effort for users to operate on large graph data sets and complex algorithms using such platforms.

Acknowledgements

I owe my deepest gratitude to my guide *Prof. Yogesh Simmhan*, for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. He made sure that I learnt by practice, always motivating me to think a step further, work a little harder. My interactions with him always resulted in newer ideas which proved beneficial towards my research.

I am indebted to my co-guide, *Prof. D. Mustafi* without whose constant presence and supervision my work would not have been successful. I especially acknowledge the many useful discussions I had with my seniors (*Anshu*, *Ravi*, *Jayanth*, *Abhilash*, *Rajrup* and *Aakash*) and fellow interns (*Diptanshu* and *Himanshu*), here at DREAM:Lab, IISc that helped me understand some of the subtle technicalities of the problem which further helped me contribute to my research in a better way.

This thesis is dedicated to my parents and family for keeping faith in me. Without their support, none of this would have been ever possible.

Contents

List of Figures

Abbreviations

VM	**V**irtual **M**achine
OS	**O**perating **S**ystem
LXC	**L**inu**x** **C**ontainers
BSP	**B**ulk **S**ynchronous **P**arallel
IoT	**I**nternet **o**f **T**hings
ISA	**I**nstruction **S**et **A**rchitecture

Chapter 1

Introduction

1.1 Motivation

With the advent of Cloud Computing and Software-defined everything, there has been an increasing shift towards the use of distributed systems in production. A cluster of machines enables scaling-out when performing a task by working in distributed mode across cheaper commodity (physical or virtual) machines rather than scaling-up on a costlier or custom larger machine to perform the computation. Cloud and cluster computing have thus grown popular for Big Data platforms that are designed to weakly-scale, and these offer opportunities for system-level research problems in managing and optimising the resource allocation and usage.

Distributed System offer crucial functionalities, yet also pose unique challenges. These pros include scalability and fault tolerance, while additional overheads come through distributed coordination, version control and latency. Big Data platforms have tried to leverage these advantages and also address the limitations that are imposed. One of the major concern with using these large scale distributed systems is the non-trivial deployment and configuration of such clustered applications and their management.

Since the distributed setup offers us numerous advantages, it is justified to try and solve the shortcomings of the system thereby creating an all round setup which caters to all the important aspects of a scalable big data application in today's sense. Where scalability is involved, we cannot just rely on the bare metal setups as it would not be possible to physically add and remove nodes as per requirement to a cluster. Here comes a need for Cloud distributed setups which essentially gives us an option of Cloud Virtual Machines (VMs). Also, a relatively new

technology (Containers) can also be used for the same purposes which provide us with the functionalities of virtual machines along with additional benefits.

1.2 Virtual Machines vs Containers

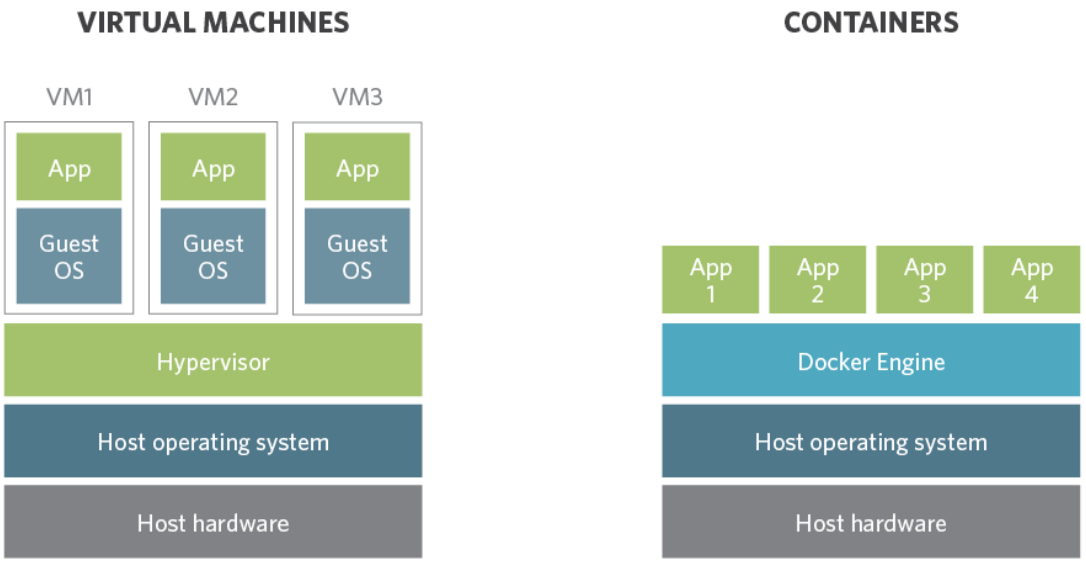


FIGURE 1.1: Virtual Machines vs Conatiners [9]

With the increase in compute capacity and processing power of bare metal servers, Virtual Machines were introduced as a means to divide, sandbox, and share the resource capacity of the underlying machine with multiple users. VMs are designed to run on these bare metal setups, with the applications running within the VM being emulated, which translates instructions from one ISA to another, or the instructions being run natively using hardware virtualization support that is now common. This process of emulation is done by a *Hypervisor* (also called as a virtual machine monitor), which is a layer which sits between the host Operating System and the guest Operating System, and is responsible for orchestrating the VM and its environment. It is due to the hypervisor, that the VMs support different Operating Systems (OSes) compared to the one being executed by the host machine. This allows us to have a Unix based VM alongside a Linux-based one on top of a Windows host machine. The VMs themselves are instantiated from an image whose size can span many GBs, and each VM instance consumes discrete quantity of CPU and memory resources for running the entire guest OS and their applications.

Containers, on the other hand, work on the principle of OS-level virtualization, leveraging the host kernel's capabilities. Here, the OS kernel – their binaries and libraries – is shared with the software specific to that container being ported and executed within this container. This allows containers to be executed on different host OS distributions, as long as their kernel versions are the same. Each container writes on its own unique mount/space called as the namespace of that container. All the security functionalities, permissions, and network characteristics are implemented using the host kernel making the container exceptionally "light-weight", with their image size and memory footprint typically in the order of MB(s). Another advantage they have is in terms of the speed of spinning up a container, where it takes minutes to get a virtual machine up and running but containers can start within seconds due to no booting time required for the OS. This also allows us to increase the density of containers in a single host machine, allowing many more containers to be started than a heavy-weight VM.

Containers can be shared across different machines in the form of images and can be deployed in numerous scenarios, including cluster nodes, Cloud VMs and even Raspberry PIs, thereby accelerating the development process by quickly packaging applications along with their dependencies. Additionally, their property of sharing a common OS reduces management overhead as only a single OS on the host machine needs to be managed. However, we cannot run a container with a guest OS that differs from the host OS because of the shared kernel, i.e., a Window container cannot sit on top of a Linux-based host.

In summary, Containers bring in **Density** (Higher utilisation of underlying hardware), **Speed** (Typically takes less than a second to start a fully functional container), **Low Overhead Management** (If host kernel is updated, all the containers running on it are updated) and **Portability** (Encapsulation of a container and all its resources makes it easier to migrate or move).

1.3 Containers for Big Data Platform Management

Containers use OS-level virtualization enabling us to run multiple isolated application environments on the same host kernel. They can be classified into two broad categories:

Operating system (OS) containers – share the host operating system's kernel, generally created from templates and images. The OS environment configuration can be the same across all OS containers. E.g., **LXC** allows the running of multiple isolated Linux containers on a

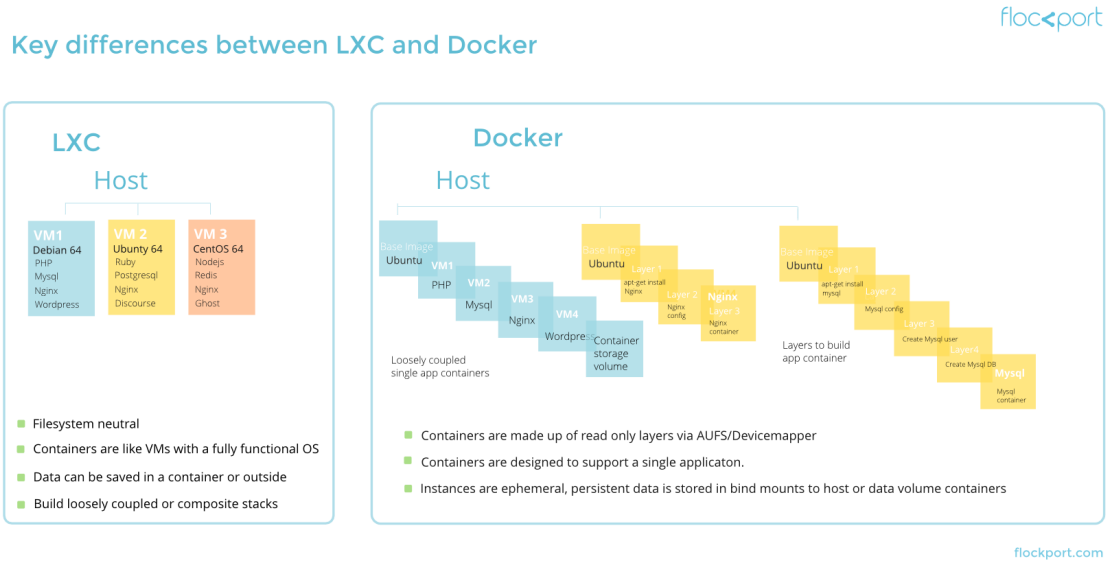


FIGURE 1.2: Linux Containers (LXC) vs Docker [4]

single LXC control host and provides many of the functionalities of a VM and hypervisor, apart from the fact that only Linux platforms are supported.

Application containers – contain application services with their resources such as required libraries. Application containers have a base layer that is common among containers working with the same application and have added layers for commands that are run in each container. When a container is run, all of the layers are combined and run in a single process. **E.g., Docker** which was typically designed to serve as a single application container, allowing the user to run their desired application with process-id being 1

LXD is an extension to the concept of LXC and behaves like a full hypervisor but eliminates the overhead of virtualization or machine emulation. It is described as a *"daemon exporting an authenticated representational state transfer application programming interface (REST API) both locally over a Unix socket and over the network using https. There are then two clients for this daemon, one is an OpenStack plugin, the other a standalone command line tool."*[5]

It provides quite a few interesting capabilities out of the box, like the provision of Live-Migration of containers from one machine to another, Secure and dense deployment of containers and their efficient management and also possible integration with docker containers, OpenStack, and Kubernetes. Although it provides a fast, dense and secure solution to container management and is compatible with almost all container technologies, one major constraint for its usage is

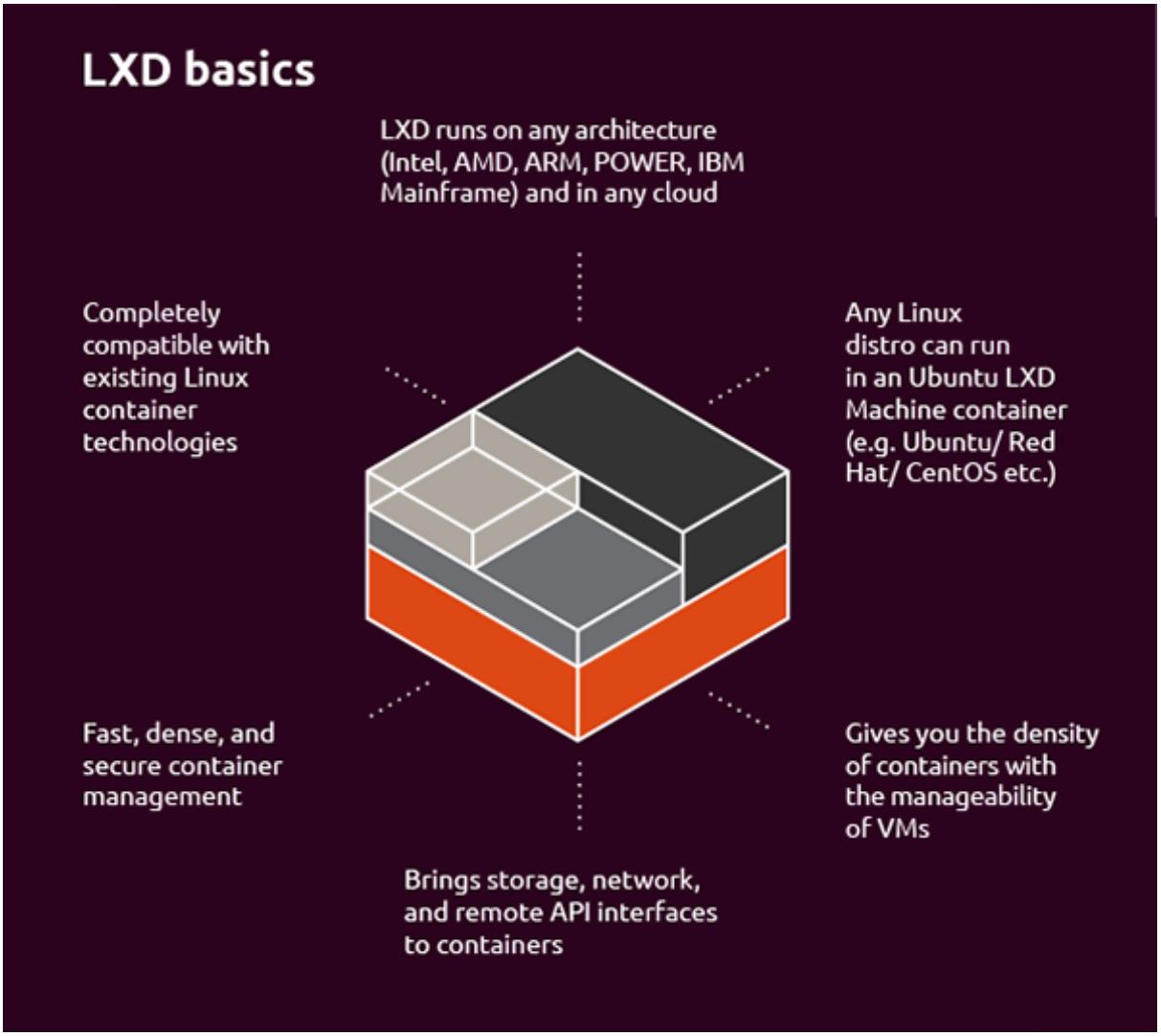


FIGURE 1.3: LXD : Working [6]

that the host system needs to have an Ubuntu OS running underneath. Having said that, it is compatible with almost all the architectures (Intel, AMD, ARM, POWER and IBM Mainframe) and can run any Linux based operating system (Ubuntu, Redhat, CentOS, etc) on top of it as the guest OS.

1.4 Docker

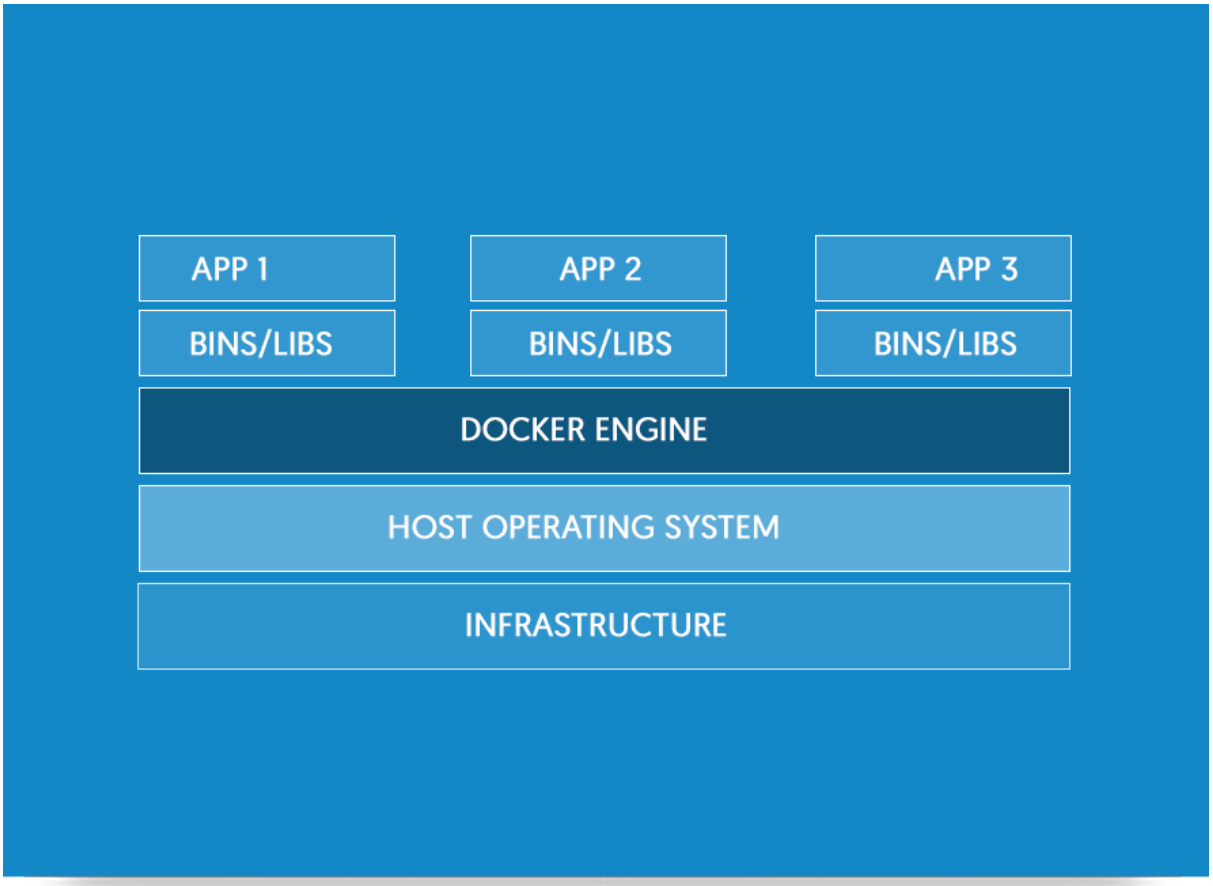
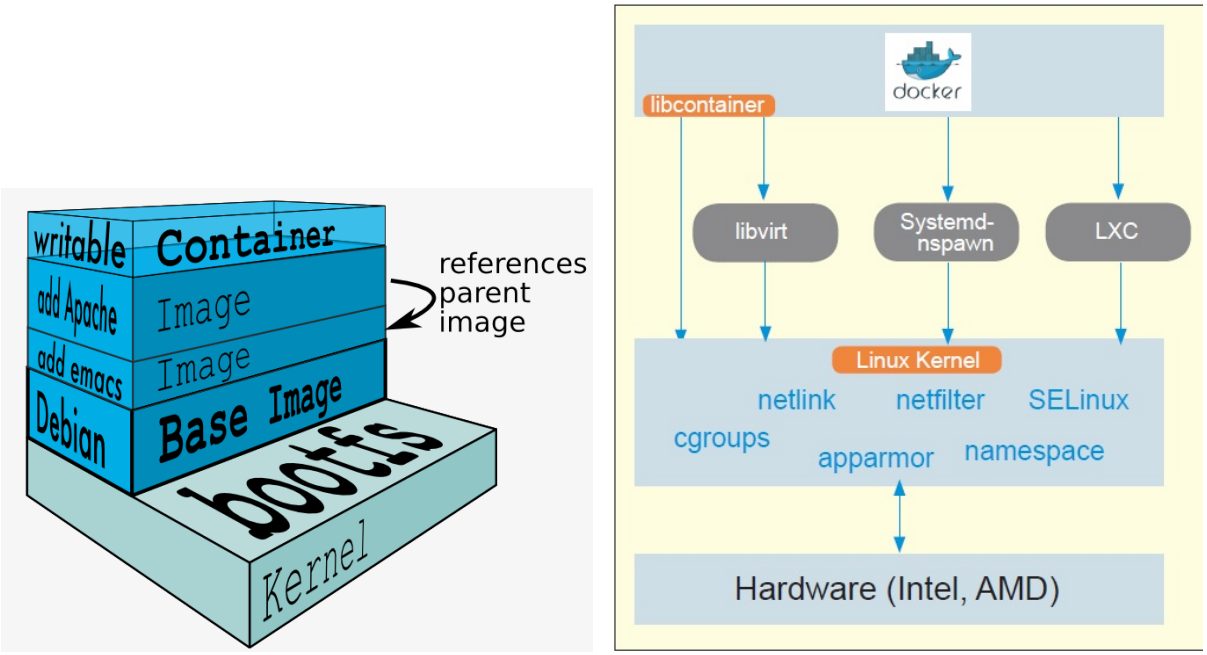


FIGURE 1.4: Docker Container: High level overview [10]

Docker is an application level container that automates the repetitive tasks of setting up and configuring development environments thereby enabling the application to be run, easily deployable, shareable, built and run by isolating software from its surroundings. It follows a layered approach in terms of the images formed of various containers on top of one another, creating a complete bundled application. This approach, though a little complex, allows reusability of various pre-built images. It also has a global public repository (Docker-Hub) which keeps account of various images and their specific versions, providing a way to not only access an image directly from the web but also offering version control along with it. Docker uses a union fs, that joins a static read-only image (that can be used by other containers) with that particular container filesystem, where all you write of the original is copied. Inherently focussed at being a single application container, docker runs the target application as PID 1. Thus, there is no reliability on traditional UNIX functionalities like cron, Syslog, logrotate; or even for that



(A) Layering of Images in a Container [3] (B) Docker internals [1]

FIGURE 1.5: Docker Internals and working with layered images

matter cleaning up grandchild processes after terminated children. Docker out of the box has the provision to manipulate namespaces, cgroups (control groups), apparmor (security)profiles, network interfaces and firewall rules, all of which is done in a consistent and predictable way without being dependent on Linux Containers (lxc) or any other package, like it used to do in some of its previous releases which confined it to only Linux based containers. But with the use of libcontainers (which manages all the namespaces, cgroups among other important functionalities) docker has increased its scope by providing its support across other Unix and Windows based platforms. In this work, we consider Docker as our preferred container platform.

Chapter 2

Distributed Graph Platforms

2.1 Introduction

Vertex-centric abstractions like Pregel and Apache Giraph are popular for composing network algorithms and their distributed execution over large graph structures with weak scaling. But they suffer from performance limitations due to longer time to converge for asymmetric graph algorithms or higher network costs. GoFFish is a subgraph-centric platform that has addressed these limitations [8]. We work with *GoFFish v3*, which has been redesigned and built on top of *Apache Hama* [7], and offers a graph processing framework to run distributed batch algorithms across billions of vertices and edges. This framework supports storing graphs, and composing and executing graph analytics. It also helps to design optimized graph algorithms that leverage the Hadoop distributed file storage system. The composed application enhances data parallel analytics at scales far superior to traditional MapReduce models using a novel distributed data partitioning approach based on edge distance heuristics and also minimizes communication overhead by grouping together tightly bound data. Hence, GoFFish provides a faster convergence and improved weak scaling with the size of the graph using a subgraph-centric abstraction, compared to the contemporary Apache Giraph platform.

Deploying and managing a Big Data platform like GoFFish is complex, and requires multiple software and platform dependencies to be met. These are described below.

While GoFFish does not use the MapReduce capability of Apache Hadoop, it uses Hadoop HDFS in order to load the graphs for the computation. So in order for our framework to run we require a fully functional HDFS deployment with Name Nodes and Data Nodes.

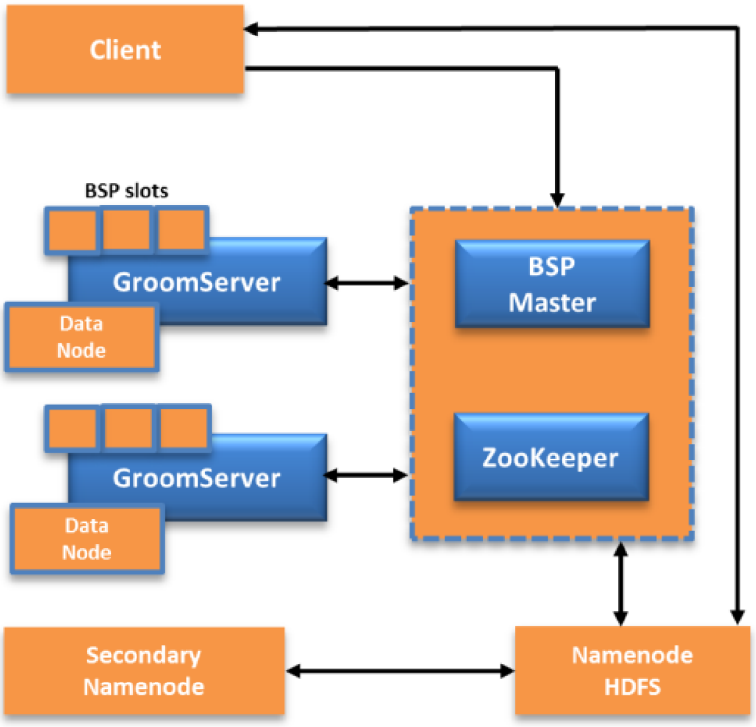


FIGURE 2.1: Apache Hama Architecture [7]

In order to fulfill the same, we deploy Apache Hadoop (v2.7.1) on our containers and configure the Hadoop Distributed File System (HDFS) to have a 1x replication factor, using all other default configurations. In the case of Distributed mode, each worker container runs a DataNode daemon for storing the data into local HDFS, and a master container runs a NameNode daemon to maintain the metadata information of all files in HDFS. Whereas for a pseudo-distributed setup using a standalone docker, both the DataNode and the NameNode are running in the same container.

Instead of developing the GoFFish framework from scratch, it uses a general data processing distributed framework Apache Hama which provides the right set of abstractions which are then leveraged in creating abstractions for graph processing in GoFFish. Hama uses the Bulk Synchronous Parallel (BSP) computing model to operate iteratively as a series of supersteps across distributed machines, and is written in Java. Hama services include the BSP master and Groom serves running on various node(s). GoFFish runs as an application on top of Apache Hama which means all the dependencies and jar files are also needed for execution.

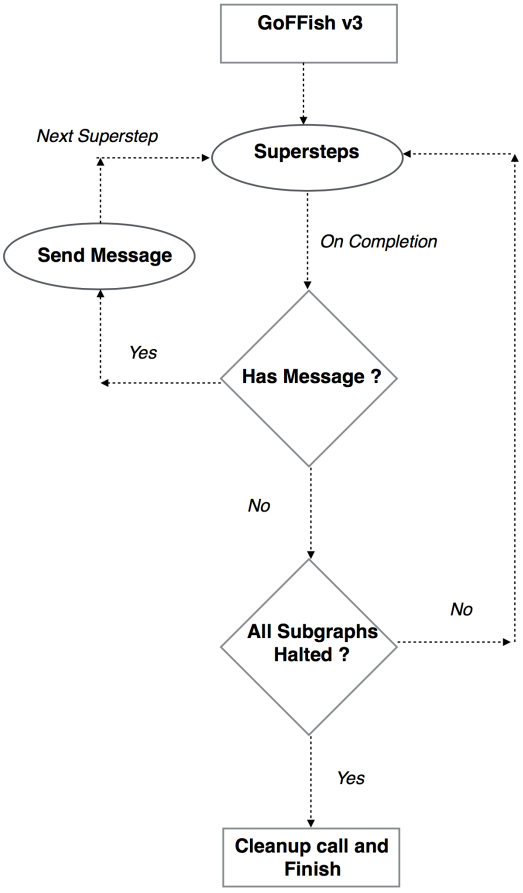


FIGURE 2.2: Sequence of operations for a GoFFish application

2.2 Using Docker Capabilities

Docker works on the principle of layering of images, which are built/deployed using a Dockerfile that defines all the dependencies and packages/tools present in an image (e.g., The OS, installed tools, software and the ports exposed to the user are all defined inside the Dockerfile). It can also link to a previous, already built image which it uses as a reference for it to build on top of, thereby ensuring a layered hierarchy.

Sometimes, we need more than one container interacting in order to collectively perform a task/computation. Here we do not consider just the container but a service as a whole (which is a combination of various containers interrelated to each other performing their own specified tasks). **Docker Compose** makes it possible for us to deploy a service as a whole on a single machine. This is done by the use of a Docker-Compose file which is a YAML file that defines all the services inside it and dependencies that one container has on another. Once executed, it

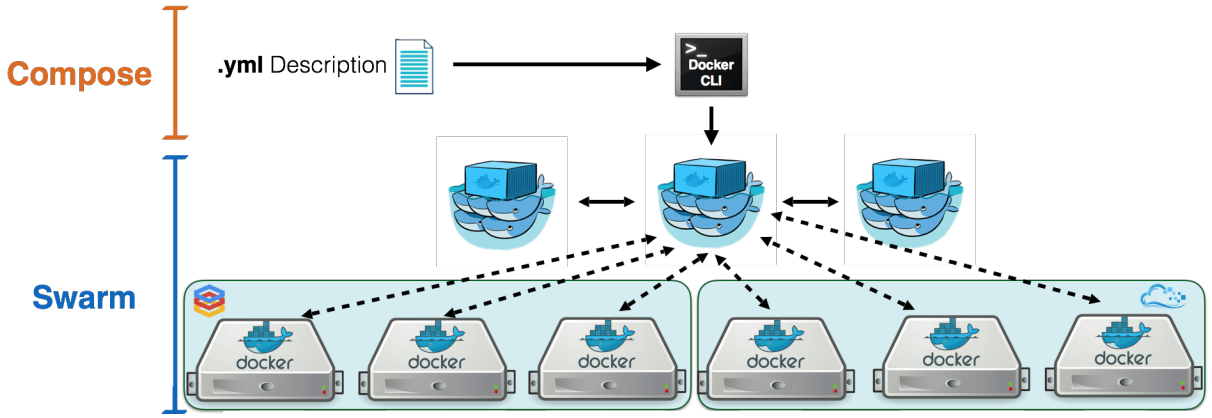


FIGURE 2.3: Integrating Docker Compose with Swarm [2]

automatically fetches/builds the images and deploys the service (consisting of all the containers defined inside it along with their dependencies).

However, if we need to deploy a distributed application (Like Apache Hadoop and Hama) we typically need more number of machines/nodes interconnected to each other on which we can deploy our service making it span across multiple machines with multiple containers being run on each of them, working together in distributed mode. **Docker Swarm** comes in handy in such scenarios as it allows us to combine multiple virtual/bare metal machines and combines them to function as a single machine. Here a Swarm Manager node orchestrates all the services which are to be deployed across numerous Swarm Worker nodes and maintains the running state of each node thereby also providing Scalability, Security, High Availability and Load Balancing functionalities to the system.

2.3 Managing GoFFish v3-h Platform using Docker

Now that it is clear that we need to build a docker image for GoFFish, we need to first make sure that all the prerequisites in terms of dependencies are met which include a fully functional Hadoop HDFS and Apache Hama setup which in turn is dependent on compatible versions of Java on a suitable Operating System with correct configurations and permissions for it to work properly. We thus chose CentOS (6.5) as our base Operating System image due to its stability with Java (8), Hadoop (2.7.1) and Hama (0.7.1) on top of each other. Once the prerequisites were met, we then added some sample graphs for testing and an executable, pre-built jar for the GoFFish application (in case of the Bin Docker image) and on-the-fly latest building functionality of the jar executable (in case of the Source Docker).

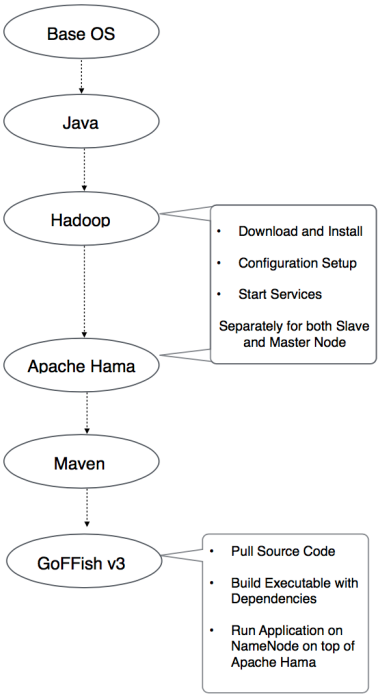


FIGURE 2.4: GoFFish without Docker

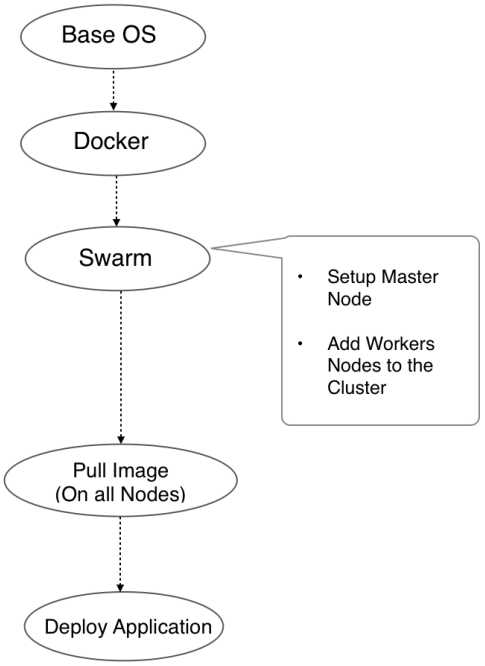


FIGURE 2.5: GoFFish Setup with Docker

Once the images are built, they need to be deployed as services wherein we have our Name-Node/Master and Data-Nodes/Groom serves defined along with their dependencies. This is where Docker Compose comes into the scenario as it helps in the deployment of our service enabling it to run in distributed mode with multiple containers.

Now that Distributed mode for our service is achieved, we need to deploy it across multiple machines which is taken care by Docker Swarm. Optimal placement strategies and deployment settings enable us to thus provide an elastic, easily managed solution for graph processing platforms.

2.4 Standalone Docker setup for GoFFish v3-h

As explained in the previous section, the Standalone docker is a container running all the services within itself in Pseudo-Distributed mode. The Dockerfile is designed such that all the packages related to Hadoop and Hama, and other GoFFish dependencies are put together in the same container. However, the role of starting these services is taken care by the **bootstrap.sh** kickstart file, which not only starts the desired processes (i.e., the name node, data node, BSP master and the room server in this case) but also establishes a clean user interface for running any goffish job/command by giving us the list of algorithms available for execution and the

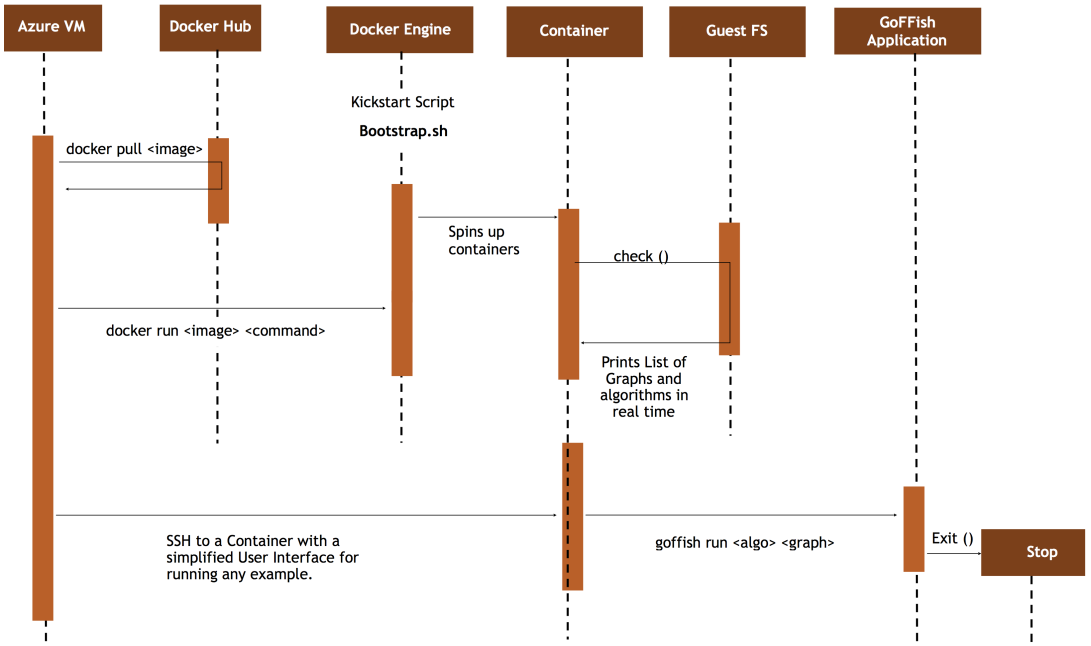


FIGURE 2.6: Sequence diagram representing flow for standalone docker for GoFFish

sample graphs for testing. All of this is dynamic in nature, allowing easy reconfiguration. Being a single container, we are able to pre-loaded various sample graphs onto hdfs during the image build time and hence saved on the graph loading time when we start a container.

The whole process of executing a sample GoFFish example using a standalone docker includes multiple steps that are described in the sequence diagram above.

2.5 Distributed Dockerised setup for GoFFish Hama

While working in a distributed mode, we provide two solutions to the problem of deployment. One being the docker with pre-built executables (i.e., the Bin Docker for GoFFish) and another is the Source Docker, which pulls the latest version from GitHub and builds it on the fly everytime we spin up a new cluster of containers.

Following the Docker layered approach, we built a GoFFish Base image consisting of Hadoop and Hama setups in Distributed mode. This is common to both GoFFish-Bin and GoFFish-Source docker, and is then used as a base to build upon by each of them. Once the image setup is complete, a compose file is built wherein all the services are defined along with some of their properties like replication, etc.

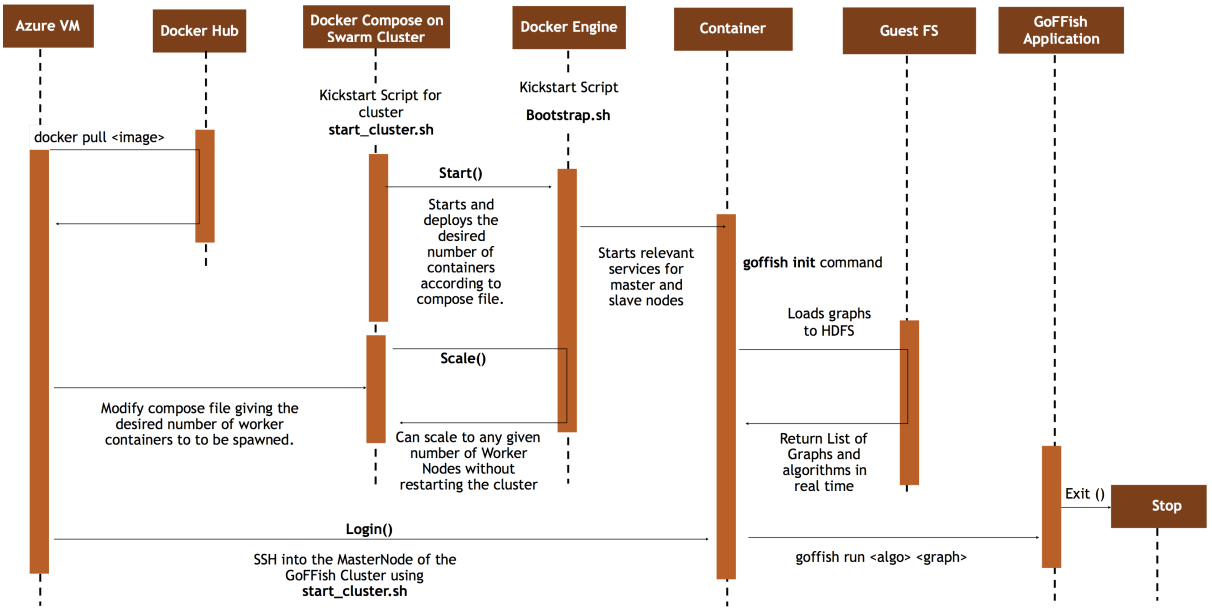


FIGURE 2.7: Sequence diagram representing flow for a GoFFish distributed cluster

One thing to note is that the docker image is same for both the name-nodes/BSP-master and data-nodes/groom-servers – the difference lies in how and where these services are started. This is carried out by our **bootstrap.sh** file, like in the case of standalone docker, as only relevant services are started on specific containers thereby ensuring that they act as a master or slave. Also, all images consist of pre-configured settings files for Hadoop and Hama, and hence all containers which are started know the address of the master node and can directly contact it to get added to the cluster. This also enables us to scale-out on the go and add new containers on the fly thereby providing easy scalability to our application.

Considering that we have already set up a Swarm cluster, our specialized bash scripts can deploy our application onto the Swarm cluster, and can further be used to start, stop, scale, visualize and login to the cluster master node from where we run all our sample examples.

Upon logging in to the master node, we need to initialize the cluster, which involves loading the graphs onto the hdfs. As this setup is distributed, we can not load the graphs until the cluster is set up. We also fetch and build the executable jar in the case of Source Docker. On execution of this command, we are again shown the same user interface for running the goffish command for standalone docker which can be used to run sample algorithms on the list of graphs available to the user.

Chapter 3

Experiments

3.1 Implementation

We use Azure Cloud Virtual Machines on top of which Docker and Docker-Swarm cluster is set up. In case of the standalone docker, we run everything inside one docker container (i.e., both Hadoop and Hama are run in Pseudo-Distributed mode), with both name node and data node (for Hadoop hdfs) and BSP master and groom server (for hama) are packaged together to run inside a single container.

In the case of a Distributed setup, a total of 9 Virtual Machines are used of which 8 are Swarm Worker nodes and 1 being the Swarm Master. Both name node and BSP master are run inside the same container and data node along with the groom server are packaged together inside a different container. Placement Strategies for Containers refrain any container running Data Node and Groom Server to be run on the Master and compulsarily execute the Name Node and BSP Master container on the Swarm Master.

Azure VM Configuration				
No. of VMs	Type of VM	Cores	Memory (Gb)	Disk (Gb (SSD))
4 (1 Master, 3 Workers)	D2	2	7	100
3 (Worker Nodes)	D11	2	14	100
1 (Worker Node)	D3	4	14	200
1 (Worker Node)	D4	8	28	400

Table 3.1: Swarm Cluster Setup (Azure)

Following are a few screenshots related to the aforementioned stages in the process of running a GoFFish Application in Distributed mode:

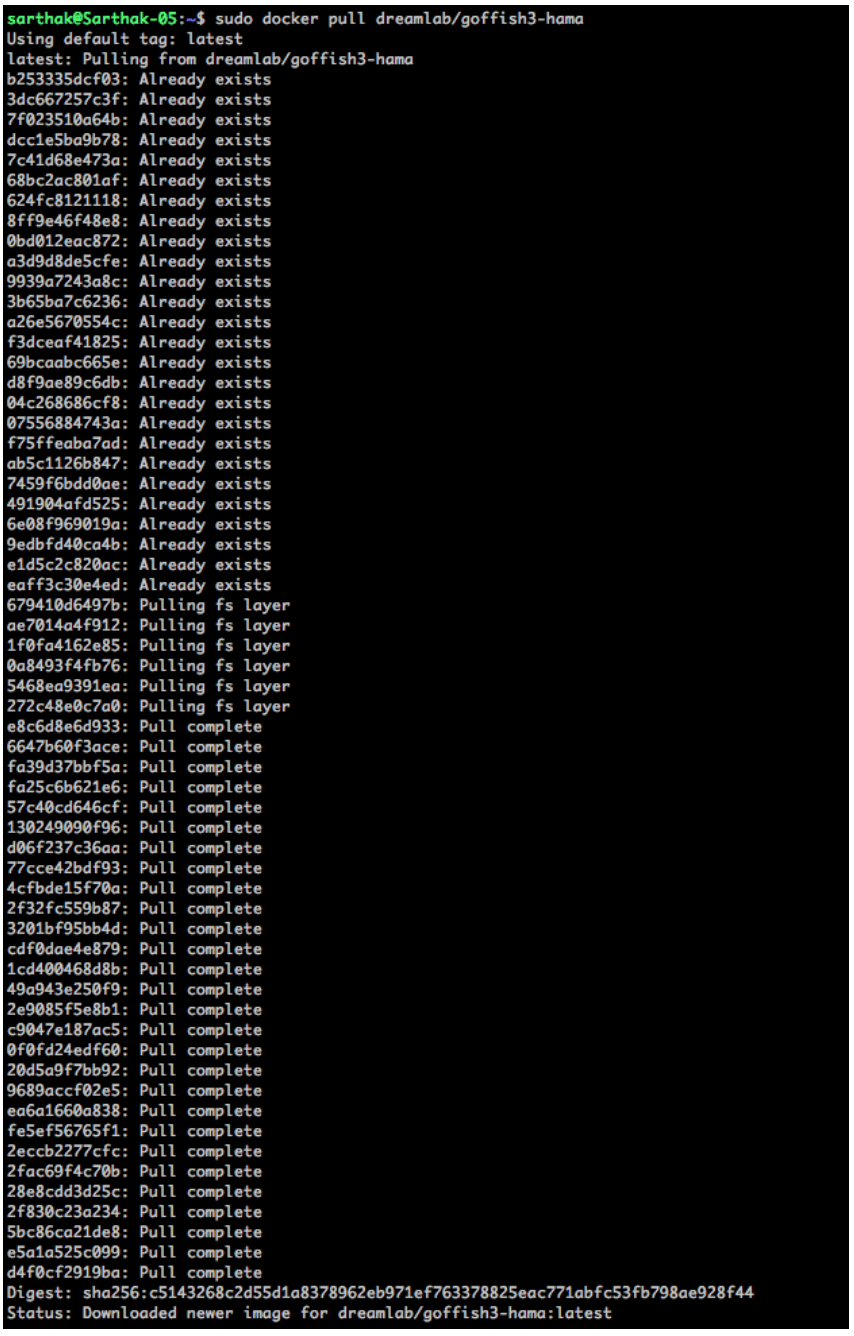


FIGURE 3.1: Pulling an image from the Docker Hub

```yaml
version: '3'
services:
  namenode:
    image: dreamlab/goffish3-hama-bin
    hostname: namenode
    networks:
      - appnet
    ports:
      - "8088:8088"
      - "50090:50090"
      - "19888:19888"
      - "40013:40013"
      - "40015:40015"
    #volumes:
      #- /home/sarthak/Docker_Share:/data
    deploy:
      replicas: 1
      placement:
        constraints: [node.role == manager]
    command: >
      /bin/bash -c "
        /etc/bootstrap.sh -d -namenode;
        while ! nc -z namenode 50070;
        do
          echo waiting ;
          sleep 1 ;
        done;
      "

  datanode:
    depends_on:
      - namenode
    image: dreamlab/goffish3-hama-bin
    networks:
      - appnet
    deploy:
      replicas: 4
      placement:
        constraints: [node.role == worker]
    command: >
      /bin/bash -c "
        while ! nc -z namenode 50070;
        do
          echo waiting ;
          sleep 1 ;
        done;
        /etc/bootstrap.sh -d -datanode;
      "

networks:
  appnet:
    external:
      name: appnet
```

FIGURE 3.2: Docker Compose file for GoFFish

```
#!/bin/bash

if [ $# -eq 0 ]; then
   clear
   echo "USAGE:"
   echo ""; echo ""; echo "";
   echo "./goffish_cluster.sh  start                 --To Start a new Goffish HAMA Cluster (Default: 4 Workers)"; echo "";
   echo "./goffish_cluster.sh  stop                  --To Stop the Goffish HAMA Cluster"; echo "";
   echo "./goffish_cluster.sh  scale  <No. of Workers> --To Scale IN or OUT the Workers"; echo "";
   echo "./goffish_cluster.sh  visualise             --To Start a Dashboard for Swarm Cluster"; echo "" ;
   echo "./goffish_cluster.sh  login                 --To login to the Master Node"; echo "" ;
elif [ $1 == "start" ]; then
   docker service ls | grep namenode
   if [ $? -eq 0 ]; then
       clear
       echo "HAMA CLUSTER ALREADY RUNNING !! "
   else
       sudo docker stack deploy --compose-file=docker-compose.yml hama_bin_cluster
   fi
elif [ $1 == "scale" ]; then
   docker service scale hama_bin_cluster_datanode=$2
elif [ $1 == "login" ]; then
   X=`docker ps | grep -oh "\w*namenode.*"`
   docker exec -it $X /bin/bash --login
elif [ $1 == "visualise" ]; then
   docker ps | grep manomarks/visualizer
   if [ $? -ne 0 ]; then
       docker run -it -d -p 4000:8080 -v /var/run/docker.sock:/var/run/docker.sock manomarks/visualizer
   fi
   clear
   echo "Visualiser Running on Port 4000 ..."
   echo "Opening in Browser ..."
   firefox 0.0.0.0:4000  2>1  &
elif [ $1 == "stop" ]; then
   docker stack rm hama_bin_cluster
fi
```

FIGURE 3.3: Startup script for deployment of GoFFish cluster

```
******************AVAILABLE ALGORITHMS************************

SingleSourceShortestPathJob
TriangleCountJob
VertexCountJob
EdgeListJob
MetaGraphJob
ConnectedComponentsJob
GraphStatsJob
PageRankJob

******************AVAILABLE GRAPHS**************************

facebook-1P
facebook-4P
google-1P
google-4P

******************COMMAND USAGE*****************************

goffish run algorithm input-graph


Refer the "ReadMe" file for Details...


bash-4.1#
```

FIGURE 3.4: Goffish command Interface

```
bash-4.1# goffish  run  VertexCountJob  facebook-4P
17/03/02 13:01:33 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
17/03/02 13:01:34 INFO bsp.FileInputFormat: Total input paths to process : 4
17/03/02 13:01:35 INFO Configuration.deprecation: user.name is deprecated. Instead, use mapreduce.job
17/03/02 13:01:37 INFO bsp.BSPJobClient: Running job: job_201703021259_0001
17/03/02 13:01:40 INFO bsp.BSPJobClient: Current supersteps number: 0
17/03/02 13:02:19 INFO bsp.BSPJobClient: Current supersteps number: 6
17/03/02 13:02:19 INFO bsp.BSPJobClient: The total number of supersteps: 6
17/03/02 13:02:19 INFO bsp.BSPJobClient: Counters: 9
17/03/02 13:02:19 INFO bsp.BSPJobClient:   in.dream_lab.goffish.GraphJobRunner$GraphJobCounter
17/03/02 13:02:19 INFO bsp.BSPJobClient:     ITERATIONS=11
17/03/02 13:02:19 INFO bsp.BSPJobClient:   org.apache.hama.bsp.JobInProgress$JobCounter
17/03/02 13:02:19 INFO bsp.BSPJobClient:     SUPERSTEPS=6
17/03/02 13:02:19 INFO bsp.BSPJobClient:     LAUNCHED_TASKS=4
17/03/02 13:02:19 INFO bsp.BSPJobClient:   org.apache.hama.bsp.BSPPeerImpl$PeerCounter
17/03/02 13:02:19 INFO bsp.BSPJobClient:     SUPERSTEP_SUM=24
17/03/02 13:02:19 INFO bsp.BSPJobClient:     TIME_IN_SYNC_MS=3600
17/03/02 13:02:19 INFO bsp.BSPJobClient:     IO_BYTES_READ=877486
17/03/02 13:02:19 INFO bsp.BSPJobClient:     TOTAL_MESSAGES_SENT=75
17/03/02 13:02:19 INFO bsp.BSPJobClient:     TASK_INPUT_RECORDS=4039
17/03/02 13:02:19 INFO bsp.BSPJobClient:     TOTAL_MESSAGES_RECEIVED=75
bash-4.1#
```

FIGURE 3.5: GoFFish command execution inside Docker

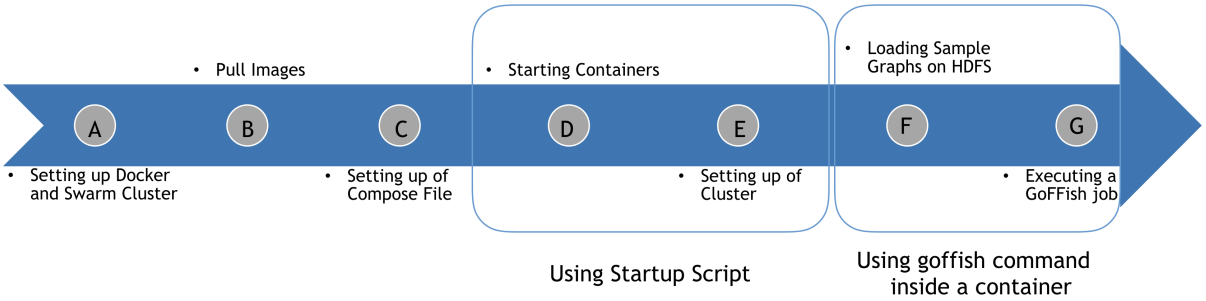


FIGURE 3.6: Timeline Representation for running an application on GoFFish Cluster (Distributed Mode)

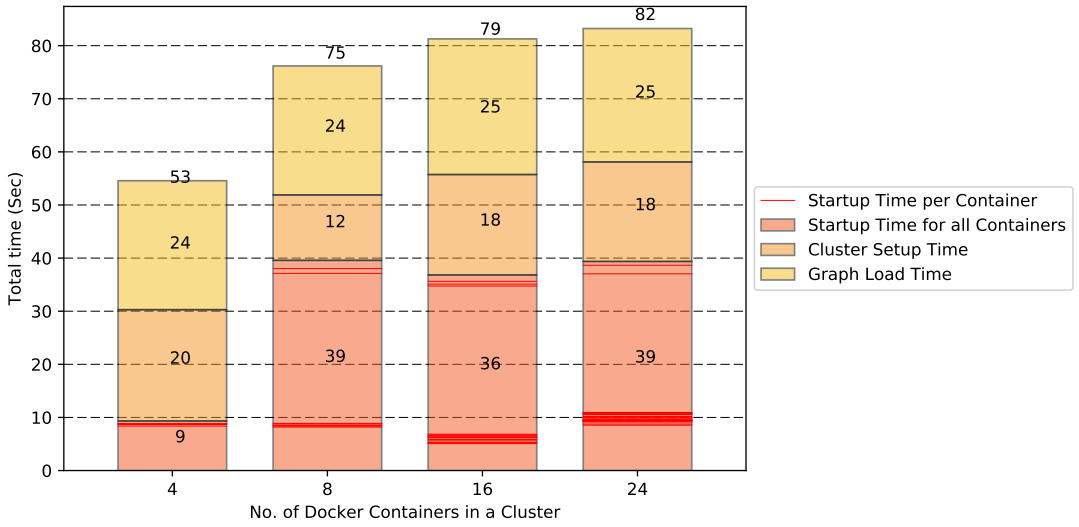


FIGURE 3.7: Stacked Bar Plot for Startup, Setup and Graph load times for the cluster

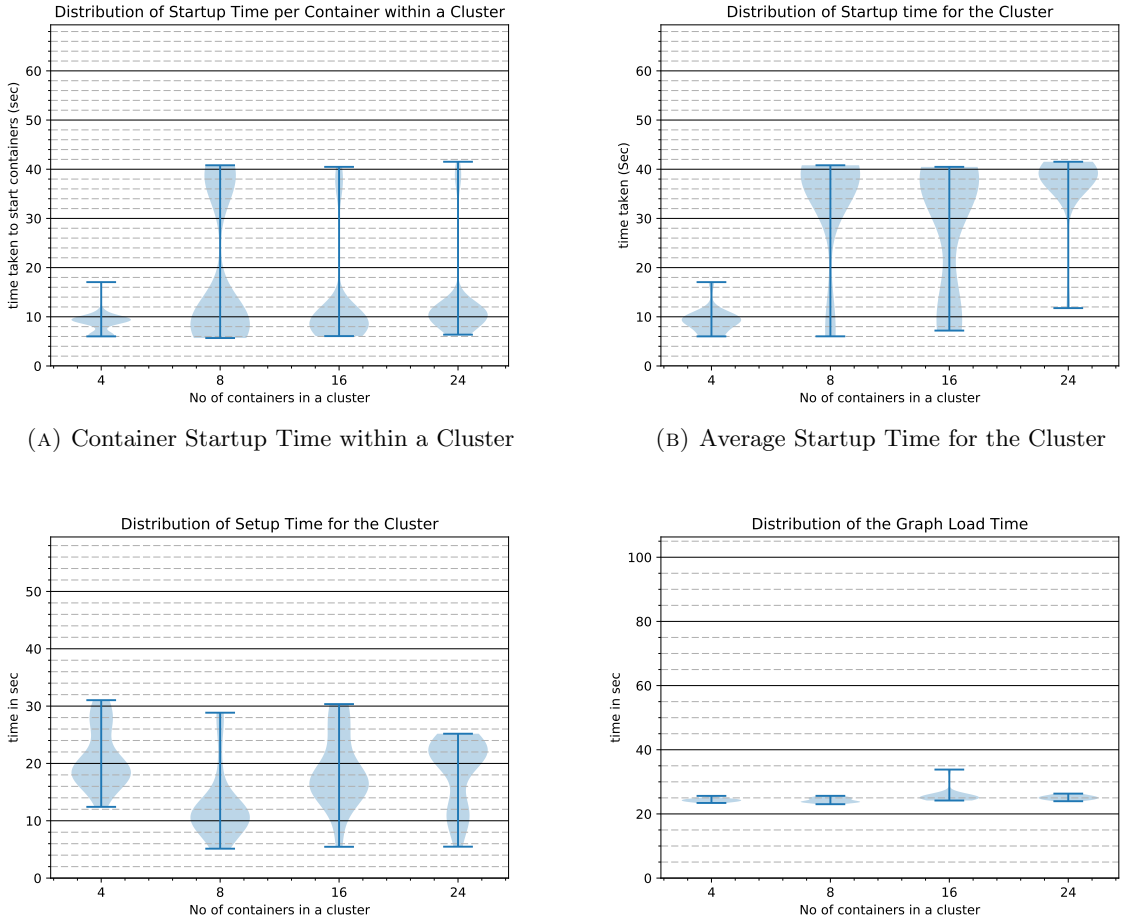
3.2 Validation

Considering all the setup (i.e., Docker and Swarm cluster is set up along with all images present on all the nodes) is complete, validation experiments monitor the following three processes in running a GoFFish Cluster:

- **Starting all the containers which are a part of the cluster.**
  (Time taken from State-D to State-E (in Fig 3.6))
- **After starting all containers, waiting till the time both Hadoop and Hama clusters are set up.**
  (Time taken from State-E to State-F (in Fig 3.6))
- **Loading our sample graphs on the cluster.**
  (Time between State-F to State-G (in Fig 3.6))

We run numerous iterations in order to log these three major timings for our setup and plot our results in Figure 3.7 and 3.8.

Fig:3.7 represents the time taken in order to complete startup of containers, time taken to set up of the clusters(once all containers have started) and the time taken to load graphs on HDFS



(A) Container Startup Time within a Cluster

(B) Average Startup Time for the Cluster

(C) Average Cluster Setup Time

(D) Average Graph Loading Time

FIGURE 3.8: Violin plots showing distribution of Startup, Setup and Graph load times

on the cluster and the "Red lines" represent the time at which each container inside the cluster starts.

Fig:3.8 not only represents the startup time for the cluster, but also all the containers within a cluster separately as violin plot along with the same for both cluster setup and graph load timings.

From Fig:3.7, we observe that it typically takes around 10 seconds to run most of the containers inside the cluster, but occasionally we see some containers taking 30-40 seconds in order to start up successfully. Which can also be explained from Fig:3.8(A) and Fig:3.8(B).

However, both Cluster setup and graph loading time stabilise as we go on increasing the number of containers in our cluster to about 18 seconds (in the case of cluster setup) and roughly 25 seconds (as far as the graph loading time is concerned). This is validated from the violin plots from Fig:3.8(C) and Fig:3.8(D)

Chapter 4

Conclusion and Future Work

4.1 Conclusion

Manually setting up and configuring a Big Data platform cluster is complex and time consuming. In order to manually setup a cluster of servers with all the prerequisites and dependencies, it takes on an average 2-3 hours for an individual before they can seamlessly run a GoFFish application. Also, such a setup would also make it very difficult to scale in and out as per demand due to the physical work involved to add and remove nodes from the cluster. If we consider Cloud VMs as an alternative, we observe that it takes one minute or more for a VM just to start, with the additional overheads for further configurations.

In this thesis, we have eased this deployment and management overheads using Docker. From our experiments, we observe that we bring the combined time to start each container in a cluster, setting up all services in our cluster and loading our sample graphs (potentially making it ready to run the GoFFish application) down from the order of hours to the order of seconds. We also provide functionalities of scaling up on the go without restarting the cluster and providing better management of our nodes inside the cluster. Thus it proves to be an all round solution for our distributed setup problem, and reduces the time for research, development and usage of these graph platforms.

4.2 Future Work

There are several additional ides to pursue further. One interesting problem is on leveraging container migration for graph processing frameworks at the infrastructure layer. Linux Containers (lxc and lxd) enable us to run multiple isolated and light-weight Linux systems on the same host kernel. The host itself can be present in a VM on public Clouds while using them. LXC and LXD support stateful snapshot and live migration using Checkpointing and Restore (CRIU). One can envision lxc containers hosting one GoFFish worker each with one or more partitions assigned to them. Multiple containers, hence workers, can be instantiated on a single host. Then, when rebalancing and elasticity are required, we can use live-migration to move the container with the workers and its partitions from one host to another. This can even happen during an application execution. While we are currently planning such a container mechanism to monitor and orchestrate GoFFish v3 on a cluster, in future, this can play a role in elastic placement policies and offer flexibility in the level of abstraction for platform and cluster management.

Moreover, the role of containers is not confined to graph processing, and can be envisioned in the Edge+Cloud domain. In field of Internet of Things (IoT), we have many edge devices like Raspberry-Pi, Smart Wearable, Sensors, etc. with constrained but non-trivial compute and memory capacity. This allows applications to partially run on these edge devices and partly on the Cloud. For the edge, we can leverage the host kernel's sharing property of containers which allows them to have a minimal memory overhead on the host machine and still offer a sandbox to execute the application. It would be infeasible to use a hypervisor-based virtual machines which consume a major chunk of memory and compute capacity on the edge. Thus containers can play a pivotal role in areas where we require quick deployment across thousands of edge devices, fast deployment times and low resource overheads in order to run diverse IoT applications.

Bibliography

[1] *Docker internals*. https://www.kobo.com/in/en/ebook/working-with-docker/.

[2] *Integrating Docker Compose with Swarm*. https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm/. 2015.

[3] *Layering of images in a Docker Container*. http://stackoverflow.com/questions/30016521/working-on-a-dockerfile-in-order-to-build-a-wordpress-image.

[4] *LXC vs Docker*. https://www.flockport.com/lxc-vs-docker/. 2014.

[5] *LXD: A container orchestration system*. http://www.zdnet.com/article/ubuntu-15-04-container-friendly-linux-for-cloud-and-servers-arrives-soon/.

[6] *LXD: Working and Benifits*. http://linux.softpedia.com/blog/infographic-lxd-machine-containers-from-ubuntu-linux-492602.shtml. 2016.

[7] Kamran Siddique et al. "Apache Hama: An Emerging Bulk Synchronous Parallel Computing Framework for Big Data Applications". In: 2016.

[8] Yogesh Simmhan et al. "GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics". In: *International European Conference on Parallel Processing (EuroPar)*. 2014.

[9] *Virtual Machines vs Containers*. http://searchservervirtualization.techtarget.com/answer/Containers-vs-VMs-Whats-the-difference. 2015.

[10] *Working of Docker Container*. https://webapplog.com/node-docker/.