Tutorial on Deploying Computer Science and Engineering Test-bed on Multiple Cloud Using Docker

Chameleon Cloud Tutorial

National Science Foundation Program Solicitation # NSF 13-602 CISE Research Infrastructure: Mid-Scale Infrastructure - NSFCloud (CRI: NSFCloud)

Contents

Introduction

This document serves as a collection of an ongoing series of tutorials authored for Docker and services surrounding Docker on the Chameleon Cloud platform. Spanning topics from basic installation to creating customized Docker applications that run across multiple hosts, these sections are designed to serve as a starting point from which users can create their own container-based applications.

This document also covers Kubernetes, a solution that uses Docker containers across a wide variety of hosts to offer a resilient and reliable solution for applications. We demonstrate both Kubernetes and Docker's native solution so users can make a knowledgeable choice about which solution is best. We start with an in-depth look at some of the technologies underlying Docker that enable additional features in management and security.

1. Dockers

1.1. Docker Hub

The Docker Hub is a cloud-based registry service for building and shipping application or service containers. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

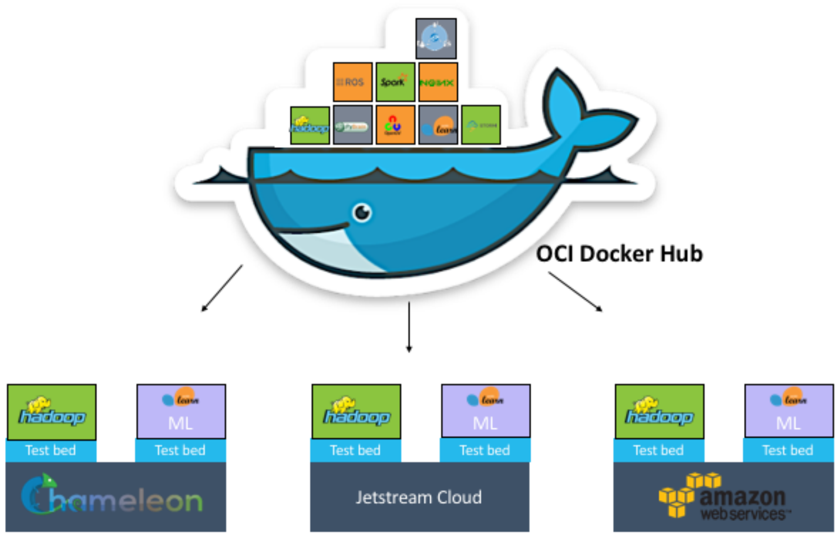

Figure 1: Science and Engineering Multi Cloud Test bed

1.2. Containers

Containers include the application and all of its dependencies, but share the kernel with other containers. They run as an isolated process in userspace on the host operating system. They're also not tied to any specific infrastructure – Docker containers run on any computer, on any infrastructure and in any cloud [1].

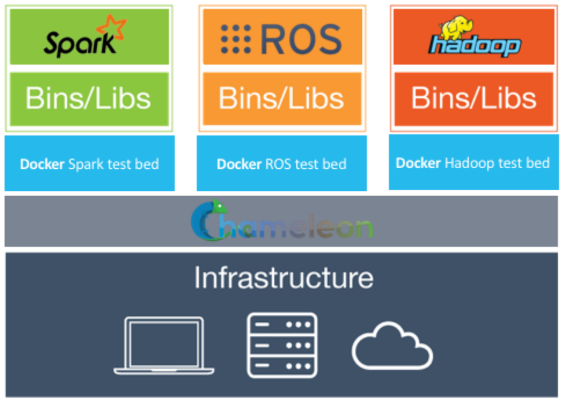

Figure 2: Containers *[1]*

1.3. Linux Architecture Features and Concepts

Being one of the leaders in the container-based world, Docker often takes advantage of several features belonging to the Linux kernel as a means to better its service. In particular, Docker's use of namespaces and control groups (cgroups) and how each play a role in resource management and security cannot be overlooked. In order to understand what Docker provides through these features, one must first understand what they accomplish individually.

1.3.1. Namespaces

To start off with, namespaces are a very convenient feature of Linux that serves to provide a somewhat virtualized resource that acts as a go between for processes within the group and the actual resources. By sectioning off these processes into their own groups, they are unable to see the resources being used by separate groups while still remaining visible to those processes within the same group. Using this allows users to segment things such as: Networking, PIDs, Users, Mounts, and more. Essentially, the process will have no knowledge of processes outside of its group and would have no need to know about them, providing an isolated bundle of processes.

When applying this knowledge towards the implementation with Docker, the advantages then become clear in the form of containers and linked-containers being permitted to run on the same host, but oblivious about any other running containers. Docker, by default, will separate a newly created container into its own namespace to distance containers from one another. This is done across most of the namespaces listed before and allows the container to act as a solo application on the host.

Like many other Linux features, namespaces establish themselves using a virtual filesystem in the fashion of /proc/$PID/ (where $PID is the pid of the process). Within this directory, Linux already displays intimate information about processes. With namespaces, additional information is available primarily through the ns directory (namespaces) which holds file handles pointing to the corresponding namespace. Additionally, within other directories such as mounts and net, even more information about the current namespaces are available. Given that one of the core goals of a container was to replace the need for virtual machines, namespaces provide an opportunity to bridge the gap through process isolation. This gives the appearance of each as having their own virtual machine.

This usage of namespaces plays hand-in-hand with cgroups and their many uses.

1.3.2. CGroups

When speaking about cgroups, the first idea that should come to mind is of resource management. Because where cgroups excels is in an area similar to namespaces, by separating and grouping processes into whatever organizations the user prefers and inflicting global (group-wide) rules upon them. Cgroups provide a kernel-level tool for resource management and accounting that is greatly relied upon by Docker and many other modern tools. Beginning with the organization, processes are organized into cgroups based first and foremost off of inheritance. That is, child processes will inherit their initial group based off the parent group they belong to.

NOTE: Only certain properties are inherited from parent cgroups that can also be modified upon child process creation. Cgroups are used to divide up the resources of a system into separate catagories that can be described in terms of limits and shares among the host resources. The first and most prevalent example is the sharing of the CPU amongst different processes. Initially, one might think to allocate a certain number of cores to system processes and leave the rest for userspace processes (assuming the system contains enough cores). From there, the user processes may spawn a webserver which would be placed in its own cgroup along with its children so that they only use a certain percentage of the CPU time, RAM, I/O, Network, etc. These limitations and rules are used to define the resource behavior of programs within their cgroups so that each has their "fair share".

Much like with CPU scheduling, cgroups are often used to evenly distribute resources amongst its processes, but the difference is that it does not need to be equal. For example, when running unfamiliar applications, one thought might be to run all user-level programs in their own cgroup so as to limit their usage of the system resources to a certain extent such that they won't be able to monopolize and steal away these resources from the system. Whether the exhaustion of a resource is intentional or accidental, cgroups provide a safety barrier (similar to that of saving 5% of the hard disk for root) that can be used for the safety of the host as a whole.

In terms of how Docker uses cgroups is that Docker will place all the containers, unless told otherwise, into their own cgroup that are each set to equally share the system resources. However, by passing in certain arguments, one can raise the priority of certain containers, lower the I/O permitted by others, and all around customize to the best efforts of the system. Additionally, since cgroups occur at the system level, none of the applications will ever need to be aware that they are being throttled or administrated over in any way, again providing the illusion of each container belonging on its own host.

Similar to the namespaces, cgroups manifest themselves in the virtual filesystem. Depending on your installation, they are often found at /sys/fs/cgroup/ or /cgroup/. From this root directory, you can see that each subsystem has its own directory within which you can create cgroups within. Aside from that, you are also capable of mounting a cgroup and assigning the subsystems desired to be nested within using the mount command. Cgroups subscribe to a hierarchical structure in order to create nested groups of related tasks such that related processes can be stored together. This can be extended to running all tasks related to the webserver within one cgroup, internally, creating different cgroups for the database, backend, etc.

1.4. Docker Fundamentals

Docker is conceptually similar to virtual machines but has much less resource overheard because it doesn't run a full guest OS. Docker containers start in seconds vs minutes, take up less space, and are less hardware demanding because they share resources with the host OS[1].

1.4.1. Terms and Concepts

Most of the docker descriptions are taken directly from their glossary.[2]

Docker Engine or "Docker": The docker daemon process running on the host which manages images and containers

Image: Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have a state and it never changes.

Images fulfill different needs and workflows. System images are useful to build off or provide an environment to work in. Processing-type images take input files and produce processed output files. One-off images run some predetermined task and generate a result or success message. We'll look at specific examples below.

Docker Hub: Docker provides the Docker Hub service to host and build images. Users create Docker images and push them to Hub for others to use and expand on. Hub also allows for automated builds. Automated builds link to a repo (GitHub or Bitbucket) and build an image on Hub servers using files from the repo automatically as you push.

Container: A container is a runtime instance of a docker image. A Docker container consists of:

- A Docker image
- Execution environment
- A standard set of instructions
- The concept is borrowed from Shipping Containers, which define a standard to ship goods globally. Docker defines a standard to ship software.

Dockerfile: A Dockerfile is a text document that contains all the commands you would normally execute manually in order to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.

1.5. Hands on Exercise

1.5.1. Outline

This is a hands on exercise for using Docker on Cloud. Reader can use instances from any cloud provider. For the demonstration purpose NSF funded Chameleon Cloud resources are used. Please visit [3] for details. At the end of this exercise reader will have setup a demo website utilizing 5 Docker containers and 2 hosts. See the official Docker docs for more details[4] .

The following tools are used in this exercise:

Postgres: An SQL database. [5]

Nginx "engine x": A web server. [6]

uWGSI: An application server that connects to Nginx. A simple Python app that generates the demo page is used in this demostration. [7]

Reader will be setting up one host with a Postgres (SQL database) container. The other host will be setup with Nginx (web server) and uWGSI containers. uWGSI is an interface to Python script that generates actual page. To connect the uWGSI container across hosts to the Postgres container we will use *ambassador* containers, one on each host. Several online resources have been used for generating this walk through exercise, reference links [8-11] are available in references section for further reading.

1.5.2. Procedure

Please follow the following steps.

1.5.2.1. Spinning up cloud resources

Create 2 Chameleon baremetal servers. We used a CentOS 7 image for the demonstration but feel free to use any other distro as long it runs Docker.

1.5.2.2. Installing software

Install Docker on each server using yum.

```
$ sudo yum install docker
```

This installs the Docker daemon and client tools. The Docker daemon needs to be running before you can use Docker. If you're getting errors with every Docker command this may be the cause. Start it with

```
$ sudo service docker start
```

1.5.2.3. Container setup

There are two ways to deploy the containers. One can pull already built containers from our cloudandbigdatalab Docker Hub [12] and run them. Or pull this GitHub repo [13] and build the Docker images using the Dockerfile in each directory. To edit the site content one will need to build the images after making edits, although database can be edited by simply connecting to it. The ambassador containers are pulled from the Docker hub as they are maintained by Docker. Some useful commands can be found in Appendix 1. We strongly recommend going through Appendix 1 before moving on to next step.

As mentioned in the previous paragraph this exercise can be performed in two different ways,

a) Pull the image from Docker hub
b) Build the image using files from Github

1.5.2.3(a) Pulling the image from Docker hub approach

Setting up Host 1

Follow the steps shown below to start the containers on host 1.

```
$ sudo docker run --name postgres -d cloudandbigdatalab/postgres
```

This command pulls the image form the Docker hub if it not available locally and then starts the container. In particular the above command pulls the postgres repository of cloudandbigdatalab user from Docker hub and starts the container. In the command, option '-d' runs the container as a daemon, that is, it runs in the background.

```
$ sudo docker run --name host2_ambassador -d \

  --link postgres:postgres \

  -p 5432:5432 \

  svendowideit/ambassador
```

This command pulls the ambassador image from Docker hub and then starts the container with 'host2_ambassador' name. Then it links the container to postgres and maps port 5432 from within container to outside.

Setting up Host 2

Follow the steps shown below to start the containers on host 2. Add the IP address of the host 1 in specified commands. If you need to restart the uWGSI container you will need to remove both uWGSI and Nginx containers then run again. The uWGSI container must be run first.

```
$ sudo docker run --name host1_ambassador -d \

 --expose 5432 \

 -e POSTGRES_PORT_5432_TCP=tcp://host1_ip:5432 \

 svendowideit/ambassador
```

This command starts ambassador container, exposes the port 5432 (postgres default) to linking containers and sets environment variable for postgres. Do not forget to add the IP address of the first instance (host-1) to the command.

```
$ sudo docker run --name uwsgi -d \

 --link host1_ambassador:postgres \

 cloudandbigdatalab/uwsgi
```

This command starts uWGSI container and links it to host1_ambassador container.

```
$ sudo docker run --name nginx -d \

--link uwsgi:uwsgi \

-p 80:80 cloudandbigdatalab/nginx
```

This command starts Nginx container, links it to uWSGI container and maps http default port, port 80 to outside.

1.5.2.3(b) Building the image approach

Setting up Host 1

To build the Docker image clone the from github repo.

```
git clone https://github.com/cloudandbigdatalab/tutorial-cham-docker-1.git
```

Once the repository is cloned. Change to postgres directory and build the image

```
     $ cd postgres

     $ sudo docker build -t postgres .
```

After the above commands run the same commands used for setting up host 1 in method (a) and don't forget to replace cloudandbigdatalab/image_name with image_name in the commands.

Setting up Host 2

To build the Docker image clone the from github repo.

```
git clone https://github.com/cloudandbigdatalab/tutorial-cham-docker-1.git
```

Once the repository is cloned. Change to `uwsgi` directory and build the image

```
     $ cd postgres

     $ sudo docker build -t postgres .
```

Change to `nginx` directory and build the image

```
     $ cd ../nginx

     $ sudo docker build -t nginx .
```

After the above commands run the same commands used for setting up host 2 in method (a) and don't forget to replace cloudandbigdatalab/image_name with image_name in the commands.

1.5.2.4. Testing the website

Once the setup is complete and the test webpage should be available at " `http://host2_IP/` ".

Note: While using Chameleon cloud the ports 80 and 5432 should be unblocked manually by adding the security group rules.

2. Machine, Compose, and Swarm

2.1 Machine [14]

Machine allows user to create Docker hosts and control them without interacting with the host machines directly. This way the user don't have to SSH to machines running the Docker daemon to run containers. In detail, Machine enables creation of Docker hosts on a computer, on cloud providers, or inside a data center. It automatically creates hosts, installs Docker on them, then configures the Docker client to talk to them. A “machine” is the combination of a Docker host and a configured client.

Machine supplies a number of commands for managing the Docker hosts. Using these commands one can start, inspect, stop, and restart a host, upgrade the Docker client and daemon, and configure a Docker client to talk to a host.

2.2 Compose [15]

Compose is a tool for defining and running multi-container applications with Docker. With Compose, user can define a multi-container application in a single file, then spin an application up in a single command which does everything that needs to be done to get it running. Compose is great for development environments, staging servers, and CI. It is not recommended for production use yet. Using Compose is basically a three-step process.

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

Compose has commands for managing the whole lifecycle of your application, the commands can provide the following services: start, stop and rebuild services, view the status of running services, stream the log output of running services and run a one-off command on a service

2.3. Swarm [16, 17]

Docker Swarm is native clustering for Docker. It allows users create and access to a pool of Docker hosts using the full suite of Docker tools. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts. And of course, the Docker client itself is also supported.

Like other Docker projects, Docker Swarm follows the “swap, plug, and play” principle. As initial development settles, an API will develop to enable pluggable backends. This means you can swap out the scheduling backend Docker Swarm uses out-of-the-box with a backend you prefer. Swarm’s swappable design provides a smooth out-of-box experience for most use cases, and allows large-scale production deployments to swap for more powerful backends, like Meso.

2.4. Hands on Exercise

This exercise will cover usage of Docker Machine, Compose and Swarm. Ultimately these tools are intended to be used together but because they're not yet mature that synthesis is limited. We'll discuss the limitations in more detail throughout the tutorial. We'll instead focus on using each tool individually and demonstrate them together in ways that currently work.

It is recommended to use CentOS. For demonstration we have used Chameleon Cloud [3] CentOS bare metal instance. This can also be implemented in other environments but the set of commands we share here might not work.

2.4.1. Initial Setup

Run these set of commands to install Docker using yum installer, create a docker group and add the current user ‘cc’ to it (this helps in not using sudo for docker commands). Logout and then login to the account and then start the Docker daemon.

```
$ sudo yum update -y

$ sudo yum install -y docker

$ sudo groupadd docker

$ sudo usermod -a -G docker cc

$ sudo systemctl start docker.service
```

Once the Docker is setup, machine and compose are to be installed. As both compose and machine are not yet in production stage it is advisable to follow the installation steps from the following websites. From our experience installing machine from the binaries is the best approach. Here are the websites for installation of machine and compose.

Docker-machine installation instructions: https://docs.docker.com/machine/install-machine/

Docker-compose installation instructions: https://docs.docker.com/compose/install/

2.4.2. Compose Hands on

In this part of the exercise we demonstrate the usage of docker-compose. Once the github repository is cloned, docker-compose is used to spawn the container automatically, as specified in the YAML document.

Use the following command to clone the git repository.

```
$ git clone https://github.com/cloudandbigdatalab/tutorial-cham-docker-2.git
```

Change your director to the tutorial-cham-docker-2 and run the composition.

```
$  cd tutorial-cham-docker-2

$ docker-compose -p tutorial up -d
```

To update the images and restart the containers use the following commands.

```
$ docker-compose pull

$ docker-compose -p tutorial up -d
```

Here is the command for checking the running containers.

```
$ docker-compose -p tutorial ps
```

The above command should list all the containers as shown below.

```
Name                Command                              State          Ports
------------------------------------------------------------------------------------------------------------------------
tutorial_db_1       /docker-entrypoint.sh postgres       Up             5432/tcp
tutorial_page_1     ./startup.sh                         Up             3031/tcp
tutorial_server_1   nginx -g daemon off;                 Up             443/tcp, 0.0.0.0:80->80/tcp
```

Now visit the ip of Chameleon machine in the browser to see the page running.

2.4.3 Machine Hands on

Machine enables the creation of Docker hosts on cloud providers. In this part of the exercise we demonstrate creation of remote hosts on cloud provider using Machine. For demonstration purpose we used Chameleon cloud and Rackspace cloud. There are some issues with the Chameleon cloud API so it is advisable to use some cloud provider like Rackspace or Amazon for creation of virtual machines with Machine. In this exercise we used Chameleon cloud instance for spawning the instances on Rackspace cloud.

To create a host using docker-machine on Rackspace cloud, environment variables for OS_USERNAME, OS_API_KEY and OS_REGION_NAME. The following link provides some details about setting up the environment variables for Rackspace cloud.

Once the environment variables are setup, the docker-machine can be used to spawn the remote hosts. In this command "- d rackspace " specifies the driver as Rackspace.

```
$ docker-machine create -d rackspace docker-main
```

This command usually takes several minutes to setup the remote hosts. Now point the Docker to the remote host using the following command.

```
$ eval "$(docker-machine env docker-main)"
```

Once the above command is used the containers running in the local host are destroyed. This can be observed by using the following command.

```
$ docker ps
```

Its time run the composition on the remote host. These commands are similar to docker-compose exercise that is, run the composition.

```
$  cd tutorial-cham-docker-2

$ docker-compose -p tutorial up -d
```

Let’s check if all the containers are up and running

```
$ docker-compose -p tutorial ps
```

The output should be similar to the one from docker-compose exercise.

To check the IP of the newly spawned remote host use the following command.

```
$ docker-machine ip docker-main
```

Now visit the IP of Chameleon machine in the browser to see the page running.

2.4.4 Swarm Hands on

Chameleon cloud along with Rackspace cloud has been used for demonstration of this exercise and Machine is used to setup the Swarm cluster. We managed the cluster on Rackspace cloud from Chameleon cloud bare metal instance. A swarm cluster can also be setup manually.

For this demo, multi-container setup used earlier cannot be used. This is for two reasons:

1. Currently in Docker, linked containers must be running on the same host. This defeats the point of Swarm. Docker's networking is being overhauled to allow cross-host links and the feature is available in experimental builds. We were unable to get it working at the time of this writing however.

2. Even with cross-host linking, there's no automatic proxying or load balancing. So if for example we scaled the page container to 10, that's easy enough. But we'd also have to configure Nginx to load balance between those containers. Or we could have a proxy container in between the two. This is all possible but again we didn't get it working at the time of this writing. This is something you must build into your app design, there's no automatic mechanisms for this as of yet.

We're still using an (extremely sparse) docker-compose.yml for this. It consists of one service / container that runs folding@home [18]. We're going to run it and scale it across a few nodes.

Generate a swarm token and set it to an environment variable SWARM_TOKEN. Token helps in identifying the swarm.

```
$ export SWARM_TOKEN=$(docker run swarm create)
```

Creating a swarm master instance. The appropriate environment variables are to be set with the cloud provider and user account information, similar to previous Machine hands on exercise. This should create an instance for swarm master.

```
$ docker-machine create -d rackspace --swarm --swarm-master  \
 --swarm-  discovery=token://$SWARM_TOKEN docker-swarm-master
```

Creating swarm worker nodes. The following command should create 2 worker nodes.

```
$   for ((i = 0; i < 2; i++)); do    \
docker-machine create -d rackspace –swarm     \
 --swarm-discovery=token://$SWARM_TOKEN docker-swarm-node-$i;   \
 done
```

Pointing the docker client at the Swarm cluster.

```
$   eval "$(docker-machine env --swarm docker-swarm-master)
```

To check the information on created nodes use the following command.

```
$  docker info
```

Then the result should look something similar to this.

```
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 docker-swarm-master: xxx.xxx.xxx.xxx:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.014 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-37-generic,
operatingsystem=Ubuntu 14.04.1 LTS, provider=rackspace, storagedriver=aufs
 docker-swarm-node-0: xxx.xxx.xxx.xxx:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.014 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-37-generic,
 operatingsystem=Ubuntu 14.04.1 LTS, provider=rackspace, storagedriver=aufs
 docker-swarm-node-1: xxx.xxx.xxx.xxx:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.014 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-37-generic,
 operatingsystem=Ubuntu 14.04.1 LTS, provider=rackspace, storagedriver=aufs
CPUs: 3
Total Memory: 3.041 GiB
Name: 73e232b31975
```

For this exercise use the docker-compose.yaml from the swarm directory.

```
$  cd tutorial-cham-docker-2/swarm/

$ docker-compose -p tutorial up -d
```

Get the running container information using the following command.

```
$ docker-compose -p tutorial ps

     Name                        Command                    State     Ports
--------------------------------------------------------------------------------------
tutorial_worker_1     /bin/sh -c /etc/init.d/FAH ...     Up
```

It can be observed that only one worker container is running. To scale the number of running containers to 6 use the following command.

```
$  docker-compose -p tutorial scale worker=6
```

```
$ docker-compose -p tutorial ps

     Name                    Command                         State     Ports
----------------------------------------------------------------------------------------
tutorial_worker_1      /bin/sh -c /etc/init.d/FAH ...     Up
tutorial_worker_2      /bin/sh -c /etc/init.d/FAH ...     Up
tutorial_worker_3      /bin/sh -c /etc/init.d/FAH ...     Up
tutorial_worker_4      /bin/sh -c /etc/init.d/FAH ...     Up
tutorial_worker_5      /bin/sh -c /etc/init.d/FAH ...     Up
tutorial_worker_6      /bin/sh -c /etc/init.d/FAH ...     Up
```

To observe at the `NAMES` field and see the containers spread across the 3 hosts in the cluster use docker ps command.

```
        $ docker ps

        CONTAINER ID      IMAGE                 COMMAND            CREATED          STATUS
PORTS            NAMES
        a093e3b7b2b6      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   33 seconds ago      Up 33
seconds                   docker-swarm-master/tutorial_worker_3
        31ca2233bf7f      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   33 seconds ago      Up 33
seconds                   docker-swarm-node-1/tutorial_worker_5
        14268f5c3238      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   33 seconds ago      Up 33
seconds                   docker-swarm-node-0/tutorial_worker_6
        578b13ff2462      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   34 seconds ago      Up 33
seconds                   docker-swarm-master/tutorial_worker_4
        9987e2ab5a8b      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   34 seconds ago      Up 33
seconds                   docker-swarm-node-0/tutorial_worker_2
        2d5e5d042dba      jordan0day/folding-at-home   "/bin/sh -c '/etc/in   About a minute ago   Up
About a minute                 docker-swarm-node-1/tutorial_worker_1
```

2.4.5. Conclusion

Docker intends for Compose, Machine, and Swarm to work together to enable simple yet powerful workflows. The experience of putting this tutorial together shows that's not reality today. However, Compose and Machine work pretty well on their own barring Machine's Chameleon incompatibility. The synthesis between Compose and Machine is also solid right now. Swarm is problematic, as it is not in production stage yet this is expected. But Docker does disclaim that these tools are not production ready yet. In the future they should work better for multi-container apps and services.

3. Kubernetes

3.1. Kubernetes Fundamentals

In Greek, the word Kubernetes means “helmsman of a ship” and this is exactly what Googles Kubernetes does. While Docker provides the containers and its environment in a cluster, Kubernetes orchestrates and manages them. The helmsman of Docker containers. Kubernetes figures out which host to put the things on and schedules them. Kubernetes manage the Docker containers with the help of many different subsystems. In this section we explain about some of them.

Pod

Pod is a logical collection of multiple Docker containers. Often, each pods have only one container in them. Only closely related containers are collated inside a pod. It can be seen as a collocated group of Docker containers that share an IP and storage volumes. Thus pods helps to represent one or multiple containers as a single application, providing easy deployment and scaling. The basic structure of pods helps it to be reused across environments. Pods are defined by a JSON or YAML file called a pod manifest. It is repeatable and manageable. Pods generally have a main container and some helper containers to facilitate related task. Pods are created and destroyed as and when required.

As described in Kubernetes documentation[19] , Pods can be used to host vertically integrated application stacks, but their primary motivation is to support co-located, co-managed helper programs, such as:

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

Individual pods are not intended to run multiple instances of the same application, in general.

Service

Service provides a single, stable name for set of pods and acts as a basic load balancer. Service’s are advertised through environment variables and is automatically configured and runs inside the clusters. Service function as an interface between a swarm of containers and the user which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service[20]. This way, the user does not have to worry about the backends as the frontend clients gets the freedom to communicate to one single point alone. That is, Service abstraction provides decoupling between the frontend and backend. For example, when you connect to a load balancer, it sends the traffic to a healthy pod to handle and if one of the pods goes down, it will automatically get dropped out of the load balancing providing get real-time load balancing automatically for all the components of the system. The pods in the backend may change, scale or be swapped but frontend will not be aware of any of this happening inside. Thus it gives a stable address and shields clients from implementation details. You will have a Service in front of each of the Replication Controllers and they find each other through that service.

For Kubernetes-native applications, Kubernetes offers a simple *Endpoints API* that is updated whenever the set of Pods in a *Service* changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Service's which redirects to the backend Pods[20].

Replication Controller

As the name implies, a Replication Controller manages the lifecycle of pods and ensures that a specified number of pod "replicas" are running at any one time. If the user intends to run 5 pods at a time, the Replication Controller takes care of it by replacing pods that are deleted or terminated for any reason, such as in the case of node failure or disruptive node maintenance, such as a kernel upgrade, ensuring that 5 pods will be running, always. Similarly if another comes online for some reason, the Replication Controller will kill it off. Thus it can be thought of as a declarative stage, where you specify the number of replica's that are ultimately required to run in your system. This gives users direct control of Pods and provide a fine-grained control for scaling, supervising multiple pods across multiple nodes[21].

Labels

Kubernetes uses Labels to organize and group components. Labels are key/value pairs which are attached to objects such as pods. Specific and unique details like object name, role, frontend, environment, production etc. are mentioned in a Label. This helps to identify attributes of objects that are meaningful and relevant to users, but which do not directly imply semantics to the core system. Labels support the working of Service and Replication Controller by talking to API in groups. Replication Controllers use the same label to all containers spawned from a particular template also allowing management of all instances as a group disregarding the number of containers. Each object can have multiple labels but a unique key. So grouping of pods can be done in multiple fashion as required, assigning many labels, which will better the fine-grained control. Objects can then be selected based on a single or combined label requirement[22, 23].

Annotations

Annotations are, like labels, key-value maps. They store user provided arbitrary non-identifying metadata in them. API clients rely on these Annotations to get more detailed information, which are not specified in labels. Annotations store large amount of information, which can be stored in a structured or unstructured manner inside them.

Names [24]

Names are generally client-provided. Only one object of a given kind can have a given name at a time (i.e., they are spatially unique). But if you delete an object, you can make a new object with the same name. Names are the used to refer to an object in a resource URL, such as /api/v1/pods/some-name. By convention, the names of Kubernetes resources should be up to maximum length of 253 characters and consist of lower case alphanumeric characters, -, and ., but certain resources have more specific restrictions.

Namespaces

As and when the size of the cluster increases, or when multiple users working together in different projects need to use a common cluster, we use Namespaces to provide a scope for names across projects. It is used to divide the cluster resources between multiple users working under different projects or groups. The names inside a Namespace is to be unique and it can be different for different namespaces. This creates independent names for each group in a cluster, reducing the complexities that can arise.

Namespaces are a great way to separate resources via resource quota but the use of namespaces to differentiate between slightly different resources may be avoided wherever possible. The informed used of Labels is advised by Kubernetes team, to distinguish resources within the same namespace, instead of creating namespaces inside an already existing one[25].

3.2. Hands on Exercise

The fundamental idea behind Kubernetes is that, as a user, there is no need to know about which host an application is running on. This allows the user to focus only on the application and not have to worry about the specifics of the host or management.

The setup of Kubernetes relies on a single host serving as a Master. Master will serve as the primary controller that will manage the Kubernetes installation. Any additional hosts that are to be used for application deployment are defined as a Node. For the exercise, the design will be to assign one instance as the Master and the second as the Node.[26-28]

For this walk through exercise demonstration we used NSF Chameleon Cloud. Services from any other node can also be used. We assume that the user has basic knowledge of Linux and has proper understanding about terms public IP and private IP.

3.2.1. Kubernetes setup

The steps in this section help in setting up a 2 node Kubernetes cluster. Throughout the tutorial <master_ipaddress> and <worker_ipaddress> will be seen, these are supposed to be replaced by the private IP of master and worker instances.

Install Kubernetes in both the servers.

```
$ sudo yum install kubernetes etcd -y
```

Setup instructions for Master node:

Open **/etc/kubernetes/apiserve**r using your favorite editor.

```
$ sudo vi /etc/kubernetes/apiserver
```

Make sure that the following lines are uncommented and edit them to look as shown below.

```
        KUBE_API_ADDRESS="--address=0.0.0.0"

        KUBE_API_PORT="--port=8080"

        KUBELET_PORT="--kubelet_port=10250"

        KUBE_ETCD_SERVERS="--etcd_servers=http://127.0.0.1:2379"

        KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

        KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,ServiceAccount,ResourceQuota"
```

Then add the following line at the end of the file and then save it. Make sure that <master_ipaddress> is replaced by the private IP of the master node.

```
        KUBE_MASTER="--master=http:// <master_ipaddress>:8080"
```

Now open **/etc/kubernetes/config** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit them to look as shown below.

```
        KUBE_LOGTOSTDERR="--logtostderr=true"

        KUBE_LOG_LEVEL="--v=0"

        KUBE_ALLOW_PRIV="--allow_privileged=false"

        KUBE_MASTER="--master=http:// <master_ipaddress>:8080"
```

Now open **/etc/kubernetes/controller-manager** using your favorite editor similar to the first step of this section, add the following line at the end of the file and replace <worker_ipaddress> with the private IP of second instance.

```
        KUBLET_ADDRESSES="--machines=<worker_ipaddress>"
```

Now open **/etc/kubernetes/kubelet** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit them to look as shown below.

```
KUBELET_ADDRESS="--address=127.0.0.1"

KUBELET_HOSTNAME="--hostname_override=127.0.0.1"

KUBELET_API_SERVER="--api_servers=http://127.0.0.1:8080"
```

Now open **/etc/kubernetes/proxy** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit them to look as shown below.

```
KUBE_PROXY_ARGS="--master=http:// <master_ipaddress>:8080"
```

Setup instructions for Worker node:

Open **/etc/kubernetes/apiserve**r using your favorite editor.

```
$ sudo vi /etc/kubernetes/apiserver
```

Make sure that the following lines are uncommented and edit them to look as shown below.

```
KUBE_API_ADDRESS="--address=127.0.0.1

KUBE_ETCD_SERVERS="--etcd_servers=http:// <master_ipaddress>:4001"

KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,ServiceAcco
unt,ResourceQuota"
```

Now open **/etc/kubernetes/config** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit or add lines to look as shown below.

```
        KUBE_MASTER="--master=http:// <master_ipaddress>:8080"

        KUBE_ETCD_SERVERS="--etcd_servers=http:// <master_ipaddress>:4001"
```

Now open **/etc/kubernetes/kubelet** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit or add lines to look as shown below.

```
        KUBELET_ADDRESS="--address=0.0.0.0"

        KUBELET_PORT="--port=10250"

        KUBELET_HOSTNAME="--hostname_override=<node_ipaddress>"

        KUBELET_API_SERVER="--api_servers=http://<master_ipaddress>:8080"
```

Now open **/etc/kubernetes/proxy** using your favorite editor similar to the first step of this section, make sure that the following lines are uncommented and edit or add lines to look as shown below.

```
        KUBE_PROXY_ARGS="--master=http://<master_ipaddress>:8080"
```

On the Master (the first host), restart the service in order for the configuration changes to take effect.

```
        $ sudo systemctl restart kube-apiserver
```

Additionally, enable each service so that is will start at boot for the server.

```
        $ for cmd in restart enable status; do sudo systemctl $cmd etcd kube-apiserver kube-scheduler

kube-controller-manager; done
```

The Worker instance will also need to restart and enable similar services.

```
$ for cmd in restart enable status; do sudo systemctl $cmd kube-proxy kubelet docker; done
```

With this the 2 node Kubernetes setup is complete. The command, **kubectl** (Kubernetes Control), is the primary command used to interact with Kubernetes and the attached cluster. Passing **get** to the command is similar to reading variables with the final argument being **nodes**, any and all attached nodes that are responding will be displayed as shown below.

```
$ kubectl get nodes
NAME                LABELS                          STATUS
10.12.0.110   kubernetes.io/hostname=10.12.0.110   Ready
```

Appendix 1

In this section the useful commands are explained. It is recommended to go through this section before finishing the first exercise.

To pull an image use

```
$ sudo docker pull image_name
```

The following command downloads the image and then runs the container which is named as container name specified in the command. To download and run an image use

```
$ sudo docker run --name container_name  -d image_name
```

Show running containers

```
$ sudo docker ps
```

Show all the containers along with the stopped containers

```
$ sudo docker ps -a
```

The following command shows the container's logs. This displays the stdout of the container. This is very useful for debugging.

```
$ sudo docker logs  container_name
```

Stop the container

```
$ sudo docker stop container_name_or_id
```

Remove the container

```
$ sudo docker rm container_name_or_id
```

To stop and remove container together

```
$ sudo docker rm -f container_name_or_id
```

To remove the image

```
$ sudo docker rmi image_name_or_id
```

References

1. Docker. *Docker- What is docker*. 2015; Available from: https://www.docker.com/whatisdocker.
2. Docker. *Docker Glossary*. 2015; Available from: https://docs.docker.com/reference/glossary/.
3. Cloud, C. *NSF Chameleon Cloud website*. 2015; Available from: www.chameleoncloud.org.
4. Docker. *Docker Documentation*. 2015; Available from: https://docs.docker.com/.
5. PostgreSQL. *PostgreSQL*. 2015; Available from: http://www.postgresql.org.
6. NGINX. *NGINX*. 2015; Available from: http://www.nginx.com.
7. uWGSI. *uWGSI Documentation*. 2015; Available from: https://uwsgi-docs.readthedocs.org/en/latest/.
8. Elllingwood, J. *How to setup uWSGI and nginx to serve python apps on ubuntu 14.04*. 2015; Available from: https://www.digitalocean.com/community/tutorials/how-to-set-up-uwsgi-and-nginx-to-serve-python-apps-on-ubuntu-14-04.
9. Yattag. *Yattag Documentation*. 2015; Available from: http://www.yattag.org/.
10. Docker. *Docker Examples*. 2015; Available from: https://docs.docker.com/examples/postgresql_service/.
11. Django. *Django Documentation*. 2015; Available from: https://www.djangoproject.com/.
12. Institute, O.C. *Cloud and Big Data Lab Docker Hub*. 2015; Available from: https://hub.docker.com/u/cloudandbigdatalab/.
13. Institute, O.C. *CBDL Docker tutorial 1*. 2015; Available from: https://github.com/cloudandbigdatalab/tutorial-cham-docker-1.
14. Docker. *Docker Machine*. 2015; Available from: https://docs.docker.com/machine/.
15. Docker. *Docker Compose* 2015; Available from: https://docs.docker.com/compose/.
16. Docker. *Docker Swarm*. 2015; Available from: https://docs.docker.com/swarm/.
17. Luzzardi, A. *Scaling Docker with Swarm*. 2015; Available from: https://blog.docker.com/2015/02/scaling-docker-with-swarm/.
18. Folding@home. *Folding Website*. 2015; Available from: https://folding.stanford.edu/.
19. Kubernetes. *Kubernetes Pods Documentation*. 2015; Available from: http://kubernetes.io/v1.0/docs/user-guide/pods.html.
20. Kubernetes. *Kubernetes Services Documentation*. 2015; Available from: http://kubernetes.io/v1.0/docs/user-guide/services.html#overview.
21. Kubernetes. *Kubernetes Replication Controller*. 2015; Available from: http://kubernetes.io/v1.0/docs/user-guide/replication-controller.html.
22. Kubernetes. *Kubernetes Labels Documentation*. 2015; Available from: http://kubernetes.io/v1.0/docs/user-guide/labels.html.
23. Ellingwood, J. *An Introduction to Kubernetes*. 2014; Available from: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.
24. Kubernetes. *Kubernetes Documentation*. 2015; Available from: http://kubernetes.io/v1.0/index.html.
25. Community, K. *Namespaces*. 2015; Available from: https://github.com/kubernetes/kubernetes/blob/release-1.0/docs/user-guide/namespaces.md.
26. Kalchev, P. *Installing Kubernetes on CentOS 7*. 2015; Available from: https://www.plamer.net/blog/installing-kubernetes-on-centos-7/.

27. Ethand. *Deploy a Multi-node Kubernetes Cluster on CentOS7*. 2015; Available from: https://devops.profitbricks.com/tutorials/deploy-a-multi-node-kubernetes-cluster-on-centos-7/.
28. Kulkarni, S., *Getting started on CentOS7*. 2015.