A survey of Docker Swarm scheduling strategies

Anwar Hassen
Aalto University

Abstract—Images and containers are the building blocks of docker technology. Images are made of read only layers while containers are instances of images with an addition writable layer.

Managing individual images and containers is error prone and does not scale up. Alternative mechanisms are available. Docker compose helps manage groups of containers on the same host, while Docker swarm is used for load balancing when multiple hosts are involved.

Scheduling in Docker Swarm relies on filters and strategies. Filters are used to narrow the domain of nodes for scheduling by taking the node and container properties as inputs, among other parameters. Strategies are used to decide on which node the next container runs using three alternatives: random, spread, binpack.

I. INTRODUCTION

Virtualization is the abstraction of computer resources such that computer resources, such as memory and CPU are shared among multiple Operating systems [1]. An analogy would be a multi user operating system in which multiple users make use of the same operating system and resources managed by it in an interleaving manner. In both cases, no single entity is able make use of the resources 100% of the time. In these situations, sharing in a controlled manner would lead to a more efficient utilization of resources.

Based on how much of the underlying operating system is shared between virtual machines, virtualization can be done with hypervisors or containers, with the former having its own variations [2]. With hypervisors, each virtual machine has its own operating system while with containers all the virtual machines share the same operating system. Each method has its own benefits and drawbacks. More sharing reduces isolation while less sharing increases the size of images [2].

The benefits of using containers for virtualization and the kernel features that enable isolation such as cgroups, and namespaces are described here [2]. Docker, a container based technology allows different isolation mechanism by providing execution driver API, and as of Docker 0.9 libcontainer [10] is used making docker independent of LXC [11].

In this paper, the building blocks of docker technology and their management alternatives are discussed. The methods used in multi container scenarios, the scheduling mechanisms used for load balancing between nodes are covered.

II. DOCKER PRIMITIVES

A. Read only layers images with with a writable thin layer

Docker primitives are images and their running instances. Docker image size was minimal due to it usage of a union based file system such as AUPS [3]. In a union based file system, for example Unionfs [5],separate physical directories are grouped or merged and shown as one logical unit to the file system. Any conflict between the branches is resolved using a priority based system [6]. As show in figure 1, any change on the upper layer should be applied on the relevant branches atomically. However, atomic operations are challenging when the number of branches is beyond some threshold. One way of ensuring atomicity is by making only one branch writable [6].

When a container is created, it uses an existing image which is made of many layers reflecting changes over time, a process that is handled by the Docker storage driver. Any changes or additions are saved on a new read-write layer.
on top of the current one. Once the container is stopped the changes or additions are lost, leaving the original image intact. If the changes are to persist, a new image consisting of the old image and the new layer needs to be created. And, When multiple containers are run from the same base image, each would keep its own changes on a separate layer. Saving changes results in multiple unique images. Since only changes are saved, the new layers are small in size[9].

B. individual Docker Image and container management

The default storage for docker images is in the public image registry [12], it is possible to search images and download. Once images are pulled, it is possible to see the layers they are made of, and add other layers to create new images. There are different operations to manipulate and make queries on a given image [13].

With containers, besides starting and stopping them, it is possible to get the standard input and output of a running container, attaching to it, executing a one time command, listing processes running in it are possible. Linking with other containers is also possible. One could get a detailed configuration information, in json format, concerning a given container, both running and stopped, by using relevant commands. [14] [15].

When an application involves multiple containers, there is a need for each component to know about other parts of the application. To this end, there are multiple alternatives for linking and networking containers, discussed here [16] [17]. It is also possible to share data between a container and the host on which the container is running using data volumes. The data volumes exist independent of any container making the data live longer, even when the containers are deleted. It is also possible to use containers as data volumes which is useful in a totally containerized system. [18]

III. MANAGING IN GROUP

The mechanisms for image and container management discussed above are not scalable when the number of containers involved is large. Linking between containers and provisioning hosts and scheduling containers on specific nodes and becomes challenging. This is where a mechanism for managing containers in groups comes into play. In this category, one can use Docker machine, Docker Compose and Docker swarm, among others.

A. docker compose

As discussed in the manual management section, creating and linking individual containers and corresponding images is error prone. When using a Microservices architectural approach for developing applications, an application is made of small, independent services with relevant communication channels between some of the services [22].

With Docker compose, a single file is used to create and configure services. Running the application as a whole requires less effort as all services are handled from one tool. Each service is configured and its links with other services is explicitly declared in the same file. [23] An example of docker compose file is shown below [25].

```json
version: '2'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    depends_on:
      - redis
  redis:
    image: redis
```

Once services and their interactions are defined, scaling a specific service is possible [24]. The application and its components can be reproduced in a consistent manner which makes docker compose preferable over managing containers individually.

B. Docker swarm

Docker swarm is a clustering mechanism for load balancing that provides logical view of clusters of Docker nodes. Docker compose, on the other hand, is used for managing multiple containers on a single host machine. With Docker swarm, the user does not know on which specific node a container is running, and failures of some the nodes does not break the cluster as Swarm may run the same container on a different node. The Swarm cluster has a Swarm manager and worker nodes, all running Docker Swarm.

Given a set of nodes in a cluster, the Swarm manager needs to decide where to run a container. First, it needs to select some nodes based on some criteria. Once eligible nodes are selected, the next process is to choose one of them every time a container is started. Docker swarm uses filters to narrow the domain of nodes and ranking strategies to schedule a container on a particular node as described below.

1) Swarm filters: Filters are conditions that are inputs to the scheduler for choosing some nodes among all available ones. Filters could be of two types, those based on the properties of the node or that of the containers. Node filters are constraints such as operating system of the host, health of the node, or containerslot which limits the number of containers on a given host, stopped containers inclusive [19].

Container filters are affinity, dependency and port. Affinity, among other cases, helps run new containers in proximity to existing containers using identifiers. Another case is running a container on a node with specific image. Dependency filters are used for creating containers next to other containers
they depend on. The port filter maps containers with nodes on the matching port number on the host. Both node and container filters result in a pool of nodes that are available for scheduling containers to run on them. The decision which one of them to choose is based on their rank which is discussed next [19].

2) Swarm ranking strategies: The three ranking strategies are random, binpack and spread, which is the default. With random, nodes are selected randomly for containers to run on them, also used for node selection when the nodes are of equal metrics. One case is when nodes have the same amount of available RAM. With binpack, nodes with more containers are preferred which results in less fragmentation as it intends to pack as many containers as possible on a single node. On the other hand, spread strategy prefers nodes with less containers on them, stopped containers inclusive[twenty]. A related matter is rescheduling, which is a decision to be made when a node on which a container runs has failed. Rescheduling is a configurable option, and if set, it restarts the container on a different node. The default is not to restart it [20].

C. Observations

With the aim of measuring performance difference between the three strategies, test was done with a two and five node Swarm clusters. In the test with two nodes, images were pulled on demand only when a node was selected as the next host. With the five nodes test, images were pulled upon cluster creation. The maximum number of containers per node was fixed for the spread and random strategies. However, with binpack the value was variable as a node will always be packed before choosing the next one. The number of containers was increased gradually, resulting in running time units shown in table 1.

<table>
<thead>
<tr>
<th>Number of containers</th>
<th>50</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>spread</td>
<td>1.0</td>
<td>2.3</td>
<td>10.8</td>
<td>22.2</td>
<td>43.1</td>
</tr>
<tr>
<td>random</td>
<td>1.2</td>
<td>3.4</td>
<td>11.6</td>
<td>27.5</td>
<td>44.1</td>
</tr>
<tr>
<td>binpack</td>
<td>5.1</td>
<td>8.3</td>
<td>15.9</td>
<td>28.0</td>
<td>44.6</td>
</tr>
</tbody>
</table>

Table 1. Running time as the number of container increases

The result was that as the number of containers increases, the difference in performance between the strategies gets smaller, as shown in Table 1 for the five node cluster. However, when using binpack, some some issues were encountered. One was failure to schedule the first container, regardless of when images were pulled. The other issue, when a node was fully packed and a new node needed to be selected, a considerable delay was observed

IV. CONCLUSIONS

Management of docker images and containers on multiples nodes requires a clustering mechanisms. One such tool is Docker swarm. It uses filters and ranking strategies to schedule a container on a given node. Tests were done to observe the performance differences between the three strategies, for the default filters.

Of all three strategies, binpack was expected to perform better with assumption that decision were to be made once when a node was full packed which could have resulted in decision equal to the number of available nodes. With spread, the sequence of nodes for the first round could have served as a decision for subsequent rounds. However, since the number of nodes and containers involved for the experiment was minimal.

When resources are available, testing with millions of containers in thousands of nodes could reveal more pattern and performance differences between the three strategies. The focus of the test should be on spread and binpack, to see if decision were made faster without needing computation.

REFERENCES

[22] Mesoservices, a definition of this new architectural term. url: http://unionfs.filesystems.org/.