DRAP: Dynamic and Resource-Aware Placement of Docker Containers in a Heterogeneous Cluster

***Abstract*—This is an abstract.**

I. Introduction

II. Related Work

III. Background and Motivation

In this section, we discuss the Docker containers' framework and Swarmkit, the default cluster management tool.

A. Docker Containers

Container technology or containerization is a virtualization method for deploying and running distributed applications without launching an entire virtual machine for each of them. Instead, multiple isolated service units of the application, called containers, are sharing the host operating system and physical resources. The concept of container virtualization is yesterday's news, Unix-like operating systems leveraged the technology for over a decade. However, new containerization platforms, such as Docker, make it into the mainstream of application development. Based on previously available open source technologies (e.g. cgroup), Docker introduces a way of simplifying the tooling required to create and manage containers.

A Docker worker machine runs a local Docker daemon. New containers may be created on a worker by sending commands to its local daemon, such as "docker run -it ubuntu bash". A Docker container image is a lightweight, standalone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, and settings. In general, each container targets on a specific service of an application. If the application needs to scale up this particular service, it initiates duplicated containers by using the same image. One physical machine can host multiple application with different services in a standalone mode.

Fig 1 illustrates the structure of a physical machine that is hosting four Docker containers for two applications. As the figure shows, the $AppA$ includes two services that are provided by $AppA1$ and $AppA2$ and $AppB$ contains one service which is provided by two Docker containers, $AppB1$ and $AppB1'$.

B. Container Orchestration

When deploying the applications into a production environment, it's difficult to achieve resilience and scalability on a single container host. Typically, a multi-node cluster is used to provide the infrastructures for running containers at scale. Introduced by Docker, Swarmkit is an open source toolkit for container orchestration in the cluster environment.

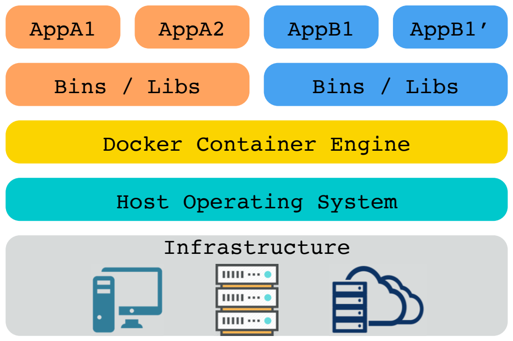

Fig. 1. Docker Containers

There are two types of nodes in a cluster that running Swarmkit, worker nodes and manager nodes. Worker nodes are responsible for running tasks and on the other hand, manager nodes accept specifications from the user and are responsible for reconciling the desired state with the actual cluster state. Fig. 2 shows the decentralized architecture of a SwarmKit cluster. A manager node is in charge of several worker nodes and there is a overlap between manager nodes to tolerate failures. Worker and manager nodes are equal in the system since a worker node can be promoted to a manager and a manager node can be demoted to a worker. Manager nodes are formed into a Raft consensus group to maintain global cluster's states. The Raft consensus algorithm is used to ensure that all the manager nodes that are in charge of managing and scheduling tasks in the cluster, are storing the same consistent states.

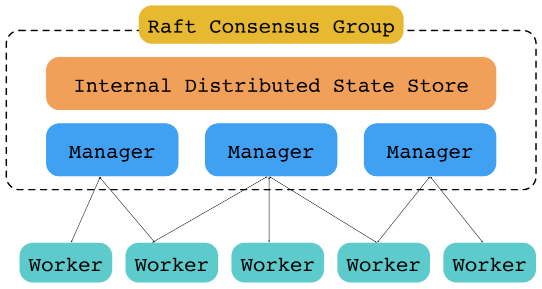

Fig. 2. Docker S[illegible]it

A Docker container can be initiated with specific requirements (e.g. memory and CPU) and user-defined labels. The scheduler that runs on a manager combines the user-input information with states of each node to make various scheduling decisions, such as choosing the best node to perform a task. Specifically, it utilizes filters and scheduling strategies to assign tasks. There are four filters in available. *ReadyFilter:* checks that the node is ready to schedule tasks; *ResourceFilter:* checks that the node has enough resources available to run; *PluginFilter:* checks that the node has a specific volume plugin installed. *ConstraintFilter:* selects only nodes that match certain labels.

If there are multiple nodes pass the filtering process, Swarmkit supports three scheduling strategies, spread (currently available), binpack and random (under development based on Swarm Mode). *Spread strategy:* places a container on the node with the fewest running containers. *Binpack strategy:* places a container onto the most packed node in the cluster. *Random strategy:* randomly places the container into the cluster.

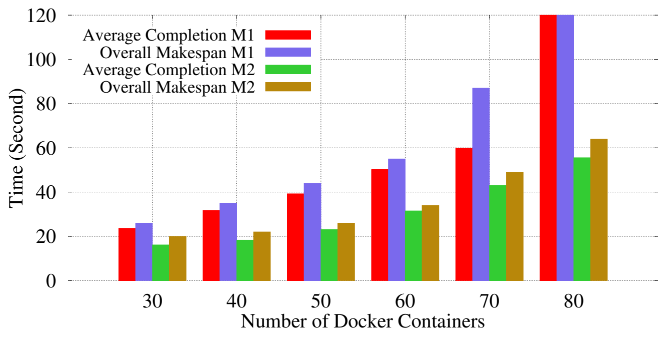

Fig. 3. Starting Dockers on a single machine

The default spread strategy, which attempts to schedule a service task based on the number of active containers on each node, can roughly assess the resources on the nodes. However this assessment fails to reflect various nodes in a heterogeneous cluster setting. Considering the heterogeneity, the nodes in such a cluster have different configurations in terms of memory, CPU and network. Therefore, running the same amount of containers on these nodes result in different experiences. Fig 3 plots the average starting delay of and overall makespan of set of Tomcat Docker containers. We conduct the experiments on two machines, M1 with 8GB memory, 4-core CPU and M2 has 16GB memory and 8-core CPU. On each particular machines, M1 or M2, we can see that the more containers it hosted, the larger starting delay and makespan. However, M1 costs 23.67s in average to start 30 Tomcat containers and M2 costs 18.32s to start 40 containers. Additionally, when trying to initiate 80 Tomcat containers, M1 fails to complete the job (infinite completion and makespan cost), while M2 finishes it smoothly.

IV. System Architecture

V. Problem Formulation

VI. Solution

VII. Evalution

VIII. Evalution