DRAPS: Dynamic and Resource-Aware Placement Scheme for Docker Containers in a Heterogeneous Cluster

Ying Mao*, Jenna Oak*, Anthony Pompili*, Tao Han†
*Department of Computer Science, The College of New Jersey, Email: {maoy, oakj1, pompila1}@tcnj.edu
†Department of Electrical & Computer Engineering, University of North Carolina at Charlotte, Email: tao.han@uncc.edu

Abstract—**Virtualization is a promising technology that has facilitated cloud computing to become the next wave of the Internet revolution. Adopted by data centers, millions of applications that powered by various virtual machines improve the qualify of services. Although virtual machines are well-isolated among each other, they suffer from redundant boot volumes and slow provisioning time. To address the limitations, containers are born to deploy and run distributed applications without launching entire virtual machines. As a dominant player, Docker is an open-source implementation of container technology. When managing a cluster of Docker containers, the management tool, Swarmkit, does not take the heterogeneities in both physical nodes and virtualized containers into consideration. The heterogeneity lies in the fact that different nodes in the cluster may have various configurations, in terms of resource types and availabilities, etc., and the demands generated by services are varied, such as CPU-intensive (e.g. Clustering services) as well as memory-intensive (e.g. Web services). In this paper, we targeted on investigating the Docker container cluster and developed, DRAPS, a resource-aware placement s to boost the system performance in a heterogeneous cluster.**

I. Introduction

In the past decades, we have witnessed a spectacular information explosion over the Internet. Hundreds of thousands of users are consuming the Internet through various services, such as websites, mobile applications, and online games. The service providers, at the back-end side, are supported by state-of-art infrastructures on the cloud, such as Amazon Web Service [1] and Microsoft Azure [2]. Targeting on providing the services at scale, virtualization is one of the emerging technologies used in data centers and cloud environments to improve both hardware and development efficiency.

At the system level, the virtual machine is a widely-adopted virtualization method [3], which isolate CPU, memory, block I/O, and network resources, etc [4]. In a large-scale system, however, providing services through virtual machines would mean the users are probably running many duplicate instances of the same OS and many redundant boot volumes [5]. Recent research shows that virtual machines suffer from noticeable performance overhead, large storage requirement, and limited scalability [6].

To address the limitations, containers are designed for deploying and running distributed applications without launching entire virtual machines. Instead, multiple isolated service units of the application, called containers, share 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. On a physical machine, containers are essentially just regular processes, in the system view, that enjoy a virtualized resource environment, not just CPU and memory, but also bandwidth, ports, disk i/o, etc.

We use "Docker run image" command to start a Docker container on physical machines. In addition to the disk image that we would like to initiate, users can specify a few options, such as "-m" and "-c", to limit a container's access to resources. While options set a maximum amount, resource contention still happens among containers on every host machine. Upon receiving the "Docker run" commands from clients, the cluster, as the first step, should select a physical machine to host those containers. The default container placement scheme, named Spread, uses a bin-pack strategy and tries to assign a container on the node with the fewest running containers. While Spread aims at equally distributing tasks among all nodes, it omits two major characteristics of the system. First of all, the nodes in a cluster do not necessary have to be identical with each other. It is a common setting to have multiple node types, in terms of total resource, in the cluster. For example, a cutting edge server can easily run more processes concurrently than a off-the-shelf desktop. Secondly, the resource demands from containers are different. Starting with various images, services provided by containers are varied, which leads to a diverse resource demands. For instance, a clustering service, e.g. Kmeans, may need more computational power and a logging service, e.g. Logstash, may request more bandwidth.

In this project, we propose a new container placement scheme, DRAPS, a Dynamic and Resource-Aware Placement Scheme. Different from the default Spread scheme, DRAPS assigns containers based on current available resources in a heterogeneous cluster and dynamic demands from containers of various services. First, DRAPS identifies the dominate resource type of a service by monitoring containers that offer this service. It, then, places the containers with complementary needs to the same machine in order to reduce the balance resource usages on the nodes. If one type of resource, finally, becomes a bottleneck in the system, it migrates the resource-intensive containers to other nodes. Our main contributions are as follows:

- We first introduce the concept of dominate resource type that considers the dynamic demands from different services.
- We propose a complete container placement scheme, DRAPS, which assigns the tasks to appropriate nodes and balance resource usages in a heterogeneous cluster.
- We implement DRAPS into the popular container orchestration tool, Swarmkit, and conduct the experiment with 18 services in 4 types. The evaluation of the cloud demonstrates that DRAP outperforms the default Spread and reduces usage as much as 42.6% on one specific node.

II. Related Work

Virtualization serves as one of the fundamental technologies in cloud computing systems. As a popular application, virtual machines (VMs) have been studied for decades. However, in the reality, VMs suffer from noticeable performance overhead, large storage requirement, and limited scalability [6]. More recently, containerization, a lightweight virtualization technique, is drawing increasingly popular in both industry and academic.

The benefits and challenges of containerized systems have been studied in many aspects. A comprehensive performance study is presented in [7], where it explores the traditional virtual machine deployments, and contrast them with the use of Linux containers. The evaluation focuses on overheads and experiments show that containers result in equal or better performance than VMs. Although containers outperform VMs, the research [8] shows that the startup latency is considerably larger than expected. This is due to a layered and distributed image architecture, in which copying package data accounts for most of container startup time. The authors propose Slacker that can significantly reduce the startup latency. While Slacker reduces the amount of copying and transferring packages, if the the image is locally available, the startup could be even faster. CoMICon [9] addresses the problem by sharing the image in a cooperative manner From different aspect, SCoPe [10] tries to manage the provisioning time for large scale containers. It presents a statistical model to characterize the provisioning time in terms of system features and this model is used to guide provisioning strategy.

Besides the investigations on standalone containers, the cluster of containers is another important aspect in this field. Docker Swarmkit [11] and Google Kubernetes [12] are dominate cluster management tools in the market. The authors in [13], first, conduct a comparison study of scalabilities under both of them. Then, Firmament is proposed to achieve low latency in large-scale clusters by using multiple min-cost max-flow algorithms. On the other hand, focusing on workload scheduling, the paper [14] describes an Ant Colony Optimization algorithm for a cluster of Docker containers. However, the algorithm does not distinguish various containers, which are usually have a divese requirements.

In this paper, we investigate the container orchestration in the prospective of resource awareness. While users can set limits on resources, containers are still competing the resources in a physical machine. Starting from different images, the containers target various services, which results in different requirements on resources. Through analyzing the dynamic resource demands, our work studies a node placement scheme that balance the resource usages in a heterogeneous cluster. The

III. Background and Motivation

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

A. Docker 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 $AppB1'$.

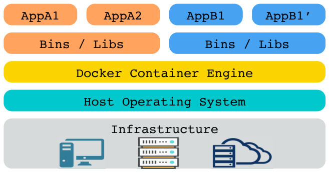


Fig. 1: Docker Containers

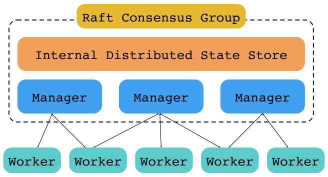


Fig. 2: Docker Swarmkit

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.

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.

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.

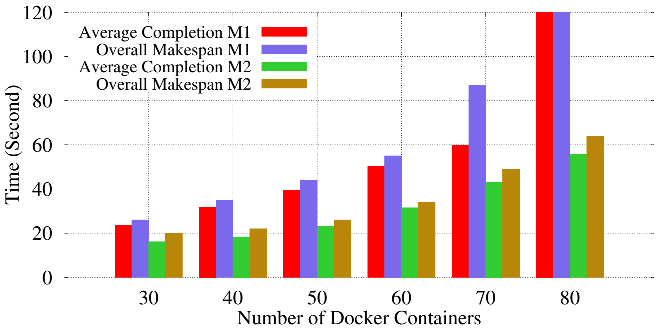


Fig. 3: Starting Dockers on a single machine

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 and M2 finishes it

IV. DRAP System

In this section, we, first, discuss the original modules in Docker manager and worker. Then, we illustrate the major components in DRAP.

A. Framework of Manager and Worker Nodes

As described in the previous section, there are multiple managers and workers in the system. A manager has six hierarchical modules. **Client API**, accepts the commands from clients and creates service objects; **Orchestrator**, handles the lifecycle of service objects and manages mechanics for service discovery and load balancing; **Allocator**, provides network model specific allocation functionality and allocates IP addresses to tasks; **Scheduler**, assigns tasks to worker nodes; **Dispatcher**, communicates with worker nodes, checks their states, and collects the **heartbeats** from them;

A worker node, on the other hand, manages the Dockers containers and send back their states to managers through periodical heartbeat messages. An **executor** is used to run the tasks that assign to the containers in this worker.

B. DRAP modules

To simplify the implementation, we integrate the DRAP components into the current framework. As shown on Fig 4, it mainly consists of three parts, a container monitor that resides in the worker nodes, a worker monitor, and a DRAP scheduler that implement in manager nodes.

Container Monitor: a container monitor collects the runtime resources usage statistics of Docker containers on worker nodes. At each application level, the monitored resources contain memory, CPU percentage, block I/O, and network I/O. The average usage report in a given time window of top users will be injected into the DRAP-Heartbeat messages and sent back to managers. At the host system level, the tracking information includes, I/O wait, reminder percentage of available memory, CPU, and bandwidth. The information is used by worker nodes to conduct a self-examination to identify the bottleneck of itself. If a bottleneck is found, an DRAP-Alert message will be produced and sent back to managers.

Work Monitor: a worker monitor processes the messages from worker nodes. It maintains a table for each worker and the corresponding containers. Through analyzing the data, it will generate tasks, such as migrating a resource-intensive container to another host.

DRAP-Scheduler: the DRAP-Scheduler assigns a task to a specific node based on the current available resources. For a duplicated Docker container, DRAP-Scheduler checks its characteristics on resources consumption, such as memory intensive, through the records of the previous containers in the same services.

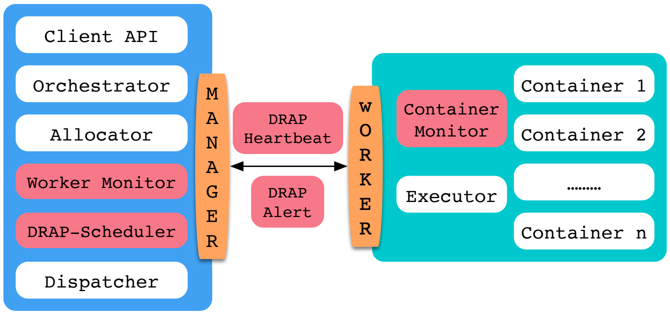


Fig. 4: Docker Framework with DRAP Implemention

V. Problem Formulation

The DRAP scheduler aims to optimize the container placement such that the available resources on each worker node are maximized. In this paper, we assume that a container requires four resources which includes memory, CPU, bandwidth, and I/O for running its services. Since the workload in a container changes over time, the resource requirements in a container also exhibit temporal dynamics. Therefore, we formulate the resource requirements of a container as a function of time. Denote $r_i^k(t)$ as the kth resource requirement of the ith container at time t. For example, $r_i^1(t)$, $r_i^2(t)$, $r_i^3(t)$, and $r_i^4(t)$ represent the memory, CPU, bandwidth, and I/O requirements of the ith container at time t, respectively. Let $x_{i,j} = \{0, 1\}$ be the container placement indicator. If $x_{i,j} = 1$, the ith container is placed in the jth work node. Denote W_j^k as the total amount of the kth resource in the jth work node. Let $\mathcal{C}$, $\mathcal{N}$, $\mathcal{K}$ be the set of containers, work nodes, and the resources, respectively. The utilization ratio of the k resource in the jth work node can be expressed as

$$u_j^k(t) = \frac{\sum_{i \in \mathcal{C}} x_{i,j} r_i^k(t)}{W_j^k} \tag{1}$$

We assume that the utilization ratio of the jth work node is defined by its highest utilized resource. Then, the utilization ratio of the jth work node is $\max_{k \in \mathcal{K}} u_j^k(t)$. The the highest resource utilization among all the work nodes can be identified as

$$\nu = \max_{j \in \mathcal{N}} \max_{k \in \mathcal{K}} u_j^k(t). \tag{2}$$

Since our objective when designing the DRAP scheduler is to maximize the available resources in each worker node, the DRAP scheduling problem can be formulated as

$$\max_{x_{i,j}} \quad \nu \tag{3}$$

$$s.t. \quad \sum_j x_{i,j} = 1; \forall i \in \mathcal{C}; \tag{4}$$

$$u_j^k(t) \leq 1, \forall k \in \mathcal{K}, \forall j \in \mathcal{N}. \tag{5}$$

The constraint in E.q. (4) requires that each container should be placed in one worker node. The constrain in E.q. (5) enforces that the utilization ratio of any resource in a worker is less than one.

Lemma 1. *The DRAP scheduling problem is an NP-hard problem.*

Proof: In proving the Lemma, we consider a simple case of the DRAP scheduling problem in which the resource requirements of each container are constant over time.The simplified DRAP scheduling problem equals to the multidimensional bin packing problem which is NP-hard [15]–[17]. Hence, the lemma can be proved by reducing any instance of the multidimensional bin packing to the simplified DRAP scheduling problem. For the sake of simplicity, we omit the detail proof in the paper. ∎

TABLE I: Notation Table

M_{id} / W_{id}	Manager ID / Worker ID
S_{id} / C_{id}	Service ID / Container ID
W_j^m	The total memory on a specific worker
W_j^c	The total CPU on a specific worker
W_j^b	The total bandwidth on a specific worker
W_j^o	The total I/O wait time on a specific worker
$\{KS\}$	A set stores known services IDs
$DOM(S_{id})$	A function records dominate resource attribute

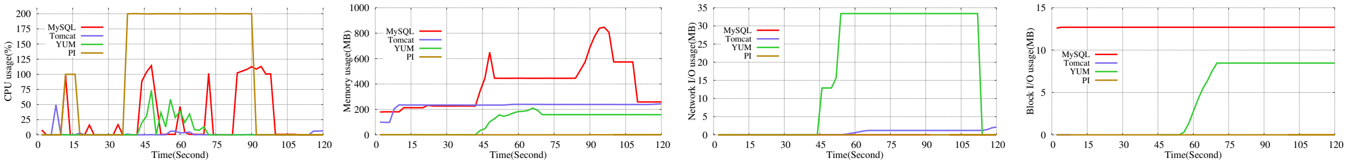
VI. DRAP in a Heterogeneous Cluster

Previously, we discuss the different modules in DRAP and their major responsibilities. In this section, we present the detailed design of DRAP, in a heterogeneous cluster, which aims to increase resource availability on each worker node and boost the service performance. To achieve the objectives, DRAP system consists of three strategics, 1) Identify dominate resource demands of containers; 2) Initial container placement; 3) Migrate a container

A. Identify Resource Demands of Containers

Before improving the overall resource efficiency, the system needs to understand the dynamic resource demand of various containers. A container is, usually, focused on providing a specific service, such as web browsing, data sorting, and database querying. Different algorithms and operations will be applied to the services, which result in a diverse resource demands. As an intuitive example, we conduct the experiments on NSF Cloudlab [18] (M400 node hosted by University of Utah). The containers are initiated by using the following four images and the data is collected through "docker stats" command.

1) MySQL: the relational database management system. Tested workloads: scan, select, count, join.
2) Tomcat: provides HTTP web services with Java. Tested workloads: HTTP queries at 10/second and 20/second of a HelloWorld webpage.
3) YUM: a software package manager that installs, updates, and removes packages. Tested workload: download and install "vim" package.



(a) Usage of CPU (b) Usage of Memory (c) Usage of Network I/O (d) Usage of Block I/O

Fig. 5: Resource demonds under different workloads on four services, MySQL, Tomcat, YUM, PI.

4) PI: a service to calculate PI. Tested workload: top 3,000 digits with single thread, top 7,500 digits with two threads.

Figs. 5a to 5d plots the dynamic resource demands under different workloads on the above four Docker containers. The figures illustrate very diverse usage patterns on four types of resources, CPU, memory, network I/O and block I/O. For example, without workload, container PI consumes very limited resources. However, when the jobs arrive at 10th and 38th second, the CPU usage jumps to 100% for a single thread job and 200% for a two-threads job. The usages of the other three types of resources still remain at very low level. For MySQL service container, with tested operations, the CPU usage shows a burst when client submit a request. At time 84, a "join" operation that involves 3 tables is submitted, we can find CPU usage jumps as well as memory usage. The reason is that join operation needs lots of computation and copies of tables in memory. Different usage trends are found on YUM and Tomcat services, where YUM uses less CPU and memory, but more network I/O and block I/O to download and install packages. On the other hand, Tomcat consumes a very small amount of network I/O and block I/O due the size of a tested HelloWorld page, but more than 200MB memory is used to maintain the service.

To balance the resource usage, it's crucial to place the containers with complementary demands on the same worker, As shown on the graphs, despite multiple types of resources, there is a dominate resource demand of a service in a given period.

In DRAP, we need to identify the a dominate resource demand for each service. A manager, in the system, can monitor all containers' resource usage and group them by their associated service ID. Suppose the service $s_i \in S$ contains m running containers that store in a set, RC_{s_i}. The resources consumed by $c_i \in RC_{s_i}$ is denoted by a vector, R_{c_i}, where each attribute, r_i, in the vector represents a type of resources, such as memory and CPU. If there are q types of resources in the system, the average resource cost of s_i is a vector, R_{s_i},

$$\begin{aligned} R_{s_i} &= \sum\nolimits_{c_i \in RC_{s_i}} R_{c_i} \\ &=< \sum_{c_i \in RC_{s_i}} r_1/m, \sum_{c_i \in RC_{s_i}} r_2/m, ..., \sum_{c_i \in RC_{s_i}} r_q/m > \end{aligned}$$

On the worker nodes, there is a limited amount of resources in each type. The resource limit is a vector that contains q attributes, $< l_1, l_2, ..., l_q >$. The limit of a system, $< L_1, L_2, ..., L_q >$, is obtained by sum up the the vectors from workers. Therefore, R_{s_i} can be represented by a percent of total resources in the system, for the i^{th} type, the container cost for s_i in average is $\sum_{c_i \in RC_{s_i}} r_i/m \div L_i$. With the analysis, we define the dominate function,

$$DOM(s_i) = max\{\sum_{c_i \in RC_{s_i}} r_i/m \div L_i\}$$

Function $DOM(s_i)$ returns the type of a dominate resource demand of service s_i within a given time period. Depending on the running containers for s_i and the current cost of them, the value of $DOM(s_i)$ changes along with system goes.

B. Initial Container Placement

To use a Swarmkit cluster, clients need to execute a "docker run" command to start a new container. Therefore, the first task for the cluster is to choose a worker node to host the container. As discussed in section III, the default container placement strategy fails to take dynamic resource contention into consideration. It is because that the managers in Swarmkit do not have a mechanism to monitor the current available resource. DRAP, on the other hand, addresses the problem by introducing DRAP-Heartbeat. DRAP-Heartbeat is an enhanced heartbeat message that not only the states of worker node, but also the containers' resource usage over a given time window, the usage includes, memory, cpu, bandwidth, and block i/o. At the manager side, the data will be organized into a table that keep tracking the current available resource on each worker and its corresponding containers' resource usages.

Running on manager, Algorithm 1 assigns a container placement task to a specific worker. Firstly, each manager maintains a known service set that records dockers' characteristics, such as the usage of memory, CPU, bandwidth, and block i/o (line 1). The initial candidate worker are all running workers (line 2). When a starting new container task arrives, the algorithm applies all filters that user specified to shrink the candidate work set, W_{cand} (line 3-6). Then, it checks whether the container belongs to a known service (line 7). If it is, S_{dom} parameter will be used to store the container's dominate resource attribute in four types, memory ("M"), CPU ("C"), bandwidth ("N"), and block i/o ("IO") (line 8). The W_{cand} set will be sorted according to the dominate resource attribute and return the W_{id} with highest available resource in S_{dom} type (line 9-20). If the container cannot be found in $\{KS\}$, W_{id} with the highest available resource in average will be chosen (line 22-23).

Algorithm 1 Container Placement on Managers

```
Maintains a known characteristics service set ${KS}$
${W_{cand}}$ = All running $W_{id}$;
Function ContainerPlacement($S_{ID}$)
for $w_{id} \in \{W_{cand}\}$ do
    if !$Filters(w_{id})$ then
        Remove $w_{id}$ from $\{W_{cand}\}$
if $S_{ID} \in \{KS\}$ then
    $S_{DOM} = DOM(S_{ID})$
    Sort $W_{cand}$ according to $r_{S_{DOM}}$
    Return $w_{id}$ with highest $r_{S_{DOM}}$
else
    Sort $\sum_{i=0}^{i=q} r_i/m$ for $w_{id} \in W_{cand}$
    Return $w_{id}$ with highest average available resource
```

Algorithm 2 Process DRAP Alert Message from w_i

```
1:  $\{W_{cand}\}$ = All running workers expect $w_i$;
2:  Function ReceiveAlertMsg($C_{id}$)
3:  Extract the bottleneck type $r_i$
4:  Find corresponding $S_{id}$ for $C_{id}$
5:  if $\forall w_{id} \in W_{cand} \to S_{id} \in w_{id}$ then
6:      if $S_{id} \in \{KS\}$ then
7:          $S_{DOM} = DOM(S_{id})$
8:          Sort $W_{cand}$ according to $r_{S_{DOM}}$
9:          Return $w_{id}$ with highest $r_{S_{DOM}}$
10:     else
11:         Sort $W_{cand}$ according to $r_i$
12:         Return $w_{id}$ with highest $r_i$
13: else
14:     for $w_{id} \in W_{cand}$ do
15:         if $S_{id} \in w_{id}$ then
16:             Remove $w_{id}$ from $W_{cand}$
17:     if $S_{id} \in \{KS\}$ then
18:         $S_{DOM} = DOM(S_{id})$
19:         Sort $W_{cand}$ according to $r_{S_{DOM}}$
20:         Return $w_{id}$ with highest $r_{S_{DOM}}$
21:     else
22:         Sort $W_{cand}$ according to $r_i$
23:         Return $w_{id}$ with highest $r_i$
```

C. Migrating a Container

In a Swarmkit cluster, resource contention happens on every worker. The Container Monitor, a module of DRAP, that runs on each worker node records resource usages of all hosting containers. In addition, the worker keeps tracking available resources on itself. Whenever it finds one type of resources is draining and becomes a bottleneck, it sends to managers a DRAP Alert message that contains the bottleneck type and the most costly container of this type. Upon receiving the DRAP Alert message, the manager needs to migrate this container to an appropriate worker and kill it on the worker to release the resources.

Algorithm 2 presents the procedure to process an alert message from w_i. It first builds a candidate set W_{cand}, which includes all running workers expect w_i that sends the alert (line 1). Then, the manager extracts the resource type, r_i that causes the bottleneck and finds the corresponding S_{id} for the C_{id} (lines 2-4). With W_{cand} and S_{id}, the algorithm can decide whether this S_{id} is a global service (line 5). If S_{id} is a global service and it is in the known service set, $\{KS\}$, the algorithm returns w_{id} that is included the W_{cand}, with highest available $r_{S_{DOM}}$. On the other hand, it returns w_{id} with highest available r_i if S_{id} is not in $\{KS\}$ and S_{DOM} is unknown (lines 6-12). When S_{id} is not a global service, we want to increase the reliability of S_{id} by placing its containers to different workers as much as possible. In this situation, we have a similar process expect a different W_{cand}, where W_{cand} is all running workers that do not hosting any containers for S_{id} (lines 13 - 23).

VII. Performance Evaluation

In this section, we evaluate the performance of `DRAPS` and compare it with Swarmkit management tool from Docker.

A. System Implementation and Testbed Setup

We implement our new container placement scheme, `DRAPS`, on Docker Community Editions (CE) v17. As described in section IV, the major modules in `DRAPS` are integrated into the existing Docker Swarmkit framework.

To evaluate `DRAPS`, we build a heterogeneous cluster on Alibaba Cloud [19], which supports multiple types of computing nodes. Specifically, we use three different types of instances, small (1 CPU core and 4G memory), medium (4 CPU cores and 8G memory) and large (8 CPU cores and 16G memory). In the small-scale testing, we setup a cluster with 3 nodes, one of each type, and configure it with 1 manager and 3 worker (one of the three physical node hosts both manager and worker). In experiments of scalability testing, we configure the cluster with 1 manger and 9 workers, 3 instances of every type.

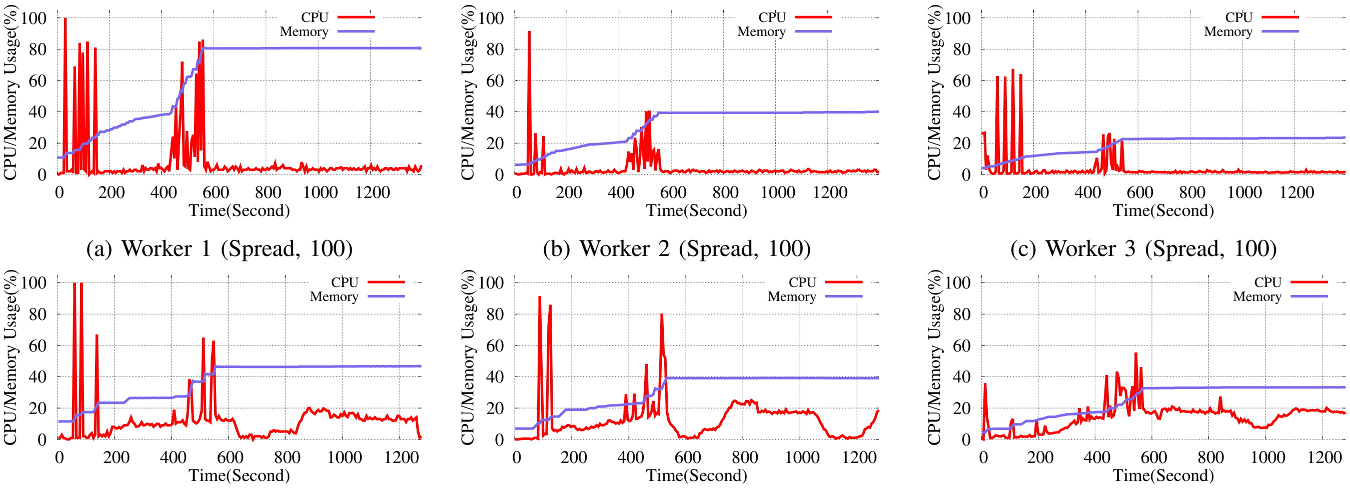
B. Workloads

The main objective of `DRAPS` is to understand the resource demands of services and place them on the appropriate worker nodes. As we discussed in section VI-A, the characteristics of services are varied. Therefore, workloads for the cluster is images that target on various services. In the evaluation, we select 18 different images in 4 types from Docker Hub [20] to build our image pool.

1) **Database Services**: MongoDB, MySQL, Postgres, Cassandra, RethinkDB;
2) **Storage/Caching Services**: Registry, Memcached;
3) **Web Services**: Tomcat, Httpd, Redis, HAProxy, Jetty, Nginx, GlassFish;
4) **Message Services**: RabbitMQ, Apache ZooKeeper, ActiveMQ, Ghost;

C. Evaluation

1) Idle containers: In this subsection, we present the result of a cluster with idle containers. If a container is in a running state but does not serving any clients, we call it a idle container. Idle container is an important concept since every



(a) Worker 1 (Spread, 100)

(b) Worker 2 (Spread, 100)

(c) Worker 3 (Spread, 100)

(d) Worker 1 (DRAPS, 100)

(e) Worker 2 (DRAPS, 100)

(f) Worker 3 (DRAPS, 100)

Fig. 6: Memory and CPU resources usage comparison between Spread and DRAPS placement scheme (100 containers)

node, right after initialization will act as an idle container. Understanding the resource demands of an idle container will help us select In these experiments, we first randomly choose 14 images form the pool, and each image will be used to initiate 10 containers. Therefore, there are 140 containers in the cluster. Those containers are started one by one with 5 seconds interval. This is because previous containers will result in different available resources on worker nodes, which we can utilize to test DRAPS.

Fig 6 illustrates a comparison of memory and CPU usages between Spread, a Swarmkit default placement scheme, with DRAPS. As we can see from the subfigures, most of the CPU usage happens during 0 to 500s. This is caused by submission pattern that used to initiate containers. The percentage grows continuously from 0 to 500 since we have 100 containers and the submission interval is 5 seconds. While in both systems, the usage of CPU stays in a low level in average. However, the memory usage keeps increasing along with the number of containers on each worker. Due to the idle container setting, the utilization of memory is stable after 500s (all the containers have successfully initiated). There are some jitters on the curve of CPU, which is because that some supporting programs, such as Docker engine and service daemon, are running on the same worker and, of course, they are consuming resources. Comparing the memory usage rates after 500s, DRAPS significantly reduces rate on worker 1, from 80.5% to 46.7%. On worker 2, Spread and DRAPS achieve similar performance on memory, 39.1% verse 40.6%. On worker 3, Spread results in 23.6% and DRAPS consumes 33.3%. The DRAPS outperforms Spread by considering the heterogeneity in the cluster and various resource demands of services. When a task arrives at the system, it selects a worker based on the service demands and current available resources. Fig 7 shows the number of containers on workers. For Swarmkit with Spread, it uses a bin-pack strategy and tries to equally distribute the containers to every worker, which results in 34, 33, 33 containers for worker 1, 2, 3. While in DRAPS, worker 3 has more powerful than others and hosts more containers than worker 1, which has limited resource.

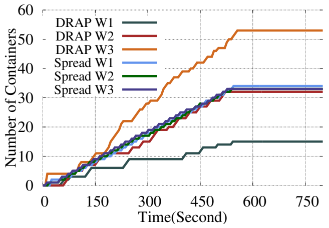


Fig. 7: Number of containers on each worker

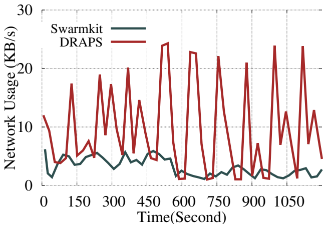
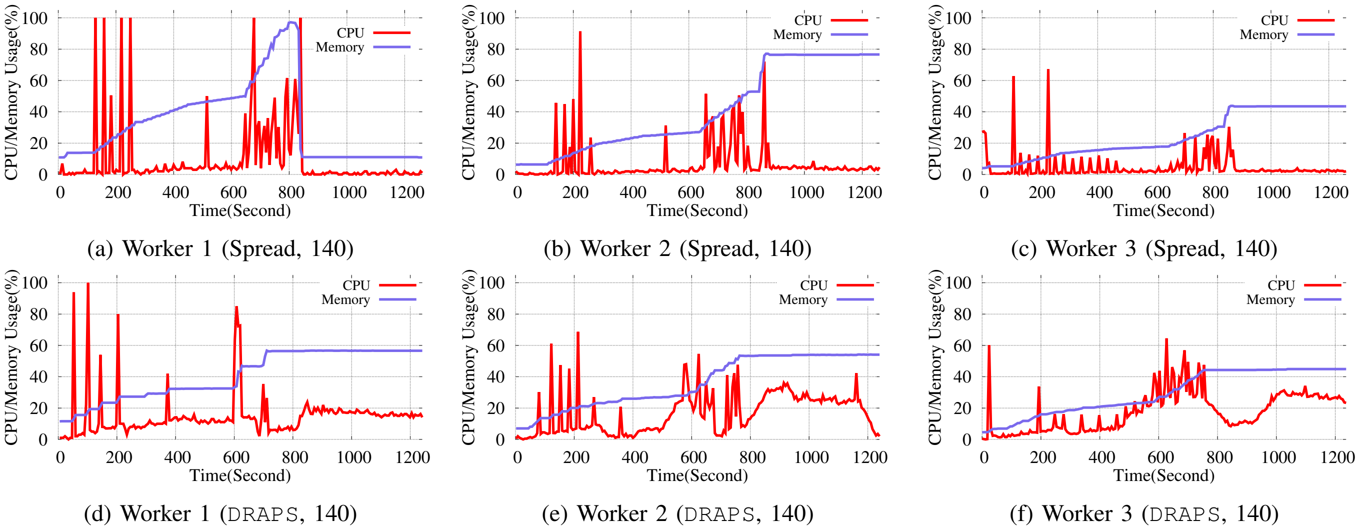


Fig. 8: Network consumption comparison on worker 3

While DRAPS achieves better performance, it introduces more data transfers between managers and workers through heartbeat messages. Fig 8 plots the network consumption of Swarmkit and DRAPS on worker 3, which hosts both a manager and a worker. As expected, DRAPS consumes more bandwidth than Swarmkit due to the enhanced heartbeat messages includes more statistical information resource usages of containers. Considering the distributed architecture, the system can have multiple managers and each of them in charge of controllable number of workers, the increase of bandwidth consumption that brought by DRAPS is reasonable.

Next, we conduct the same experiments with 40% more containers to test the scalability of DRAPS. Fig 9 plots the system performance with 140 Docker containers. Comparing the figures, the first impression is that on Fig 9a, the usages suddenly drop from 95.2% to 11.1% for memory and 100% to 0 for CPU. The reason lies in the fact that, at time 726, the memory becomes the bottleneck on work 1 with Spread scheme. However, the manager does not award this situation on worker 1, and assign a new container to it. Worker 1 fails to start the new container, and drains the memory, which results in the death of all containers on it. The the Docker engine



(a) Worker 1 (Spread, 140)

(b) Worker 2 (Spread, 140)

(c) Worker 3 (Spread, 140)

(d) Worker 1 (DRAPS, 140)

(e) Worker 2 (DRAPS, 140)

(f) Worker 3 (DRAPS, 140)

Fig. 9: Memory and CPU resources usage comparison between Spread and DRAPS placement scheme (140 containers)

decides to kill them all when it can not communicate with them. On the other hand, DRAPS considers dynamic resources usages on workers, and it stops assigning task to a worker if it has already overwhelming. It is shown on Fig 9d that the usages of memory and CPU remains at 46.3% and 18.8% for worker 1 with DRAPS. While worker 2 with Spread still runs smoothly at the end of the testing, its memory usage is at a high level, 76.6%, comparing to work 2 with DRAPS the same value is 54.1%.

2) Loaded containers: Besides idle containers, we set up a mix environment that includes both idle and loaded containers. If loaded container is running and there are clients using the services, we call it a loaded container.

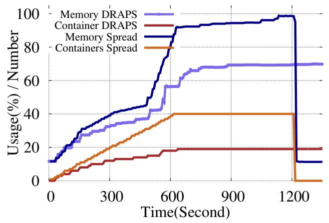


Fig. 10: Comparison of memory usage and number of containers

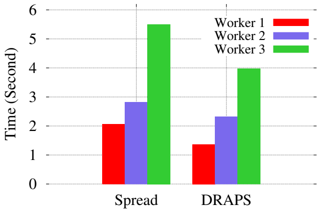


Fig. 11: Value of IOWait on each worker node

VIII. Conclusion

References

[1] Amazon web service. https://aws.amazon.com/.

[2] Microsoft azure. https://azure.microsoft.com/en-us/.

[3] Virtual machine. https://en.wikipedia.org/wiki/Information_explosion.

[4] R Jithin and Priya Chandran. Virtual machine isolation. *Communications in Computer and Information Science*, 420:91–102, 2014.

[5] Violeta Medina and Juan Manuel García. A survey of migration mechanisms of virtual machines. *ACM Computing Surveys (CSUR)*, 46(3):30, 2014.

[6] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, 102(1):11–31, 2014.

[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[8] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *FAST*, volume 16, pages 181–195, 2016.

[9] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. Comicon: A co-operative management system for docker container images. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, pages 116–126. IEEE, 2017.

[10] Aditya Hegde, Rahul Ghosh, Tridib Mukherjee, and Varun Sharma. Scope: A decision system for large scale container provisioning management. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*, pages 220–227. IEEE, 2016.

[11] Docker swarmkit. https://github.com/docker/swarmkit.

[12] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[13] IC Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. Usenix, 2016.

[14] Chanwit Kaewkasi and Kornrathak Chuenmuneewong. Improvement of container scheduling for docker using ant colony optimization. In *Knowledge and Smart Technology (KST), 2017 9th International Conference on*, pages 254–259. IEEE, 2017.

[15] N. Bansal, A. Caprara, and M. Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 697–708, Oct 2006.

[16] Adam Meyerson, Alan Roytman, and Brian Tagiku. *Online Multidimensional Load Balancing*, pages 287–302. Berlin, Heidelberg, 2013.

[17] Sungjin Im, Nathaniel Kell, Janardhan Kulkarni, and Debmalya Panigrahi. Tight bounds for online vector scheduling. In *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, FOCS '15, pages 525–544, Washington, DC, USA, 2015.

[18] Nsf cloudlab. https://cloudlab.us/.

[19] Alibaba cloud. https://www.alibabacloud.com/.

[20] Docker hub. https://hub.docker.com/.