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

Ying Mao*, Jenna Oak*, Anthony Pompili*, Daniel Beer*, Tao Han† Peizhao Hu‡
*Department of Computer Science, The College of New Jersey, Email: {maoy, oakj1, pompila1, beerd1}@tcnj.edu
†Department of Electrical & Computer Engineering, University of North Carolina at Charlotte, Email: tao.han@uncc.edu
‡Department of Computer Science, Rochester Institute of Technology, Email: ph@cs.rit.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 are powered by various virtual machines improve the quality of services. Although virtual machines are well-isolated among each other, they suffer from redundant boot volumes and slow provisioning time. To address limitations, containers were 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, concerning 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 target on investigating the Docker container cluster and developed, DRAPS, a resource-aware placement scheme to boost the system performance in a heterogeneous cluster.**

I. Introduction

This sentence is too informal and awkward. In the past few decades, we have witnessed a spectacular information explosion over the Internet. Hundreds of thousands of users are consuming [the internet isn't consumed...] 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-the-art infrastructures on the cloud, such as Amazon Web Service [**?**] and Microsoft Azure [**?**]. Focusing on providing the services at scale, virtualization is one of the emerging [is it really emerging? How old is it? If containerization is yesterday's news then isn't virtualization last year's news?] technologies used in data centers and cloud environments to improve both hardware and development efficiency.

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

To address these limitations, containers were 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 [reference chroot and others] and modern big data processing platforms~~plforms~~ utilize containers as a basic computing unit [**?**], [**?**], [**?**]. 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, [what does 'in the system view' mean here?] that enjoy a virtualized resource environment, not only 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 "Docker run" commands from clients, the cluster, as the first step, should select a physical machine to host those containers. [so on which machine is the user calling `docker run`? I am used to standalone docker] 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 to equally distribute 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 dominant 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 introduce the concept of dominant resource type that considers the dynamic demands from different services. [Are you really introducing it here? Hasnt this been done?]
- 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. [clarify what a type is] The evaluation demonstrates that DRAPS outperforms the default Spread and reduces usage as much as 42.6% on one specific node. [how cherry picked is this?]

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 [**?**]. More recently, containerization, a lightweight virtualization technique, is drawing increasing popularity from different aspects and on different platfroms [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**].

The benefits and challenges of containerized systems have been studied in many aspects. A comprehensive performance study is presented in [**?**], where it explores the traditional virtual machine deployments, and contrast them with the use of Linux containers. The evaluation focuses on overheads and experiments that show containers' resulting performance to be equal or superior to VMs performances. Although containers outperform VMs, the research [**?**] 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 which can significantly reduce the startup latency. While Slacker reduces the amount of copying and transferring packages, if the image is locally available, the startup could be even faster. CoMICon [**?**] addresses the problem by sharing the image in a cooperative manner. From different aspect, SCoPe [**?**] tries to manage the provisioning time for large scale containers. It presents a statistical model, used to guide provisioning strategy, to characterize the provisioning time in terms of system features.

Besides the investigations on standalone containers, the cluster of containers is another important aspect in this field. Docker Swarmkit [**?**] and Google Kubernetes [**?**] are dominant cluster management tools in the market. The authors of [**?**], first, conduct a comparison study of scalabilities under both of them. Then, firmament is proposed [it will need to be clarified that firmament is the name of a technology] 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 [**?**] describes an Ant Colony Optimization algorithm for a cluster of Docker containers. However, the algorithm does not distinguish various containers, which usually have a divese requirements. [more on how this is a disadvantage??]

In this paper, we investigate the container orchestration in the prospective of resource awareness. While users can set limits on resources [how?], containers are still competing for 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.

III. Background and Motivation

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`~~"docker run -it ubuntu bash"~~. Not sure how IEEE format wants in-line code examples. Further, what journal are we aiming for, and how do they want things? I should look at their templates. A Docker container image is a lightweight, stand-alone, 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 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 many applications with various services in a standalone mode.

B. Container Orchestration

When deploying 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 are running SwarmKit, worker nodes, and manager nodes. Worker nodes are responsible for running tasks; on the other hand, manager nodes accept specifications from the user and are responsible for reconciling the desired state with the actual cluster state. [we need to say abit more about what that means. What about the state could be undesired vs desired?]

[What is the point of this paragraph? We just jump in saying we can set limits at init time. (Note that we can also update them with `docker update ...`). Is this even relevant for this paper, since we are talking about container *placement* which is a cluster-level, rather than container-level, decision process] 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. [We may need a more clear example of what a 'task' is] Specifically, it utilizes filters and scheduling strategies to assign tasks. [What is a filter? Would someone in this field know that or do we need to define? I certainly don't know.] There are four filters 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 that pass the filtering process, SwarmKit supports three scheduling strategies: spread (currently available), binpack, and random (under development based on Swarm Mode). [Is this still the case? I know its been a while since this project was last touched.] *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.

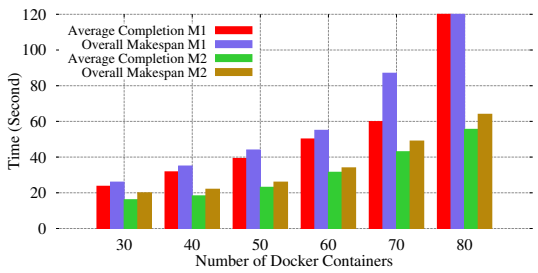


Fig. 1: Starting Dockers on a single machine

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 results in different experiences. Fig 1 plots the average starting delay of and overall makespan of the 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 machine, M1 or M2, we can see that the more containers it hosts, the larger the starting delay and makespan. However, M1 costs 23.67s on 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. [So, is the point here that the nodes arent 'heterogenous' and that #containers may not be a good way to decide which node is ready to handle a new task? That seems reasonable to me. We should say so explicitly]

IV. DRAPS System

A. Framework of Manager and Worker Nodes

As described in the previous section, there are multiple managers and workers in the system. [We weren't super clear about what it would mean to have *multiple managers*. I.e. is one of them just a backup in caise the main manager goes down?] 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 sends back their states to managers through periodical heartbeat messages. An **executor** is used to run the tasks that are assigned to the containers in this worker. [Do we need more on this? Its certainly not enough for *me* to fully understand how Swarmkit works before DRAPS is added...]

B. DRAPS modules

To simplify the implementation, we integrate the DRAPS components into the current framework. As shown on Fig 2, it ~~mainly~~ consists of three parts: a container monitor that resides in the worker nodes, a worker monitor, and a DRAPS scheduler thatare implemented ~~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 [is it applicaiton-level or container-level?], the monitored resources contain memory, CPU percentage, block I/O, and network I/O. The average usage report in a given time window [This is our first mention of time windows.] 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 its own bottleneck [We havent defined a bottleneck. Does that get done in a later section? Is that conventional?]. If a bottleneck is found, a 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 [how], it will generate tasks [for whom?], 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 [why are we talking about duplicated containers?] Docker container, DRAP-Scheduler checks its characteristics on resource consumption, such as memory intensity, through the records of the previous containers in the same services.

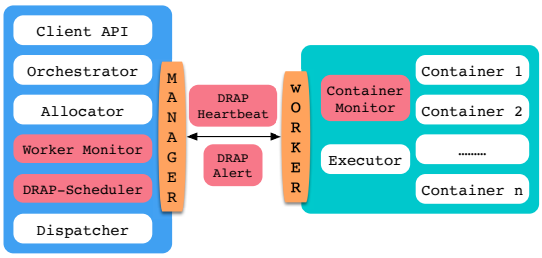


Fig. 2: Docker Framework with DRAPS Implemention

V. Problem Formulation

The DRAPS 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 multiple resources such as memory, CPU, bandwidth, and I/O for running its services. Since the services and their workloads in a container change 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. 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. [Is this DRF?] Then, the utilization ratio of the jth work node is $\max_{k \in \mathcal{K}} u_j^k(t)$. 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 DRAPS scheduler is to maximize the available resources in each worker node, the DRAPS 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. [This feels lemma dropped in, should we introduce it?]

Lemma 1. *The DRAPS scheduling problem is NP-hard~~ness~~.*

Proof. In proving the Lemma, we consider a simple case of the DRAPS scheduling problem in which the resource requirements of each container are constant over time. The simplified DRAPS scheduling problem equals to the multidimensional bin packing problem which is NP-hard [**?**], [**?**], [**?**]. Hence, the lemma can be proved by reducing any instance of the multidimensional bin packing to the simplified DRAPS scheduling problem. For the sake of simplicity, we omit the detail proof in the paper. [Should the full paper include the proof? It doesn't seem terribly difficult.] □

VI. DRAPS in a Heterogeneous Cluster

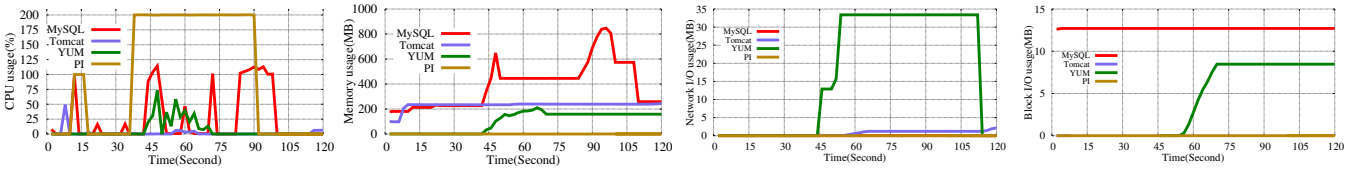
Previously, we discussed the different modules in DRAPS and their major responsibilities. We also formulated the DRAPS scheduling problem and proved that the problem is NP-hard. In this section, we present the detailed design of DRAPS with heuristic container placement and migration algorithms, in a heterogeneous cluster, which aims to increase resource availability on each worker node and boost the service performance by approximating the optimal solution of the DRAPS scheduling problem. To achieve the objectives, DRAPS system consists of three strategies: 1) Identify dominant resource demands of containers; 2) Initial container placement; 3) Migrate a container

[This is good: very clear.] point 1 is really how the system percieves its environment, 2 and 3 are the actions it can take.

A. Identify Resource Demands from Containers

Before improving the overall resource efficiency, the system needs to understand the dynamic resource demands 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 diverse resource demands. As an intuitive example, we conducted the experiments on NSF Cloudlab [**?**] (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. [do you think this is a little too easy?]
4) PI: a service to calculate PI. Tested workload: top 3,000 digits with single thread, top 7,500 digits with two threads. [What is PI?. Do you mean π ?]]



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

Fig. 3: Resource demonds under different workloads on four services, MySQL, Tomcat, YUM, PI.

Figs. 3a to 3d 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 [make sure to reference fig 1a explicitly]. The usages of the other three types of resources still remain at very low levels. For MySQL service container, with tested operations, the CPU usage shows a burst when clients submit a request [a request to who? is this a lead in to the operations? I think this discussion is too flowing and needs to be more organized.]. At time 84, a "join" operation that involves 3 tables is submitted, and we can find CPU usage jumps, as well as memory usage. This is because the join operation needs a lot 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 to the size of a tested HelloWorld page, but more than 200MB of 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, there is a dominant resource demand of a service in a given period despite multiple types of resources.

In DRAPS , we need to identify the dominant resource demand for each service. [Is a reasonable expectation in the application setting that new containers will be of known service type? If so does it imply that we know the resource profile of new containers based on the profiles of the old ones?] A manager, in the system, can monitor all of the containers' resource usage and group them by their associated service ID. [So can we let $\mathcal{S}$ be the set of services, and in this case they would be {MySQL, TomCat, YUM, and PI}??] Suppose the service $s_i \in S$ is associated with ~~contains~~ m running containers that store in a set, RC_{s_i}. [Or let $\mathcal{C}_{s_i} = \{c_j \in \mathcal{C}$ if $c_j$is running$s_i\}$] 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},

$$R_{s_i} = \sum\nolimits_{c_i \in RC_{s_i}} R_{c_i}$$
$$= \left\langle \sum_{c_i \in RC_{s_i}} r_1/m, \sum_{c_i \in RC_{s_i}} r_2/m, \ldots, \sum_{c_i \in RC_{s_i}} r_q/m \right\rangle$$

[Note: I fixed the size of the angle braces above]

[Also: this formula is too complicated and can be simplified by linearity as follows:]

$$R_{s_i} = \frac{1}{m} \sum_{c_i \in RC_{s_i}} < r_1, \ldots, r_q >$$

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 from the sum of vectors from workers. Therefore, R_{s_i} can be represented by a percentage of the total resources in the system, for the i^{th} type, the container cost for s_i in on average is $\sum_{c_i \in RC_{s_i}} r_i/m \div L_i$. With the analysis, we define the dominant function, [I think we have some redundant indices here, we cant just use i for everything: should be c_j.] $DOM(s_i) = max\{\sum_{c_i \in RC_{s_i}} r_i/m \div L_i\}$ Function $DOM(s_i)$ returns the type of a dominant resource demand of service s_i within a given time period. [Wouldn't that be $\arg\max$? This just returns the fraction used by the dominant resource] The value of $DOM(s_i)$ changes along with the system depending on the running containers for s_i and the current cost of them.

B. Initial Container Placement

To use a SwarmKit cluster, clients need to execute the command "docker run" 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 also fails to consider node heterogeneity, yes?]. This is because the managers in SwarmKit do not have a mechanism that can monitor the current available resource. DRAPS, on the other hand, addresses the problem by introducing DRAPS-Heartbeat. DRAPS-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. On the manager side, the data will be organized into a table that keeps tracking the current available resource on each worker and its corresponding containers' resource usages.

Running on managers, Algorithm 1 assigns a container initialization 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 new container starting task arrives, the algorithm applies all filters that the 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, the S_{dom} parameter will be used to store the container's dominant resource attribute (line 8). In DRAPS, we consider four types, memory, CPU, bandwidth, and block i/o. The W_{cand} set will be sorted according to the dominant resource attribute and return the W_{id} with the highest available resource in S_{dom} type (line 9-10). If the service cannot be found in $\{KS\}$, W_{id} with the highest available resource on average will be chosen (line 11-13).

Algorithm 1 Container Placement on Managers

1: Maintains a known characteristics service set $\{KS\}$
2: $\{W_{cand}\}$ = All running W_{id};
3: **Function ContainerPlacement**(S_{ID})
4: **for** $w_{id} \in \{W_{cand}\}$ **do**
5: **if** ! $Filters(w_{id})$ **then**
6: Remove w_{id} from $\{W_{cand}\}$
7: **if** $S_{ID} \in \{KS\}$ **then**
8: $S_{DOM} = DOM(S_{ID})$
9: Sort W_{cand} according to $r_{S_{DOM}}$
10: Return w_{id} with highest $r_{S_{DOM}}$
11: **else**
12: Sort $\sum_{i=0}^{i=q} r_i/m$ for $w_{id} \in W_{cand}$
13: Return w_{id} with highest average available resource

C. Migrating a Container

In a Swarmkit cluster, resource contention happens on every worker. The container conitor, which is a module of DRAPS, runs on each worker to record resource usages of hosting containers. In addition, the worker keeps tracking available resources on itself. Whenever it finds a draining type of resources becomes a bottleneck, [I think we should be more clear *in the text* about what a bottleneck is] it sends to managers a DRAPS alert message that contains the bottleneck type and the most costly container of this type. Upon receiving the DRAPS alert message, the manager needs to migrate this container to an appropriate worker and kill it on the worker to release the resources. [This obviously assumes that killing a container won't have any negative side effects. Which is true in most containerized settings, yes?]

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). [I'm not sure if this is how we should use the $\forall$ or the $\rightarrow$ operators] If S_{id} is a global service and it is in the known service set, $\{KS\}$, the algorithm returns w_{id} that is included in W_{cand}, with the highest available $r_{S_{DOM}}$. On the other hand, it returns w_{id} with the highest available r_i if S_{id} is not in $\{KS\}$ and S_{DOM} on unknown (lines 6-12). When S_{id} is not a global service [What the hell is 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} contains all running workers that do not hosting any containers for S_{id} (lines 13 - 23).

Algorithm 2 Process DRAPS 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** $\forall w_{id} \in W_{cand} \rightarrow S_{id} \in w_{id}$ **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

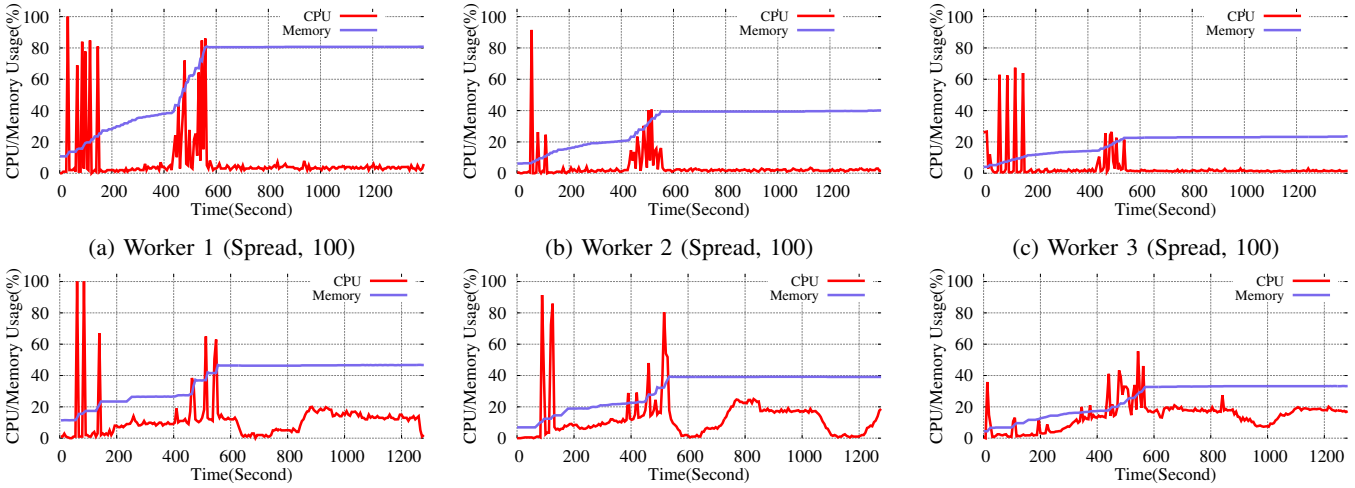
VII. Performance Evaluation

A. Implementation, Testbed and Workloads

We implement our new container placement scheme, DRAPS, on Docker Community Edition (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 [**?**], 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[so again, are we talking about heterogeny *of node type* in general?], and configure it with 1 manager and 3 worker (1 of the 3 physical nodes hosts both manager and worker). In experiments on scalability, we configure the cluster with 1 manger and 9 workers that consist 3 instances of each type.

The main objective of DRAPS is to understand resource demands of services and place them on appropriate worker



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

nodes. As we discussed in Section VI-A, characteristics of services are varied. Therefore, workloads for the cluster are images of various services. In the evaluation, we select 18 different images in 4 types from Docker Hub [**?**] to build our image pool. **Database Services**: MongoDB, MySQL, Postgres, Cassandra, RethinkDB. **Storage/Caching Services**: Registry, Memcached. **Web Services**: Tomcat, Httpd, Redis, HAProxy, Jetty, Nginx, GlassFish. **Message Services**: RabbitMQ, Apache ZooKeeper, ActiveMQ, Ghost.

B. Evaluation Results

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 serve any clients, we call it a idle container. Idle container is an important concept since every node, right after initialization will act as an idle container. Understanding the resource demands of an idle container will help us select its host. 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 4 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 from 0 to 500s. This is caused by submission pattern that used to initiate containers. The percentage grows continuously from 0 to 500s since we have 100 containers and the submission interval is 5 seconds. While in both systems, the usage of CPU stays at a low level on 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, because some supporting programs, such as the 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 5 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 power than others and hosts more containers than worker 1, which has limited resource.

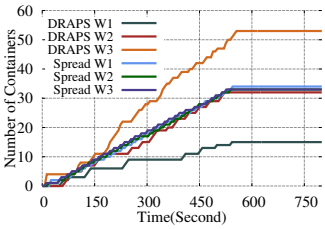


Fig. 5: Number of containers on each worker

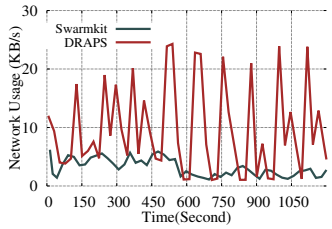


Fig. 6: Network consumption comparison on worker 3

While DRAPS achieves better performance, it introduces more data transfers between managers and workers through heartbeat messages. Fig 6 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

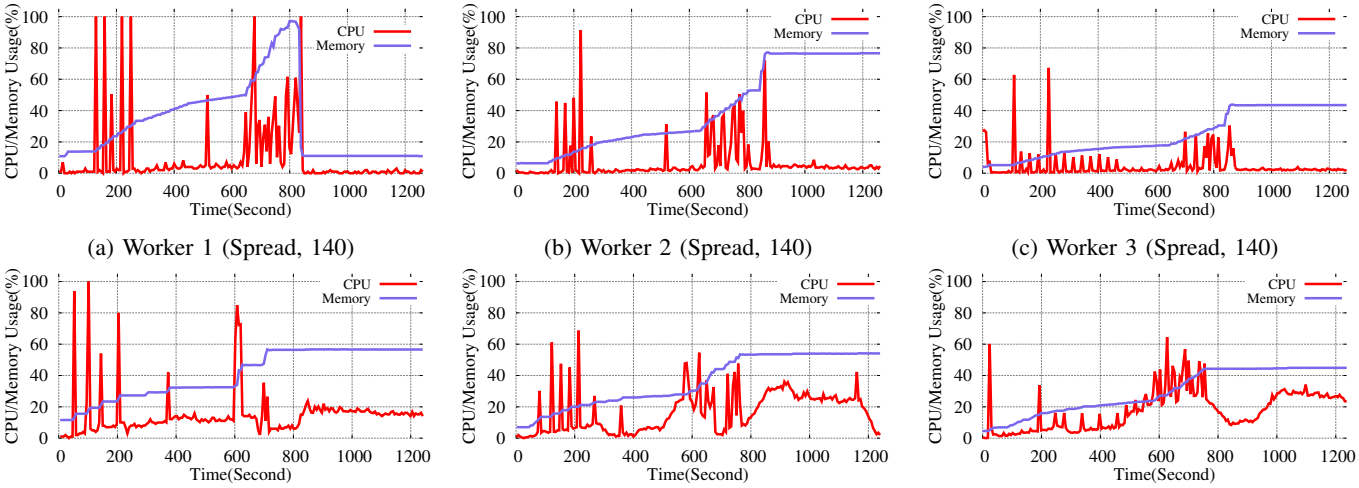
messages includes more statistical information resource usages of containers. Considering the distributed architecture, the system can have multiple managers and each of them in charge of controllable number of workers, the increase of bandwidth consumption that brought by DRAPS is reasonable.

Next, we conduct the same experiments with 40% more containers to test the scalability of DRAPS. Fig 7 plots the system performance with 140 Docker containers. Comparing the figures, the first impression is that on Fig 7a, 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 bottlenecked 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 Docker engine 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 7d 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 value is 54.1%.

2) Loaded containers: Besides idle containers, we set up a mix environment that includes both idle and loaded containers. If clients are generating workloads to the services on the running containers, we call it loaded containers. Evidenced by Fig. 3, we know that loaded containers consume more resources than idle ones. In addition, the usage pattern of a loaded container changes along with the workload. Fig 8 plots the memory usage and number of containers on Worker-1. For the experiments running with Spread, it drains the memory at time 825s that the memory usage drops from 98.5% to 11.9%. Simultaneously, the number of running containers on worker-1 drops from 44 to 9 and then, to 0 at time 825s and 837s. This is because the docker engine kills all containers when the memory is not enough to maintain the system itself. Due to less containers on worker-1 with DRAPS (44 v.s 24), it runs normally throughout the entire experiments. Fig 9 shows the value of I/O wait in percentage, which measures the percent of time the CPU is idle, but waiting for an I/O to complete. It shows a similar trend that at time 849s the value drops to 0 for Spread, while DRAPS maintains stable performance.

VIII. Conclusion

This paper studies the container placement strategy in a heterogeneous cluster. We target on distributing containers to the worker nodes with the best available resources. In this paper, we develop DRAPS, which considers various resource demands from containers and current available resources on each node. We implemented DRAPS on Docker Swarmkit platform and conducted extensive experiments. The results show a significant improvement on the system stability and scalability when comparing with the default Spread strategy.



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

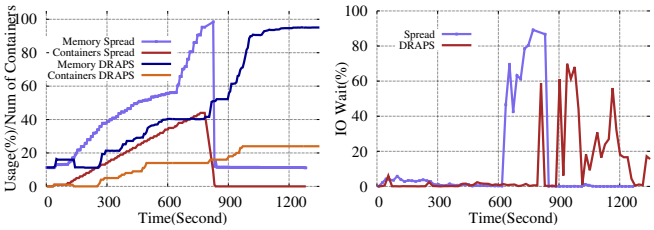


Fig. 8: Memory usage and container number on worker1

Fig. 9: Value of I/O Wait on worker1