Container Orchestration for Dispersed Computing

Pradipta Ghosh
University of Southern California
Los Angeles, California
pradiptg@usc.edu

Quynh Nguyen
University of Southern California
Los Angeles, California
quynhngu@usc.edu

Bhaskar Krishnamachari
University of Southern California
Los Angeles, California
bkrishna@usc.edu

ABSTRACT

In the era of Internet of Things, there is an increasing demand for networked computing to support the requirements of time-constrained, compute-intensive distributed applications. We present a container orchestration architecture for dispersed computing, and its implementation in an open source software called Jupiter. The system automates the distribution of computational tasks for complex computational applications described as an Directed Acyclic Graph (DAG) to efficiently distribute the tasks among a set of networked compute nodes and orchestrates the execution of the DAG thereafter. This Kubernetes based container-orchestration system supports both centralized and decentralized scheduling algorithms for optimally mapping the tasks based on information from a range of profilers: network profilers, resource profilers, and execution time profilers.

CCS CONCEPTS

• **Software and its engineering** → *Virtual machines*; • **Computing methodologies** → Self-organization; • **Computer systems organization** → Cloud computing.

KEYWORDS

Container Orchestration, Kubernetes, Dispersed Computing

ACM Reference Format:
Pradipta Ghosh, Quynh Nguyen, and Bhaskar Krishnamachari. 2019. Container Orchestration for Dispersed Computing. In *5th International Workshop on Container Technologies and Container Clouds (WOC '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366615.3368354

1 INTRODUCTION

Virtualization is a key component of modern data centers to support a large number of simultaneous user, multiple operating systems, and applications on a single computer or server while maintaining quality of service, privacy and security guarantee [15]. The key enabling technology is Virtual Machine which emulates a real computer system to the users with multiple of them running in the same physical machine. However, to support such user-to-user separation, virtual machines are 'bulky' i.e., one can not run more than 3-4 simultaneous virtual machine in a standard server. This often results in under-utilization of resources. Besides, virtual machines restrict access to physical interfaces and thus restricts the spectrum of supported applications.

What is a Container? Over the past decade, there has been a dramatic shift in virtualization technology with the introduction of Containers such as Docker containers [8]. A container [1] is *"a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings."* Therefore the operation principle of containers are different than Virtual machine. Each container is considered to the physical machine as a single process with access to the physical interfaces. Due to the lightweight nature of this virtualization, one can deploy thousands of containers in one server while keeping app-to-app separation.

Why are Containers important for Internet of Things (IoT)? The ubiquitous presence of economical low-compute-power edge devices such as cell phones, car dashboard, and drones has opened up the domain of edge or fog computing [7]. Edge Computing focuses on exploiting all the devices near end users to comply with the skyrocketing demand for computationally intensive applications such as image processing and voice recognition towards autonomy and personalized assistance. Most of such IoT devices lack the resource to accommodate virtual machines (VM). Thus, containers are the default choice of virtualization to support multiple users.

Dispersed Computing. Interestingly, a significant subset of these cutting-edge time-constrained, compute-intensive distributed applications rely on an orderly processing of the streaming data which can be represented in form of a Directed Acyclic Graph (DAG). This brings us to the newly emerging field of *Dispersed Computing* that focuses on a joint optimization of computation and communication costs to distribute the execution of a Directed Acyclic Graph (DAG) based task graph among a network of compute nodes that may be geographically distributed. Dispersed Computing can be thought of as a mixed architecture between Edge Computing and Cloud Computing where the network of compute nodes might contain either or both edge processors and cloud-based processors.

Objective. The main question is *how can we deploy and manage containers efficiently in a dispersed computing environment?* Researchers have developed a wide range of container orchestration tools such

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WOC '19, December 9–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7033-2/19/12...$15.00
https://doi.org/10.1145/3366615.3368354

as Kubernetes [6], Mesos [9], Docker Swarm [5] for normal task graphs in the context of clouds. To our knowledge, none of these tools has provisions for containerized pipelined processing of DAG-based task graphs for dispersed computing. In the grid computing domain [3], there exist some frameworks for mapping tasks on geographically distributed clusters such as Pegasus [4] and Falcon [11]. However, they assume a static and relatively well-characterized network with simple applications for centralized task mapping. In contrast, Dispersed Computing deals with networks of compute devices where the communication links are not well-characterized and dynamic.

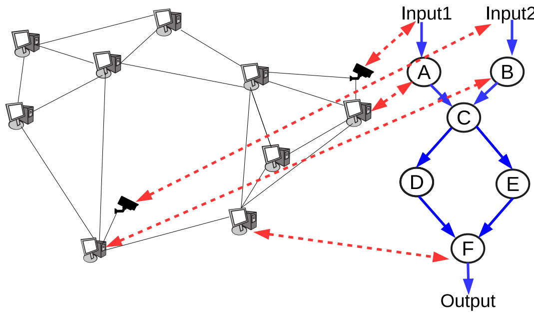

Figure 1: Illustration of the DAG based Networked Computing problem: the black lines denote communication links, the red lines denote the mapping, and the blue lines denote data flows.

Our Contribution. We present a new system architecture for containerized orchestration of DAG-based dispersed computing application task graphs, and its open-source implementation in a software system called Jupiter. Jupiter is built on top of the well-known container orchestration tool called Kubernetes [6]. Carefully leveraging the available architecture of Kubernetes, we introduce three new types of DAG orchestration containers to run on each device to support pipelined processing: *Profiler Container, Mapper Container, and Execution Container.* The Profiler Containers generates the necessary statistics required by a DAG scheduling/mapping algorithm such as the well known Heterogeneous Earliest Finish Time (HEFT) [14]. The Mapper Container contains the scheduling/mapping algorithm code and performs the mapping based on the profiler data. Lastly, the Execution Container contains the application code files and oversees the implementation of the mapping algorithm outcome. We have deployed the Jupiter system on a large cluster of more than 100 nodes in Digital Ocean to quantify the performance and overhead associated with various components of containerized dispersed computing (see §4).

2 PROBLEM DESCRIPTION AND KEY OBJECTIVES

First, let us assume that we have a network of N heterogeneous networked compute processors (NCPs) that are geographically distributed. Let's say we are interested in deploying an application-DAG that consists of T tasks where the input sources are distributed geographically. Now, the goal is to properly map the tasks from the application DAG to the NCPs such that the Makespan of the application-DAG is minimized[1]. The Makespan in this context would depend on both the compute powers of the NCPs chosen as well as the delays on the network paths between the NCPs. For an illustration, refer to Figure 1 where the application DAG consists of 6 tasks with two geographically separated input sources.

Scheduling of DAG. The scheduling goal is to optimally map the tasks on the geographically distributed NCPs such that the output can be made available the fastest. To support scheduling of a task graph in the form of a DAG, i.e, pipelined executions, we need special types of scheduler that are not available in most well-known container orchestrators such as Kubernetes. However, in this paper we are not focused on developing a scheduling algorithm rather to build a system to support such algorithms. Such mapping algorithms will require a wide range of statistics about the devices such as end to end file transfer latencies which are not readily available in existing systems.

Execution of DAG based Pipelined Task. Dispersed computing task graphs typically follow a DAG task graph with pipeline execution i.e., the output stream from each task feeds the input of the subsequent task in the DAG. In current context, it is the job of the container developer to put the right code to support such pipelined execution. Thus, a generic application independent system is missing in current container orchestration tools.

3 PROPOSED SOLUTION

3.1 Background

Before detailing our software architecture, let us provide a brief summary of Containers and the Kubernetes architecture as our proposed architecture rely heavily on them.

Containers, Dockers. The most common norm in cloud computing today is to use Virtual Machines (VMs) to support the demand of users while keeping necessary isolation between the user processes running on the same physical machine. Due to high overhead, the number of concurrent VMs on a Physical machine is restricted to a very small number such as 3-5 for a standard Quadcore desktop with 12GB RAM. Moreover, due to the lack of direct access to the Hardware, the functionality of VMs are restricted. Containers, on the other hand, are the most cutting-edge convention for processor virtualization that provides isolations similar to traditional VMs but with much less computing power requirements. Unlike VMs, a container image is a lightweight standalone executable that includes all the requires modules, libraries, codes, and tools to run it. A container directly runs on the guest OS and is considered as a single process by the guest OS. All the processes inside a container are viewed as a sub-process of the main process. Because of this low computation requirement, one can run hundreds of containers on a physical machine. The concept of a container has been around for a while [2] but became particularly popular with the advent of a specific type of containers called Docker containers [8] in 2014.

Kubernetes. Like Virtual Machines, Containers also need proper orchestration and management in a Cloud Data Center. While there exist different alternatives, Kubernetes [6], originally designed by

[1]Other objectives are also possible, but we will restrict ourselves to makespan minimization for ease of exposition.

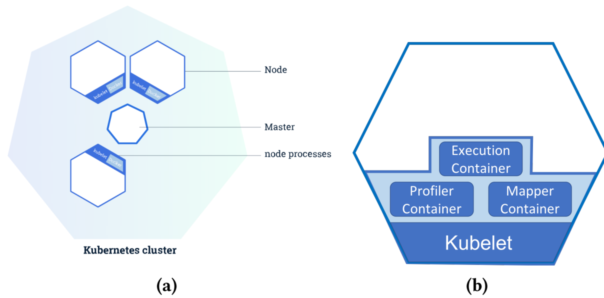

Figure 2: (a) Kubernetes Cluster Layout (Taken from https://kubernetes.io/), (b) Each Kubernetes Node with DAG Orchestation

Google, now maintained by the Cloud Native Computing Foundation, is one of the most popular and widely accepted ones.

Any Kubernetes Cluster follows a well-defined layout (presented in Figure 2a) with two main types of entities: Kubernetes Master and Kubernetes Nodes.

Kubernetes Master. Kubernetes Master is the management node of the cluster which oversees proper functioning of every nodes in the cluster. The Master is in charge of coordinating all activities including applications scheduling, scaling, and supporting fault tolerances.

Kubernetes Nodes. Kubernetes Nodes are the worker nodes in the cluster which can be physical machine or a Virtual Machine (VM). Each Kubernetes Node is manages by the Kubernetes Master by running some management related containers in each machine called Kubelet.

To deploy a container, the user needs to issue the proper command at the Kubernetes Master that will decide where to spawn the container, download/build the container on the desired Kubernetes Node, and finally queue the spawning of the container. Note that queuing a spawn does not guarantee the spawning. Each Kubernetes Node has its own queue and resources. If enough resource is available, it will take out an entry from the queue and spawn the container. A detailed overview of Kuberenetes can be found in the official website https://kubernetes.io/.

3.2 DAG Orchestration Architecture

The original Kubernetes does not support automatic scheduling or execution of a DAG based task graph as required for dispersed computing. To this end, we have developed a new architecture that builds on top of Kubernetes (Figure 2b). To have a minimalistic design of the architecture, we have developed three types of specialized container that can fulfil our goal once each device runs a copy of them. To Kubernetes these containers are standard containers that can be deployed from the Kubernetes master using standard 'kubectl' command or the Kubernetes python API. This modular design also allows us to easily port this architecture to other orchestration tools like Mesos or Docker Swarm (We have not done this). Next, we detail each of these specialized containers.

Profiler Containers. To schedule DAG on a set of networked compute nodes (NCP), we mainly need two basic information: (1) resource availability in each node, and (2) inter node file transfer latency. While resource availability is available from Kubelets running on each node, file transfer latency information is not readily available.

Concept: To gather the necessary statistics, we have developed a special type of Containers called *Profiler Containers*, that will periodically probe the links between the nodes to generate the necessary statistics. We use Kubernetes to run a copy of this container is each node. We take advantage of the fact that the NICs are not virtualized in Containers i.e., each container can access the real network hardware.

Function Logic: Each profiler container periodically sends a randomly generated file with known file size to each of the other NCPs via a well-known file transfer protocol such as Secure Copy (SCP). The file transfer times are recorded and curve-fit using a quadratic regression with respect to the file-size (f) as: $l = p + q \cdot f + r \cdot f^2$ where p, q, r are empirically determined constants. We opted for a quadratic fit as it is the empirical best fit towards approximate file transfer time for varying file sizes [10]. For completeness, we have also included a compute resource profiler inside the profiler container that periodically checks the processor usage. Moreover, to have a Kubernetes like architecture, we kept a notion of home profiler and worker profiler where 'home' refers to a master profiler that can coordinate (if needed) the profiling tasks on individual 'worker' NCPs.

Networking Requirement: One key requirement for such profiler containers is that each container needs to be uniquely addressable as if they represent the physical / virtual node. We leverage the service abstaction available in Kubernetes to this end where each profiler container is assigned an individual service and thus an unique IP.

API Requirement: To utilize the profiler data for scheduling purpose, one must be able to query the profilers to get the latest statistics. To this end, we have associated a Flask server with unique Port Number (referred as *'PROFILER-FLASK-PORT'*) and associated a Function call *local_profiler_stat*. Any device/container in the network can now directly query individual profiler containers to get its statistics.

Mapping Containers. Since the scheduling of DAG based task graph require us to follow different logic than the existing Kubernetes scheduling algorithm, similar to the profiler containers, we have developed a special type of container for scheduling called *"Mapping Containers"*.

Concept: A Mapping Container can contain one/multiple scheduling algorithms based on the requirements. For a centralized scheduling algorithm there will be only one instance of Mapping Container in the Cluster. However for a distributed scheduling algorithm, each NCP will run a copy of the Mapping Container. The code inside each Mapping Container first uses the Flask servers of the profiler containers to gather necessary stats. This information is used by the scheduling logic (such as the well-known HEFT algorithm [14]) to determine the mapping of tasks in the DAG to each NCP.

Function Logic: Each mapping container can communicate with each of the profiler as well as each of the other mapping container.

This is important for distributed scheduling. For centralized scheduling, the mapping container just query all the profiler containers to gather information about all nodes and then make more informed decision. For distributed scheduling, each mapping container only contacts the profiler container located in the same NCP. Once a mapping is available, it contact the respective execution containers (detailed below) to start the task. Over time if the mapping changes, it again communicates with the necessary execution containers to start/stop tasks. Similar to profilers, we have kept a notion of home mapping container and worker mapping container for distributed scheduling where 'home' refers to a master mapping container that can coordinate the mapping tasks on individual 'worker' NCPs.

Networking Requirement: As with profiler containers, each mapping containers are assigned unique IP address by wrapping them inside Kubernetes service abstractions. The IP address of all the services are fed to each container at boot time.

API Requirement: For lazy retrival of the mapping data as well as inter container communication in distributed scheduling, each mapping containter is also associated with a Flask server with unique Port Number (referred as *'MAPPER-FLASK-PORT'*).

Execution Containers The last type of specialized containers required for DAG processing is called *"Execution Containter"*.

Concept: In order to run a DAG task-graph on a set of NCP, the execution containers are required to perform two main tasks: (1) run the scheduled/mapped task on the selected node and (2) transfer the output of the executed tasks to the input of the next task of the DAG which may be running in a separate node. Each execution container are supplied with all the task files (e.g., a separate python script for each task). We have two different models for Execution Container deployment. We can run a copy of the execution container in each NCP. With this model, upon receiving a mapping from a mapping container, a background process running on the execution container starts a new thread with the respective task code. The second models is to deploy execution container only on the devices where a task has been mapped and re-deploy everything when a new mapping is available. The former model performs better for a very dynamic environment whereas the later is optimized for a cluster with a relatively static load. In this paper, we present only the second model for the analysis of the proposed system.

Function Logic: Each execution container has three main components: (a) a compute module that load the relevant task codes and starts a new thread once a mapping is available, (b) a monitor module that keeps track on input and output of the tasks, and (c) a transfer module which transfer the output of the task, once available, to the execution container of the next task in the DAG. Again, we have kept a notion of home and worker in Execution Containers where 'home' refers to a master Execution Containers that can coordinate the bootstrapping of the task execution.

Networking Requirement: As with profiler containers and mapping containers, each execution containers is assigned a unique IP address by wrapping them inside Kubernetes service abstractions. The IP address of all the services is fed to each container at boot time. This information is used both by the mapping containers to push the mapping information and the transfer module of the execution container to transfer the outputs.

API Requirement: For pushing the mapping information, each execution container is also associated with a Flask server with unique Port Number (referred as *'EXECUTION-FLASK-PORT'*).

3.3 The Jupiter System

In this section, we discuss briefly our open source software implementation of a framework to support DAG execution with containers on Kubernets that we refer to as the Jupiter System (available online at https://github.com/ANRGUSC/Jupiter). The Jupiter system consists of three main modules: *Profiler, Task Mapper, and the CIRCE dispatcher* which aligns with the three types of containers introduced earlier. The inputs to the Jupiter consist of the Directed Acyclic Task Graph (DAG) information, the task files, and the information (such as IP or node name) about available compute nodes.

Profiler. As explained before, we need to have some basic profiling information about the compute resource availability, runtime of tasks, and end-to-end file transfer latency. To this end Jupiter consists of two main different types of profilers (embedded in the Profiler Containers): (1) Network Profiler that maintains statistics about the bandwidth and end-to-end delay between the available NCPs, (2) Resource Profiler that profiles the resource availability of each NCP in terms of CPU and Memory availability. *For conciseness, we use the term "DRUPE" to refer to the combined network and resource profiler.*

In addition, we have a provision for a third type of profiler called the Execution profiler that profiles the execution time of each task of the DAG in each of the available NCPs. For optimal allocation of tasks, some task mappers such as HEFT[13] might need such information about the execution times of the individual tasks for each of the NCPs. HEFT like task mappers do not use the raw CPU usage statistics available from the Resource Profilers. Some scheduler require some base statistics about the execution time on each of the NCPs even before the tasks are actually mapped. However, the complete execution time information is available only after the tasks are executed on each of the available NCPs. To deal with this, we introduce a fourth type of container called 'Exec-Profiler Container' which are run on each NCP at the very beginning and torn down once a base statistics is available. An 'Exec-Profiler Container' runs the entire DAG with some sample input files and collects the statistics. The second fold of execution profiler data is available from the execution containers. Each execution containers includes a runtime-execution-profiler code which gathers timestamp information for all the tasks executed on a NCP over time. Available runtime statistics include Enter time, Execute time, Finish time, Elapse time, Duration time and Waiting time.

The information from the profilers and the input files are fed to the task mapper module which outputs a mapping of the DAG tasks into the available NCPs based on the mapping algorithm used.

Task Scheduler/Mapper. The job of the task scheduler on Jupiter architecture (contained inside the Mapping Container) is to properly map individual tasks from a DAG to the compute nodes such that the Makespan of the DAG is minimized. We define the *'Makespan'*

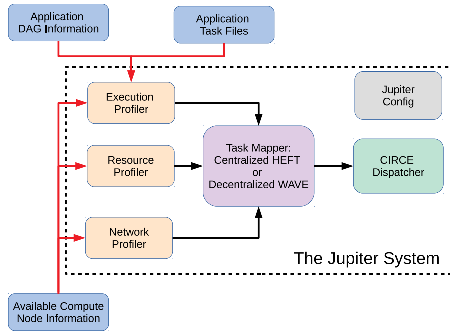

Figure 3: The Jupiter Architecture

of the DAG as the time required to generate an output via executing the entire task DAG on one set of input files or input chuck of data. In the current Jupiter system, we have provision for two different classes of task mappers: (1) Centralized HEFT[13] and (2) Decentralized WAVE [12], with a modular design that enables researchers to build and test out other task mappers. A Jupiter configuration file is used for choosing between these different options of task mappers as well as setting a range of parameters to customize for application-specific requirements. To provide more context on the mapper algorithm as well as provide some statistics on the Jupiter System, we briefly describe the HEFT algorithm below.

Heterogeneous Earliest Finish Time (HEFT): Heterogeneous Earliest Finish Time (HEFT) ([13, 14]) is a well-known heuristic in grid/cloud computing for mapping a directed acyclic task graph into a network of heterogeneous compute nodes that also accounts for the communication times between the nodes. HEFT operates in a sequence of two phases: ranking and prioritization, and processor selection. In the first phase, i.e., *ranking or prioritization* phase, HEFT defines a priority of each task t_i as follows:

$$rank_u(t_i) = \overline{\omega_i} + \max_{t_j \in succ(t_i)} (\overline{c_{i,j}} + rank_u(t_j)) \quad (1)$$

where the subscript "u" refers to "upwards rank" which is defined as the expected distance of the task from the end of the computation, t_i refers to task i, $\overline{\omega_i}$ is the average computation cost of the task i among all the compute nodes, $\overline{c_{i,j}}$ refers to the average communication cost of the data communicated between task t_i and t_j for all pairs of compute nodes, and $succ(t_i)$ refers to the set of dependent tasks in the DAG.

In the second phase i.e., the *processor selection* phase, HEFT assigns the tasks to the NCPs based on the ranks calculated in the *ranking or prioritization* phase. In each iteration of the task assignment, HEFT picks the task which has the highest priority and has all the dependent tasks already mapped. Next, HEFT schedules the task on an NCP that will minimize the earliest finish time of that task. This process continues until all the tasks are mapped. Finally, HEFT outputs the overall task to NCP mapping along with a timeline to follow for the executions.

Task Execution/Dispatch. The generated task-to-NCP-mapping is used by the CIRCE dispatcher module in our Jupiter system to dispatch the tasks on respective NCPs, monitor the input-output of each task to administer the respective task execution, and transfer the data/files between consecutive tasks of the DAG. Thus, CIRCE is more-or-less the system inside each of the execution container. We have provision for both lazy retrieval of map as well as active retrieval. Lazy retrieval implies we only change the task-to-NCP mapping if there is a dramatic change in the system and we redeploy all the tasks according to the new mapping. Active retrieval implies that a notion of 'task controller' periodically checks if there exists a better alternative for each task in the DAG. If any, the 'task controller' forces CIRCE to switch the task mapping immediately.

In Figure 3, we illustrate the software architecture of the proposed Jupiter system along with the data flow between different modules.

Automating Deployment. To automate the DAG deployment on Kubernetes, we have written a set of scripts to assist generating the Docker files for all the modules of the system based on specific applications and user configuration choices, to build the corresponding Docker images, to push those images to the Dockerhub, and to deploy and tear down the whole system in one shot on a cluster. The automatic scripts take advantage of the Python client for the Kubernetes API.

We also developed and open-sourced a set of scripts to automatically install the environment requirements for all the nodes in the cluster (Digital Ocean in our testing settings) including Kubernetes deployment for the master and the children nodes with various cluster parameters like number of nodes, number of CPU cores, memory size, limit of network and CPU resources by the Kubernetes cluster, and stress test simulation images for the NCPs.

Developing Applications. The current Jupiter system has been tested with applications developed in C and Python language which are made opensource for illustration. In addition, we have generated a set of standardized rules for writing applications with Jupiter. As long as a application code follow the rules, it can be automatically deployed with Jupiter. We also developed a dummy application generator customized for the Jupiter Orchestrator. The dummy application generator generates a DAG based on some input parameters such as depth (number of levels of the DAG), minimum width and maximum width (number of tasks in each level of the DAG), total number of tasks in the DAG, and communication to computation ratio (for a preference between computation-intensive application and Network - Delay focused application). The dummy application generator also generates the corresponding dummy application in Python with correct structure and syntax ready to be tested on Jupiter.

Integration interface. The Jupiter system provides flexibility in terms of network protocols (unicast or multicast capability, destination address or port, data transport method) and and resource profiling through a modular interface. There is a necessity to provide a transparent interface and take advantage of the Kubernetes services so that the users do not have to worry about the Kubernetes network policy and can be unaware of how the packet flow is handled behind the scenes.

4 EXPERIMENTS

In this section, we analyze runtime statistics of different components of the Jupiter system using the dummy application generator. To this, we use the dummy application generator to generate different applications with varying number of tasks ranging from 30 to 200 tasks. For these experiments, we use the well-known HEFT

Table 1: Latency (in seconds) over varying DAG size. N is number of nodes and M is number of tasks in the DAG.

Module	Time (sec)	N = 106, M = 30	N = 106, M = 50	N = 106, M = 100	N = 106, M = 150	N = 106, M = 200
DRUPE	Deployment	85.69				
	Teardown	221.36				
EP	Deployment	84.63	84.14	84.28	85.06	85.13
	Execution	449.30	549.80	789.33	1090.81	1294.12
	Teardown	183.71	205.77	211.24	229.51	318.24
HEFT	Makespan	50.53	82.82	105.34	130.02	148.18
	Deployment	1.09	0.96	0.96	1.14	1.02
	Mapping	142.53	147.30	177.18	190.73	203.14
	Mapping with EP	591.83	697.1	966.51	1281.54	1497.26
	Teardown	2.37	2.18	2.03	2.11	1.81
CIRCE	Deployment	46.09	58.96	124.90	158.89	232.19
	Teardown	22.72	42.10	73.58	115.21	206.29

algorithm for task scheduling/mapping and assume relatively static load in the cluster to deploy execution container only on the NCPs with a task mapping. All the experiments are performed on a cluster of 106 compute nodes on the cloud provider Digital Ocean. Each of the Kubernetes node has 8GB RAM and 160 GB of disk space available and is based on any of the 8 available geographic locations across the world including San Francisco, Amsterdam, New York, Singapore, London, Frankfurt, Bangalore and Toronto. The locations of the testbed were chosen following a uniform distribution. The Kubernetes master node has a 32GB RAM and 640GB disk. We collected the makespan statistics as well as deployment time, running time and teardown time of the core modules (Network and Resource Profiler (DRUPE), Execution Profiler (EP), HEFT mapping and CIRCE Dispatcher). Summary of the results is presented in Table 1.

Among the core modules, number of deployed Profiler Containers, Mapping Containers, and Exec-Profiler Containers only depends on number of nodes in the system. Therefore, the deployment time and teardown time for respective Jupiter modules are almost similar while in the case of the Dispatcher (CIRCE), number of containers, deployment time and teardown time are proportional to the DAG size and increase based on the DAG complexity. Note that the deployment and teardown down heavily relies on the underlying Kubernetes architecture and the time to download the container files on each device as we simplify invoke python APIs to start the containers.

HEFT mapping requires both network and resource profiling from DRUPE and one-time execution information from Execution Profilers. The system normally needs 15 minutes (this parameter could be modified) to kickstart the activity of DRUPE, and we also have to obtain the execution information from the execution profiler, which is considerably large and grows proportionately while we increase number of tasks in the DAG. Therefore, HEFT mapping latency is the combination of running time of the execution profilers and the mapping time necessary for performing the mapping algorithm. Finally, makespan statistics of CIRCE with 10 sample input files demonstrated that the larger number of task in the DAG, the longer it takes to finish the application, which is to be expected. Note that increasing the size of the input would also increase the makespan accordingly, so that the overhead associated with network profiling, execution profiling, and task mapping would all be smaller in comparison.

5 CONCLUSION

In this paper, we presented a new container orchestration architecture for dispersed computing and an open-source software implementation of the architecture called Jupiter that builds upon the well-known Kubernetes orchestration tool. Through a set of experiments, we demonstrate trends in different runtime statistics related to the deployment of a task graph with Jupiter. While we have a fully operating open-source system, it is still at an early state with full potential yet to be unlocked. There are a lot of research question unanswered including the development of an optimized distributed mapping algorithm and a complete and methodical evaluation of different components. We welcome the research community to explore and extend the Jupiter tool.

REFERENCES

[1] 2019. Docker website - container definition. https://www.docker.com/resources/what-container. (2019).

[2] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *OSDI*, Vol. 99. 45–58.

[3] Fran Berman, Geoffrey Fox, Tony Hey, and Anthony JG Hey. 2003. *Grid computing: making the global infrastructure a reality.* Vol. 2. John Wiley and sons.

[4] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, and others. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.

[5] Docker Swarm. 2019. https://docs.docker.com/engine/swarm/. (2019).

[6] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. Kubernetes: Up and Running Dive into the Future of Infrastructure. (2017).

[7] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358.

[8] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[9] Mesos. 2019. http://mesos.apache.org/. (2019).

[10] Quynh Nguyen, Pradipta Ghosh, and Bhaskar Krishnamachari. 2018. End-to-End Network Performance Monitoring for Dispersed Computing. In *ICNC, 2018.* IEEE, 707–711.

[11] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. 2007. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing.* ACM, 43.

[12] Pranav Sakulkar, Pradipta Ghosh, B Krishnamachari, S Avestimehr, M Annavaram, and others. 2018. *Wave: A distributed scheduling framework for dispersed computing.* Technical Report. Technical Report.

[13] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 1999. Task scheduling algorithms for heterogeneous processors. In *IEEE Heterogeneous Computing Workshop (HCW).*

[14] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[15] Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications* 1, 1 (2010), 7–18.