This Dissertation

entitled

THE CHALLENGES OF SCALING UP HIGH-THROUGHPUT WORKFLOW WITH CONTAINER TECHNOLOGY

typeset with NDdiss2$_\varepsilon$ v3.2017.2 (2017/05/09) on October 2, 2019 for

Chao Zheng

This LaTeX 2$_\varepsilon$ classfile conforms to the University of Notre Dame style guidelines as of Fall 2012. However it is still possible to generate a non-conformant document if the instructions in the class file documentation are not followed!

> Be sure to refer to the published Graduate School guidelines at `http://graduateschool.nd.edu` as well. Those guidelines override everything mentioned about formatting in the documentation for this NDdiss2$_\varepsilon$ class file.

This page can be disabled by specifying the "`noinfo`*" option to the class invocation.* (i.e.,`\documentclass[...,noinfo]{nddiss2e}`)

This page is *NOT* part of the dissertation/thesis. It should be disabled before making final, formal submission, but should be included in the version submitted for format check.

NDdiss2$_\varepsilon$ documentation can be found at these locations:

`http://graduateschool.nd.edu`
`https://ctan.org/pkg/nddiss`

THE CHALLENGES OF SCALING UP HIGH-THROUGHPUT WORKFLOW WITH CONTAINER TECHNOLOGY

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Chao Zheng

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
October 2019

THE CHALLENGES OF SCALING UP HIGH-THROUGHPUT WORKFLOW WITH CONTAINER TECHNOLOGY

Abstract

by

Chao Zheng

High-throughput computing (HTC) is about using a large amount of computing resources over a long time to accomplish many independent and parallel computational tasks. HTC workloads are often described in the form of workflow and run on distributed systems through workflow systems. However, as most workflow systems are not liable for managing the task execution environment, HTC workflows are regularly limited in dedicated HTC facilities that have required settings.

Lately, container runtimes have been widely deployed across public cloud because of its ability to deliver execution environment with lower overheads than the virtual machine. This trend provides users of HTC workflows an opportunity to use unlimited computing power on the cloud. However, migrating complex workflow systems to a container environment is cumbersome.

To containerize HTC workflows and scale them up on the cloud, I synthesize my experiences on using container technologies and develop a methodology that contains seven design factors: **i) Isolation Granularity** – the granularity of isolation should be determined by characteristics for target workloads; **ii) Container Management** – container runtimes must be adapted to the distributed environment, and the under-layer distributed systems best does the management of containers; **iii) Image Management** – a cooperated mechanism can help to speed up and improve the

efficiency of image distribution in distributed environment; **iv) Garbage Collection** – timely garbage collection is necessary given the massive amount of intermediate data generated by the HTC workflow; **v) Network Connection** – excessive network connections should be avoided considering the plenty of small transmissions; **vi) Resource Management** – customized resource management mechanisms that fully consider the characteristics of the target workflow are required; **vii) Cross-layer Cooperation** – implementation of advanced features requires cooperation between the upper-layer workflow system and the under-layer cluster manager.

In addition to HTC workflows, I validate the above factors through my work of standardizing resource provisioning process for extreme scale online workloads, and observe that they are equally applicable to the HTC workflow as well as the extreme scale online workload.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENT

First and foremost I would like to thank my Ph.D. advisor Professor Douglas Thain. It has been my honor to be his student. Professor Thain has taught me, both consciously and unconsciously, how to become a good researcher with integrity. I am grateful that he always had time to teach me as well as encouraged me to purchase my research agenda. I sincerely appreciate him for all his guidance, care, and funding. Without him, this thesis would not have been finished, and I would not be here.

Since coming to Notre Dame, I have received help and support from many wonderful people. I want to thank Professor Christian Poellabauer, whose role is not only a member of my thesis committee but also a great instructor. His class inspires my interest in operating system research. I want to thank Professor Dong Wang, who taught me how to apply my knowledge to the sub-fields of computer science other than the distributed system. I want to also thank my colleagues from cooperative computing laboratory, including Nate Kremer-Herman, Tim Shafferm, Nicholas Hazekamp, and Benjamin Tovar, without whose help I can not accomplish my research agenda. I thank all my friends at Notre Dame for their companionship and support, including Boyang Li, Bingyu Shen, Yue Zhang, and Feng Gao. With them, we have shared many interesting conversations in research, politics, and miscellaneous topics.

I had the opportunity to work for the IBM Almaden Research Center as a research intern back to summer 2017. I learned various research skills during my internship, which made me a better researcher in general. I have been fortunate to have the chance to work with Doctor Lukas Rupprecht, without whom I could not complete

the Wharf project. Lukas has been a gentle and patient mentor as well as an excellent friend, whose wisdom and guidance have proven invaluable. I would like to thank my project manager Dean Hildebrand, colleagues Vasily Tarasov and Dimitrios Skourtis. I want to also thank my manager Xiang Li in Alibaba, who taught me many useful project management skills and inspired my interest in the open-source project.

My family has been an excellent source of love and encouragement. I want to thank my parents Na Lin and Wen Zheng, for their unconditional love and support. They raised me with a love of science and supported my pursuit of higher education. They have always had faith in me, which encourage me to overcome all the obstacles that I met during my Ph.D. years.

Lastly, I want to express my gratitude to my wife Duanduan Zhang. Her love, support, and encouragement have been crucial in keeping my efforts focused. She had to move around the states with me and spend many boring weekends with me in the offices. Despite all these suffering, she never stopped supporting me and believing that I will become a doctor one day.

CHAPTER 1

INTRODUCTION

1.1 High-Throughput Workflows

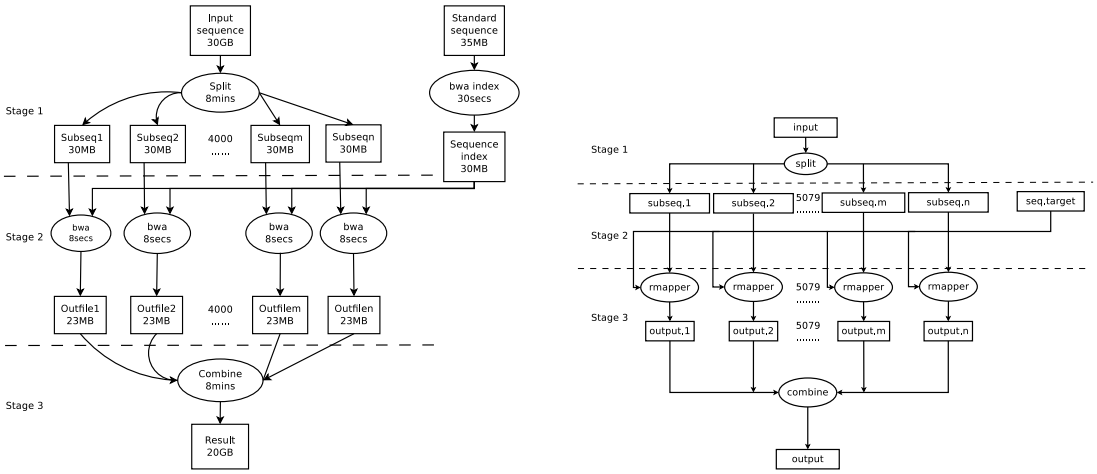
High-throughput computing (HTC) workloads usually demand a tremendous amount of computing resources for an extended time to accomplish many parallel tasks. Compared to high-performance computing (HPC), HTC focuses on accomplishing as many tasks as possible in a given period of time instead of trying to minimize the execution time of an individual task. Consequently, the key to HTC is effective resource management and exploitation.

A popular way for executing workload is users use workflow, which is often represented in the form of directed acyclic graph (DAG) (figure 1.1), to describe the data and task dependencies of large HTC workloads, and execute them through the workflow system on the distributed system.

1.2 The Challenge of Scaling Up HTC Workflows

However, most workflow systems are good at describing and running workflows but keep silent on the question of what execution environment each task of the workflow demands. As a result, HTC workflows are often limited to dedicated facilities and spend a long time waiting for compatible computing nodes.

Container technologies are recently appearing as an alternative to manage customized environments. Compared to the conventional virtual machine, rather than starting a new operating system (OS) from scratch each time, a container share the



(a) BWA Workflow (b) Shrimp Workflow

Figure 1.1: Sample Workflows, (a) The Burroughs-Wheeler Alignment (BWA) workflow consists of 4082 parallel tasks; (b) The SHRiMP workflow consists of 5091 parallel tasks aligning genomic reads against a target genome

same OS kernel with the host system, which greatly lower the overheads of delivering isolated execution environments. Also, emerging container orchestrators allow users to implement more elastic applications that can adjust underlying infrastructure during runtime. As a result, container technologies are widely deployed across private and public clouds as a new means for resource management. This trend presents users of HTC workflows a chance to access the nearly infinite computing resources in the cloud.

Nonetheless, container technologies are initially developed for **latency-sensitive programs** not discrete tasks [88, 61]. Latency-sensitive workloads, like web servers, video conferencing, and online games, intent on minimizing response time for requests, while HTC workflows concern with accomplishing the maximum amount of tasks in a given time. Their optimization goals are so separate that technologies which work for one of them, might not apply to another.

Take the resource autoscaler as an example. For platforms serving microservices, to minimize the response time to requests, the resources load must stay low, which

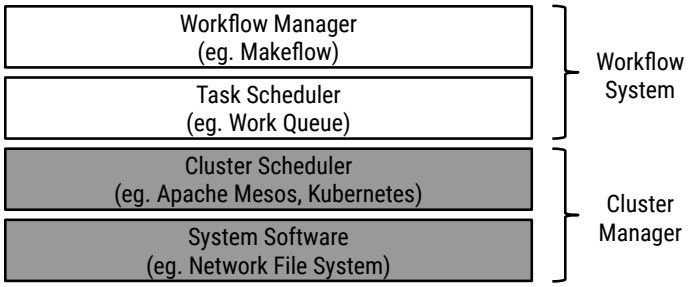


Figure 1.2. High-throughput Computing Software Stack

allows autoscalers to reactively resize the cluster based on the resource load. However, HTC workflows scavenge as many resources as they can, which results in the resource loads of the platforms are always close to saturation. Therefore, the autoscaler works for latency-sensitive workloads but will not work for the HTC workflow.

Another example is data storage. Latency-sensitive services usually are stateless and shared-nothing, which treats local memory space as a brief, single-transaction cache, while intermediate data that need to persist are stored in a stateful distributed storage. In contrast, HTC workflow often generate a massive amount of intermediate data. Repeatedly reading and writing them from and to stateful distributed storage can cause significant overheads.

1.3 How to Containerize HTC Workflows

Generally, an HTC software stack includes two components (see figure 1.2):

Workflow System, which contains two sub-components, i) **Workflow Manager**, which interacts with end user, describes the data and task dependencies of workflow; and ii) **Task Scheduler**, which receives tasks submitted by workflow manager and dispatches ready tasks to task executors running on computing nodes.

Cluster manager, which abstracts resources of computing nodes into a unified resource pool shared by applications. The cluster manager also contains two sub-

components from bottom-up: i) **System Software**, which interacts with hardware or the operating system of the individual node and encapsulate physical resources in forms that are useable to the upper-layer system, examples include virtual machine, Docker, Linux container, and distributed filesystem; ii) **Cluster Scheduler**, which unifies resources of computing nodes into a resource pool shared by applications, common ones include, Kubernetes [27], Apache Mesos [58], Apache YARN [100], HTCondor [94] and Omega [91].

Consequently, container technologies can be adapted to HTC workflow from four aspects: i) integrating container with workflow manager, ii) running task scheduler with the container, iii) bridging workflow systems to container orchestrators, and iv) optimizing container runtime with distributed software.

In this dissertation, I share practical lessons learned from five years of research on adapting container technologies to each layer of the HTC software stack, and distilled seven design factors that **can be helpful for building container-based systems for HTC workflows**. Even though this dissertation uses specific technologies as examples – i.e. Cooperative Computing Tools (CCTOOLS), Docker container runtime, Mesos and Kubernetes, I believe the seven factors are referable to users who intend to apply new technologies to the HTC software stack.

1.4 Overview of Dissertation

The rest of this disserataion is organized as follows (figure 1.3 shows the roadmap of dissertation):

Chapter 2: Background. This chapter will give the details of the systems and the software used in this dissertation.

Chapter 3: Related Works. This chapter reviews the literature on integrating container technology to the workflow system and adapting container runtime to distributed environments.

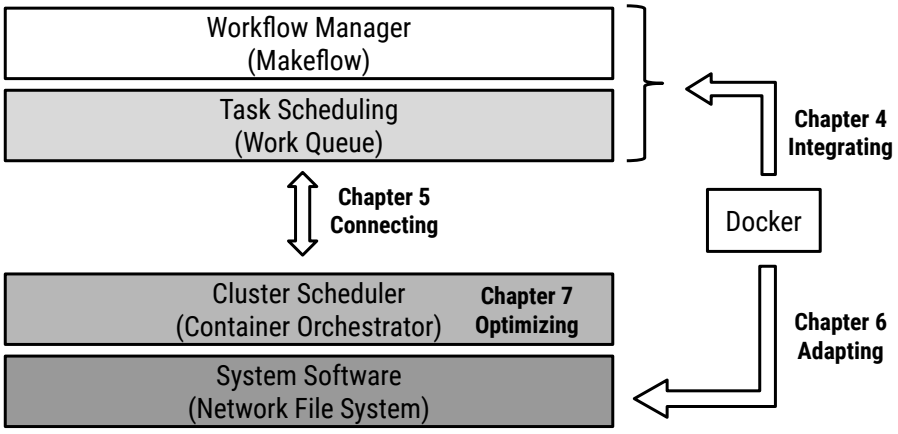


Figure 1.3. Roadmap of Dissertation

Chapter 4: Containers in Workflow System. Conventional workflow systems have been evolved for decades and are very good at executing HTC workflows. Consequently, the first option I tried is integrating container runtime into existing workflow systems. This chapter evaluates a variety of configurations for integrating container runtime into different components of the workflow system (i.e. workflow manager and task scheduler). I compare them by running practical HTC workflows with them and considering task execution time, network overhead, and resource usage. Through this work, I identify the first two design factors that are worth being considered, i.e. **i) Isolation Granularity** – the granularity of isolation should be determined by characteristics of target workloads; **ii) Container Management** – container runtime should be optimized for target workloads and the management of containers should be done by under-lying distributed system without user intervention.

Chapter 5: Container Orchestrator and Workflow System. Container orchestrators have quickly risen as a promising way for resource management on cloud. Therefore, the second option I consider is bridging the workflow system to the container orchestrator. Chapter 4 explores the possibility of connecting Make-

flow to Apache Mesos, and indicates three important design factors: **iii) Garbage Collection** – timely garbage collection is necessary given the massive amount of intermediate data generated by the HTC workflow; **iv) Network Connection** – excessive network connections should be avoided considering the large amount of small transmissions; and **v) Resource Management** – resource management mechanisms should be developed based on the characteristics of the target workflows.

Chapter 6: Distributed Container Runtime. In this chapter, I explore how to optimize the performance of the container runtime by sharing container images through the distributed filesystem and develope a distributed version of Docker runtime, i.e. Wharf, which dramatically reduces network overhead and shortens container startup time by sharing container images through the network filesystem. I summarize the lessons learned from this work into the design factor **vi) Image Management** – a cooperated mechanism can help to speed up and improve the efficiency of image distribution in distributed environment.

Chapter 7: Autoscaling HTC Workflow. Performance objectives of HTC and latency-sensitive workloads are so separate, which results in autoscaler works for one of them but does not work for the other. Chapter 6 reveals that Kubernetes's autoscaler, which resizes resource pool based on system indicator does not work for the HTC workflow. To make an accurate and timely scaling plan, I propose a new strategy that scales the resource pool based on the real-time status of the workflow system as well as the cluster manager. Through this work, I identify the sixth design factor that **vii) Cross-layer Cooperation** – to implement advanced features, like autoscaling, cooperation between the upper-layer workflow system and the underlying cluster manager is required.

Chapter 8: Cross-Checking with Online Workload. The large online workload and the HTC workflow have much in common. For example, both of them need to handle frequent and extensive resource requests; they often generate a large num-

ber of intermediate data at runtime, they all have many independent tasks running concurrently. Chapter 8 validates the seven factors by using them to design resource provisioning strategy for extreme-scale online workloads. Through this work, I conclude that the seven-factor methodology is equally applicable to both the HTC workflow and the sizeable online workload.

1.5 A Note on Terms

Next, I give definitions of some terminologies. These terms can have different meanings beyond the scope of this dissertation, but I will use them consistently as follows

- **Task:** A program instance running as a process on physcial resources.
- **Workload:** A collection of tasks that can have different properties.
- **Workflow:** An abstraction describing the data dependencies and task order of workloads in the form of DAG.
- **Workflow System:** A system translating workload into workflow and executing tasks of workflow in order.
- **Cluster Manager:** A distributed system coordinating different system softwares in cluster environment.
- **Cluster Scheduler:** A software allocating resources for tasks run on cluster.
- **Container Orchestration Tool:** A platform coordinating containers running across computing nodes (interchangeable with Container Orchestrator and Container Scheduler)
- **Online Workload:** A workload composed mostly of interactive or latency-sensitive tasks.
- **Resource Utilization:** The percentage of resources been assigned.

CHAPTER 2

BACKGROUND

2.1 Workflow Systems

There exist a variety of workflow systems that fall into several distinct communities and use cases, but all provide two standard functions of i) describing the structure of workflows, and ii) running tasks across available computing nodes.

In the Cooperative Computing Lab at the University of Notre Dame, we use two applications to perform above functions respectively (i.e. Makeflow for describing workflow structure, and Work Queue for coordinating computing nodes and systems to execute tasks).

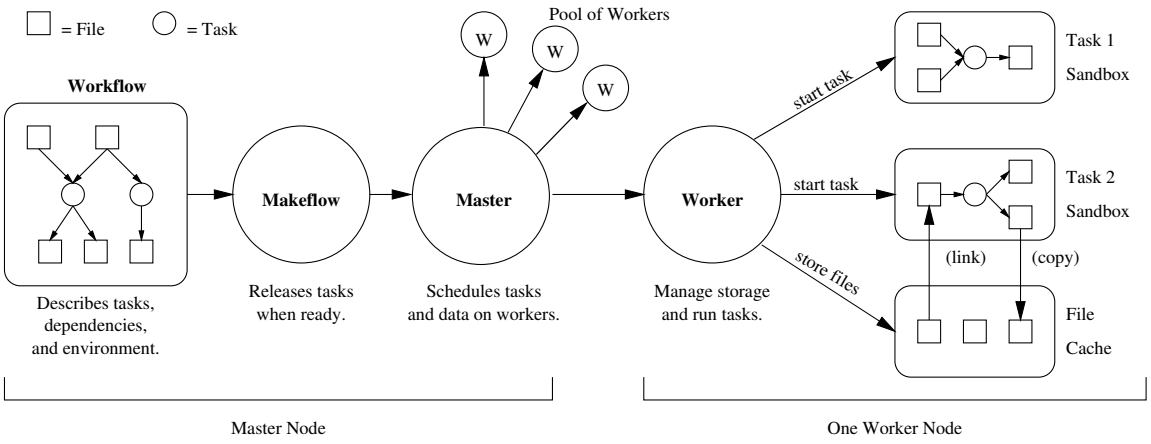


Figure 2.1: System Architecture Before Containers

Figure 2.1 shows the architecture of Makeflow and Work Queue. While our discussion focuses on these technologies, similar comments apply to other workflow systems.

2.1.1 Makeflow

Makeflow is a workflow manager for defining large complex workflows. Makeflow's syntax is similar to that of classic Make, which allows users to describe any workflow expressible in a Directed Acyclic Graph (DAG) structure. Each task of the workflow cooresponds to a rule that specifies the inputs, outputs, and command. For instance, to describe a task whose input files are `inp1` and `inp2`, output file is `oup`, and command is `./execCmd inp1 inp2 > oup`, a user would write the following rule in the Makeflow file.

```
oup: inp1 inp2 execCmd
    ./execCmd inp1 inp2 > oup
```

After the user creates the workflow description, Makeflow parses this file and generates an in-memory representation of the workflow's DAG structure, which it uses to distribute tasks to various batch systems following the data dependencies of the workflow. Makeflow is production-ready and used daily to launch complex workflows with the different batch system, including HTCondor, Work Queue, Sun Grid Engine (SGE), and Torque, among others. Makeflow has evolved to support large scale workflows that consist of millions of tasks running on thousands of nodes for months at a time, with a variety of tools available to analyze the behavior of the workflow.

2.1.2 Work Queue

Work Queue consists of a master library and a large number of worker processes that can be deployed across multiple clusters, cloud, and grid infrastructures. The master exports an API that allows the user to define tasks consisting of a command line to execute, a set of input files, and a set of expected output files. The master schedules tasks to run on remote workers.

The worker executes tasks as follows. As input files are transmitted from the master, they are stored in a cache directory. For each task to be run, the worker creates a temporary sandbox directory, links in the input files, runs the command, and then copies the output files back to the cache directory, where they await return to the master. In this way, each task is given a fresh namespace that does not interfere with other tasks. On a multicore machine, multiple tasks may safely execute simultaneously (As long as each stays in its current working directory.)

When working with Makeflow, each rule of the workflow is submitted as a task for Work Queue. The implementation of Work Queue is designed to provide precise execution semantics expected by Makeflow. Because both Makeflow and Work Queue require all files to be explicitly declared, the combination can run without a shared filesystem.

2.1.3 Resource Monitor

Workflow systems like Makeflow and Work Queue are efficient on describing and executing high-throughput workflows, but have remained silent on the question of how many resources are required by the workflow.

Users seldomly have precise knowledge of the resources (e.g., cores, memory, or disk) needed to execute a task. As a result, users have to estimate the resource requirement of the workflow and reserve a fixed amount of resources for workflows, which often leads to low resource usage or unacceptable performance.

To remedy this problem, the resource monitor [96] is developed, which measures resource usage of tasks and reports it to Makeflow. By enabling resource monitor (see figure 2.2), Makeflow can be directed to manage the computational resources assigned to tasks automatically, refining the resources allocated per job to minimize the waste. This management is built as a resource feedback loop that considers the measurement of resources used per task, the allocation and enforcement of resources, and automatic

retries on resource exhaustion.

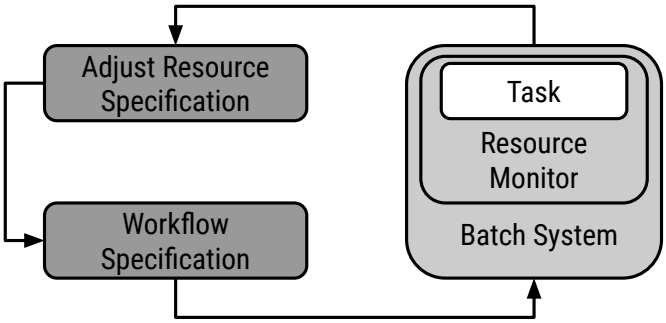


Figure 2.2: Resource Monitor

Initially, tasks are run using a maximum allowable resource size. As tasks are completed, their real usage is measured and recorded, and allocations are computed for newly created tasks to minimize the waste or maximize the throughput. Some tasks may exhaust these allocations and are retried using the maximum allowable sizes. As presented in [96], a small number of retries leads to substantial increases in throughput and decreases in resource waste.

2.2 Container Runtime

Lightweight containers are a promising operating-system virtualization technology that could be used to deliver isolated execution environments. Unlike virtual machines, containers are implemented by mounting filesystems on top of an existing operating system kernel, which largely eliminates the overheads found in traditional virtual machines. While the basic technology behind containers has been available for decades, the concept has recently seen considerable development in the Linux community, combining the `cgroups` [9] – resource control framework – and the `Linux namespaces` [30] – kernel resources partition feature – to provide complete isolation.

While the Linux kernel has included support for containers for over a decade, they

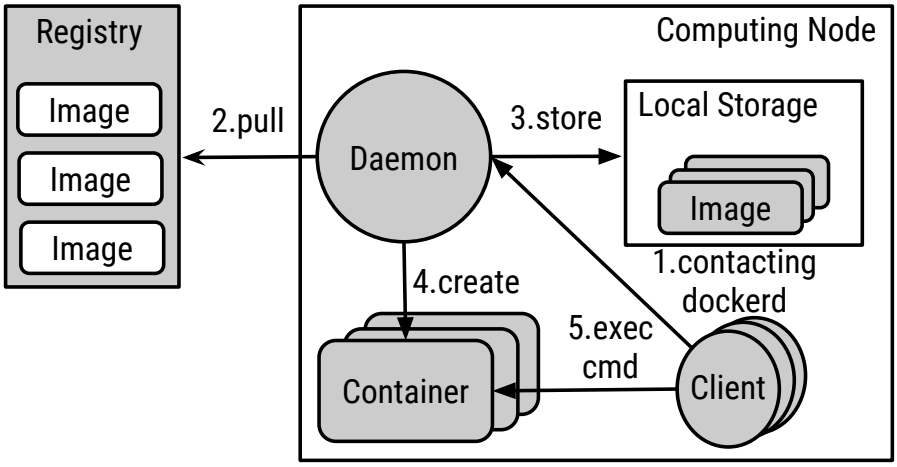


Figure 2.3: Docker Ecosystem

have only recently received extensive attention, which is due to the rise of container runtimes such as Docker [2], which drastically simplify the creation, execution, and sharing of containers.

Operating system containers [74, 92] are rapidly becoming a popular solution for sharing and isolating resources in large-scale compute clusters. As an example, Google reports to run all of its applications in containers, resulting in more than two billion launched containers per week [44]. Furthermore, all major cloud vendors have added container services to their offerings [5, 29, 18, 22].

There exist many container runtimes, e.g., Docker [15], cri-o [12], singularity [63], and LXC [3]. Among them, Docker has emerged as the most widely used one. Due to its easy interface, image management facilities, and active community, it is now relatively easy to create, share and deploy container images by name. For example, by typing this command, Docker will work as follows (see figure 2.3):

```
docker run ubuntu bwa -index seq.fasta
```

1. `docker` client (e.g., docker command line tool) contacts Docker daemon (`dockerd`) to pull the image `ubuntu`;
2. `dockerd` pulls the `ubuntu` image from remote registry;

3. `dockerd` stores image in local image storage by layer (e.g., aufs, overlay2, devicsmapper);
4. `dockerd` creates container instance by union read-only layers of the target image with an empty writable layer on top;
5. command `bwa -index seq.fasta` is executed within the newly created container.

2.3 Container Orchestrator

Container technologies have changed the way of building and delivering execution environments for applications. However, managing containers and microservices at scale is still an operational challenge. To resolve this, container orchestration tools are developed, which aim at automating the deployment, management, and scaling of container-based applications.

2.3.1 Kubernetes

Kubernetes is the container management system developed by Google, which helps developers to manage distributed applications built around micro-services and hosted in multiple containers. There exist various Kubernetes objects for describing the state of the cluster, e.g., `Pods`, `Deployment`, `Statefulset` and `Service`. The pod is the basic work unit of Kubernetes. A Pod often consists of one main container and several auxiliary containers with shared storage and network. Kubernetes allows the user to give specifications for each container, including, storage volume, disk image, forwarded ports, restart policy, and entry point. A Deployment object represents a set of identical Pods with non-unique identities, which runs multiple replicas of the application and automatically replaces failed instances. Statefulset is similar to Deployment except for each pod of the Statefulset has a unique identity. Service objects are responsible for providing a stable, virtual IP address for applications.

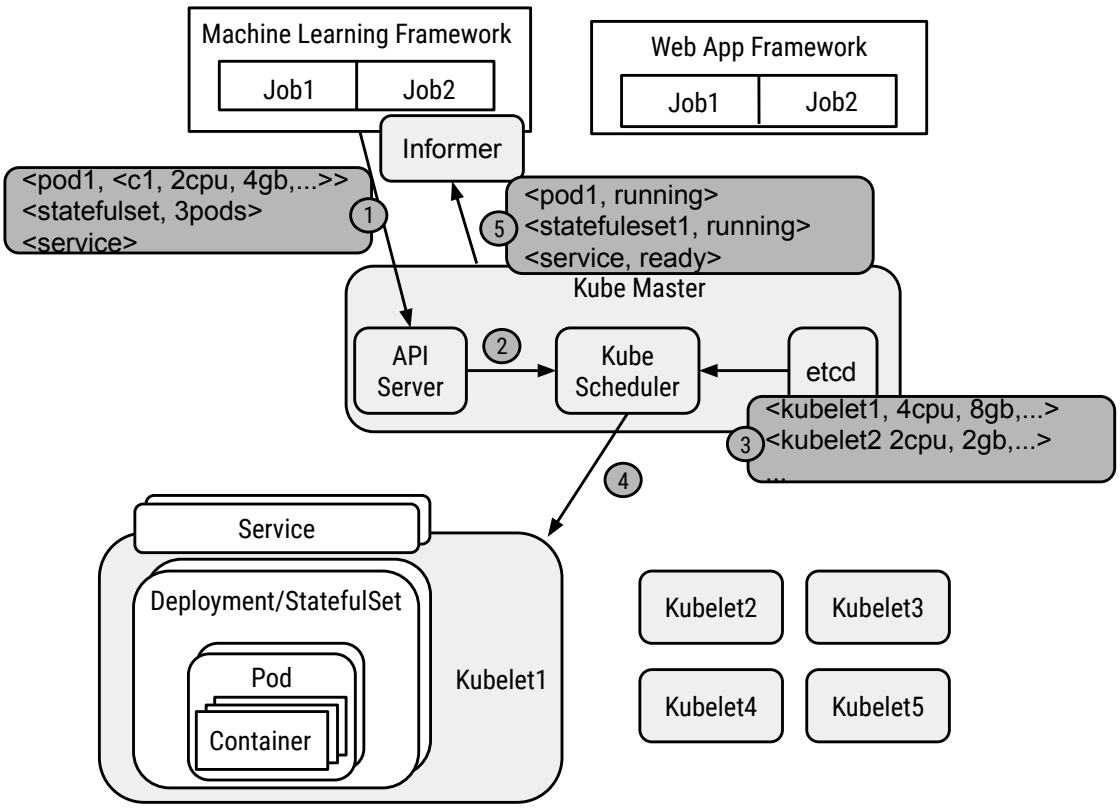


Figure 2.4: Kubernetes Architecture

While pods are ephemeral and mortal, services allow clients to visit micro-services hosted at fixed virtual IP addresses reliably.

The system architecture of Kubernetes follows the Master/Worker model, which consists of at least one master node and multiple computing nodes. The master is responsible for handling user requests, scheduling objects, and managing the cluster. Each computing node runs a container runtime, i.e. Docker and rkt, and an agent process that communicates with the master. Kubernetes adopts a monolithic scheduling architecture, which uses a global scheduler called `kube-scheduler`. The default scheduling algorithm is FitPredicate, which finds the appropriate node for a submitted object based on resource requirements and the rankings of nodes. The rankings are derived from several system parameters. For example, if multiple nodes meet the resource requirement, the scheduler prefers the machine has more allocat-

able resource, which prevents an excessive number of objects from being scheduled on only a few nodes.

Setting up micro-services or computational frameworks on Kubernetes involves following steps (see figure 2.4)

1. users sending requests for creating objects to Kubernetes `API server`;
2. `API Server` forwards requests to `kube-scheduler`;
3. `kube-scheduler` learns the latest resource specifications of computing nodes from etcd;
4. `kube-scheduler` creates objects on proper nodes;
5. if objects failed or success to run, `informer` will receive notice.

Note that the above procedure excludes the step of data transfer between the user and the Kubernetes cluster. This is because Kubernetes aims at running containerized applications that involve very little user interaction after the application starts.

2.3.2 Mesos

Mesos is a container-based resource management system. It uses containers to encapsulate computing resources and match an offer to an appropriate task. Different from `kube-scheduler` of Kubernetes, which follow the monolithic scheduling architecture, Mesos adopts the two-level scheduling model (see figure 2.5) that enables each application to have its own customized scheduler. This scheduler is used to communicate with Mesos master, which receives resource offers from Mesos master. If the scheduler finds a task that has resource requirements match an offer, it will claim the offer and ask Mesos to launch the task on the agent provide this offer. Since a long task can occupy many offers that can be used by other short tasks, Mesos encourages frameworks to run short tasks, which may be a limitation for run-

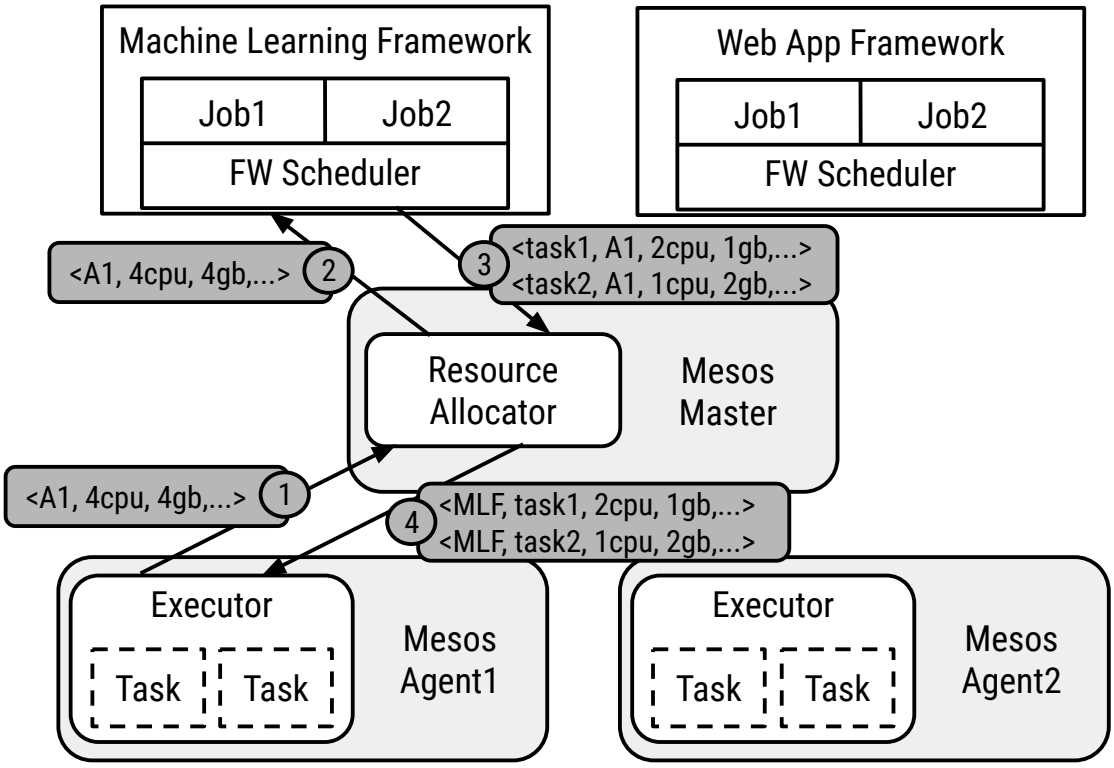


Figure 2.5: Mesos Architecture

ning high-throughput workflows on Mesos, because large workflows normally contain heterogeneous tasks that can be either short or long.

CHAPTER 3

RELATED WORK

In this chapter, I discuss works related to the dissertation. I start with introducing some notable workflow systems other than Makeflow and Work Queue (section 3.1). Then I discuss container technologies in general, including container runtimes other than Docker (section 3.2), container image management (section 4.4). I introduce several popular container orchestration platforms (section 3.4) and virtualization in HTC environments (section 3.5) Finally, I conclude with discussing cluster resource management (section 3.6) and autoscaling technologies in cloud environments (section 3.7).

3.1 Workflow Systems

There exist many workflow systems other than Makeflow and Work Queue, which share similar principles but fall into different communities and use cases. Some notable ones are:

Kepler [69] provides a graphic user interface that allows users to create a scientific workflow by merely dragging and dropping components and connecting the components to construct a specific data flow. Kepler is designed for users with little background in computer science. For example, a Quantitative analyst can develop a new component that uses R statistical analyses and share it. Others do not have to know how to program in R and the pre-programmed R components can be dragged into their workflows. Kepler also allows users to access to continually growing data repositories, computing resources, and workflow libraries global wide.

Galaxy [**59**] provides a way to compose multi-step computational workflows. Similar to Kepler, it also renders a graphical user interface for assisting users with limited computer science background to create the workflow. In addition to a workflow system, Galaxy is also a biological data integration platform that allows users to upload their experimental data. Though Galaxy is initially developed for genomics workflow, it is essentially domain agnostic and can be applied to different scientific domains.

Apache Taverna [**59**] is a collection of tools used to design and run scientific workflows from different domains. Taverna is written in Java and contains three components: i) Taverna Engine, which executes tasks of workflows parallelly across computing nodes; ii) Taverna Workbench, which is the desktop client application; and iii) Taverna Server, which empowers the user to execute workflows remotely.

Swift [**108**] project includes not only Swift runtime system for executing workflows but also Swift script language for describing workflows. The workflow described in Swift language can be executed in parallel automatically if there are no data dependencies with tasks. Furthermore, Regardless of the execution order of statements, computations of the same Swift workflow are guaranteed to be deterministic.

Pegasus [**46**] workflow management system maps scientific domain and the execution environment. It match high-level workflow abstraction to distributed resources, locates required input data, and makes plan for all data transfers and job submission operations for executing workflows. In addition, Pegasus is able to restructure the workflow to improve the efficiency and performance of running large workflows on large distributed infrastructures.

Directed Acyclic Graph Manager (DAGMan) [**51**] is a meta-scheduler for HTCondor [95]. The resource scheduler of HTCondor arranges tasks to proper machines, but does not schedule task based on dependencies. To submit tasks in order, DAGMan analyzes the DAG structure and dispatch tasks to HTCondor in such a way

as to enforce the DAG's dependencies. Additionally, DAGMan also handles failed recovery and reporting.

3.2 Container Runtimes

In addition to Docker, several container runtime systems have emerged to meet specific user requirements. Charliecloud [87] enables container workflows without requiring privileged access to data center resources through user namespaces. Singularity [63] is an alternative container solution that aims to provide reproducibility and portable environments. It prevents privileged escalation in the runtime environment, which improves cluster security when containers can run arbitrary code. Other two popular runtimes are Snappy ubuntu [36] and RedHat Project Atomic [4]. Both platforms minimize operating system level contents and automate the process of deploying containers across multiple hosts.

Combining container technology with a distributed system is commonly done. There exist container runtimes that are dedicated to distributed environments. rkt [35] is a container engine initially developed for CoreOS [1], which is characterized by its pod-native approach, pluggable execution environment, and well-defined surface area. Rather than being used independently as Docker, rkt is normally used as integration with other systems. The emerging of container orchestrators also brings forward new requirements for lightweight containers. cri-o [12] is a lightweight container runtime for Kubernetes, which implements the container runtime interface (CRI) of the Kubernetes. cri-o keeps only necessary functions required by Kubernetes, which makes it a lightweight alternative to Docker when running in Kubernetes.

3.3 Container Image Management

With the prevalence of container technologies in large distributed environments, many works have been done to improve the efficiency of managing and distributing

container images. These works mainly focus on two aspects.

One is reducing the cost of image management by **Optimizing Registry and Daemon**. VMware Harbor [37] is an optimized registry server designed for enterprise-level clusters. To expedite the distribution of container image, each cluster holds a regional registry that periodically synchronizes with a global registry. With this mechanism, clusters can pull and push the image from the region registry, which decreases the turnaround time and reduce the network overhead. CoMICon [77] introduces a decentralized, collaborative registry design to enable daemons to share images. Specifically, on each node, a CoMICon agent is running with the Docker daemon, which keeps sending the image/layer information of the daemon to CoMICon registry. Consequently, rather than pulling new layers/images from a remote registry, daemons check with CoMICon registry and retrieve the desired layers from nearby daemons if they have them. By using CoMICon, the application provisioning time has been reduced by 28%. Anwar et al. [39] analyze registry traces (e.g., image popularity, HTTP request statistics, and requests correlation) and suggest several mechanisms for speeding up image pulls (e.g., image prefetch and two-level cache).

The other is speeding up image distribution by **optimizing data transfer protocol**. Slacker [57] proposes to lazily load image content from a shared storage backend to reduce the amount of transferred available image data. By using Slacker, the container startup time can be improved by $5-20\times$. Shifter [54] supports sharing images from a distributed file system, which implements its flat image format to serve images. When the same backing filesystem is used, Shifter successfully accelerate the container startup time by $7-10\times$. Dragonfly [67] and FID [62] share the same idea of using a P2P protocol to speed up the image pulling across daemons within the same cluster. By using FID, the network overhead has been reduced by 97%, and image distribution time has been decreased by 83.50% compared to origin Docker.

3.4 Container Orchestration Platforms

Other than Kubernetes and Apache Mesos, there exist various container orchestration platforms which have been widely deployed:

Docker Swarm [**14**] is a mode for cluster management embedded in Docker Engine. A swarm is a cluster of Docker hosts with swarm mode enabled and acts as managers (managing membership) or workers (running container). As an embedded mode, swarm inherently works with Docker better than other orchestrators, which is easy to have new/old Docker hosts join/leave. However, without a robust centralized manager, swarm clusters are relatively hard to manage at scale.

Mesosphere Marathon [**28**] is software that expands upon Apache Mesos, which is designed for executing long-running applications. Apache Mesos is initially developed for generic cluster resource management with lacking features – like load balancing, health checks and application reboot – desired by micro-services. The emergence of Marathon remedies this problem and makes Apache Mesos a fully-functional platform for micro-services.

CoreOS [**1**] as a lightweight Linux distribution aims at providing infrastructure to clustered deployments, running all services and applications inside containers, which facilitate the security, reliability and scalability of clusters.

Nomad [**21**] is a lightweight workload orchestrator, which inherently supports diverse virtualization technologies, including container, virtual machine, or even standalone application. Nomad merely aims at providing cluster management and task scheduling and therefore lighter than other orchestrators.

Even though the implementation details of the above orchestrators vary, they all support Docker container runtime. Therefore, the seven design factors distilled from my previous works can be applied to them as well.

3.5 Virtulization in HTC Environments

There exist many ways to improve the performance of the HTC system whose main enabling technology is virtualization. By using hardware with virtualization technology, the latency caused by virtualization [83] is reduced. Another way is to enable the hypervisor to deal with the guest processing directly, which eliminates the overhead for latency-sensitive tasks [79]. By reducing the memory footprint in a virtualized large-scale parallel system, the system performance, reliability, and power can also be enhanced [110].

With the growing adoption of container-based computing, researchers have started to consider using container runtime in HTC environments. Generally, there exist three options for connecting container technologies to HTC systems.

The first is **integrating container runtime into the workflow system**. Batch systems like IBM Spectrum LSF [23] and Altair's PBS Professional [33] all provide support for Docker containers. Though these deployments exploit existing workflow system that has been developed over the years to support HTC workflows, it is complicated to deploy such a system on a commercial cloud which often renders more and newer hardware with evolved and compelling infrastructure.

Some other works try to look into the possibility of bridging workflow systems to container orchestrators. Some well-known ones are, KubeFlow [25], which is dedicated to deploying leading data analytics frameworks and workflow systems on Kubernetes; and Argo [6], a container-native workflow engine for executing parallel tasks on Kubernetes. This option accommodates properties of both HTC workflows and container orchestrators. However, be cautioned that most container orchestrators are initially developed for micro-services, not discrete jobs, thus system optimizations are required.

The third is allowing workflow systems and container orchestrator to coexist in

a shared environment. Navops by Unvia [31], takes this approach. It includes customized scheduling policies for HTC workloads using the Kubernetes scheduler, which makes Kubernetes the cluster resource manager responsible for assigning resources for both non-containerized HTC workloads and containerized micro-services. Navops set up the HTC scheduler as a service on a Kubernetes cluster.

3.6 Cluster Resource Management

In the cluster environment, there exists a variety of resource management mechanisms that can be divided into three categories.

Central Scheduling. Schedulers of this category usually have a central resource manager, which is responsible for assigning resource to all tasks, some well-known systems are Borg [101], TORQUE resource manager [93], old Hadoop scheduler [107] and Sun Grid Engine (SGE) [53]. As the resource manager has to hold the information of all computing nodes, when the cluster scales, the resource manager has to bear the network and storage pressure.

Two-level Scheduling. Schedulers using this mechanism have a central resource scheduler, which allocates resource quota for frameworks running on the cluster, and framework scheduler for each framework, which assigns resources to tasks within its quota. Some famous two-level schedulers are HTCondor [95] and YARN [100]. These systems enable applications to have their own task assignment semantics, which is more flexible and allow different workloads to share the resource. However, it comes with a drawback: the framework scheduler lacks the global view of the cluster, which impedes the implementation of advanced features, like task preemption and dynamic resource provisioning.

Distributed Scheduling. Schedulers of this category don't have a centralized scheduler; instead, each framework running on the cluster does not share a global resource manager but fully relies on its own scheduler, which makes scheduling de-

cisions based on its local state. As there is no synchronization process, a scheduling plan can be quickly established. A famous distributed scheduler is Sparrow [81].

3.7 Autoscaling in Cloud Computing

The critical characteristic of cloud computing is elasticity, which allows users to resize resource pool on-demand. However, deciding the right amount of resources is not trivial.

For workloads facing unchanged throughput, preserving a fixed amount of resources is possible, while applications on cloud often need to handle unplanned events, fluctuating throughput, and spiking load, which makes autoscaling techniques crucial. When it comes to autoscaling, previous studies can be classified into two groups.

Works of the first group try to solve the problem from **users' perspective and focus on capacity autoscaling in clouds**. They aim at adjusting the allocated capacity to satisfy the required performance or SLO. Therefore, the two main hurdles of these works are accurately predicting the capacity and ensuring the capacity is available when required. To accomplish these, four approaches are commonly used.

Threshold-based Autoscaling. This approach requires users to specify an indicator or a set of indicators that implies real-time system load. Then, the system autoscaler reactively increases the capacity when the system load is over the thresholds and vice versa. Due to its simplicity and intuitive nature, this method is widely adopted by cloud providers and third-party tools [16, 34]. However, setting thresholds requires users to have a deep understanding of the workload, which is rare among average users.

Queuing Theory. This approach scales the cluster based on the length of the task queue and the average waiting time, which uses either the conventional queuing model or queuing network to model the applications' behavior. Previous works have mainly focused on developing an advanced queuing model for more complicated

cases [75, 102, 99]. The main pitfall of this approach is the nonuniversality. The queuing model is usually highly optimized for specific cases and might not work when external factors are changed, e.g., task arriving rate, spiking throughput, hardware upgrading, etc.

Control Theory has been widely used to manage the resource capacity for web server systems, data centers, and other systems [65, 66, 82, 84, 85]. There exist three commonly used control system: i) Open-loop controllers, which changes the system only according to a current state; ii) Feedback controllers, which correct the system by analyzing the previous system outputs; iii) Feedforward controllers, which estimates the system behaviors based on models and react before errors occur. In practical, Feedback and Feedforward controllers often used together.

Time-series Analysis. This approach analyzes the historical data and trys to find a repetitive pattern in time-series, which usually contains four steps: i) splits the lifecycle of workloads into a sequence of data points, ii) periodically samples several intervals between data points, iii) predicts the future system load, iv) and changes the capacity. Existing works using this approach mainly focus on remove noise from historical data [65, 84] or using mathematical methods [43, 56] to improve the accuracy of prediction. Though this approach works well in domains of finance, economics, and bioinformatics, when it comes to workloads that might have unpredictable behaviors, fully relying on the prediction model might result in failing to handle the incidental event.

The works from the second group try to tackle the problem from **infrastructure provider's perspective, which aims at improving the resource utilization and energy efficiency of the whole infrastructure**. Comparing to individual users, cloud providers often rely on consolidation strategies [55, 73] that synthetically consider energy usage as well as application performance, or adopting resource overcommit strategy [41, 98] which merely increase the resource allocation of applications

if they can enhance overall revenue.

The autoscaling mechanism discussed in chapter 7 tries to solve the problem from the users' perspective and improve the accuracy of capacity estimation and timeliness of resource provisioning. It accomplishes this by combining two existing autoscaling mechanisms, i.e. Queuing Theory and Control theory, with new features provided by Kubernetes, i.e. Infrastructure As Code [24] and Cloud Native [10].

CHAPTER 4

CONTAINERS IN WORKFLOW SYSTEMS

4.1 Introduction

In this chapter, I will introduce my work of integrating the container runtime (i.e. Docker) into the workflow system (i.e. Makeflow and Work Queue) and the first two design factors. An abridged version of this chapter has been published as "Integrating containers into workflows: A case study using makeflow, work queue, and docker" in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*.

Back in 2014, Docker was just coming into people' view as a tool for building testing environments. In addition to software testing, I see an opportunity of using container runtime to manage execution enironments for HTC workflows. However, there are few container orchestration tools available back then, therefore, the fastest way of using the container runtime in HTC environment is integrating container runtime into the existing workflow system. The expected outcome is a comprehensive system that has workflow system focus on executing tasks across available computing nodes, and container runtime responsible for delivering execution environment for tasks.

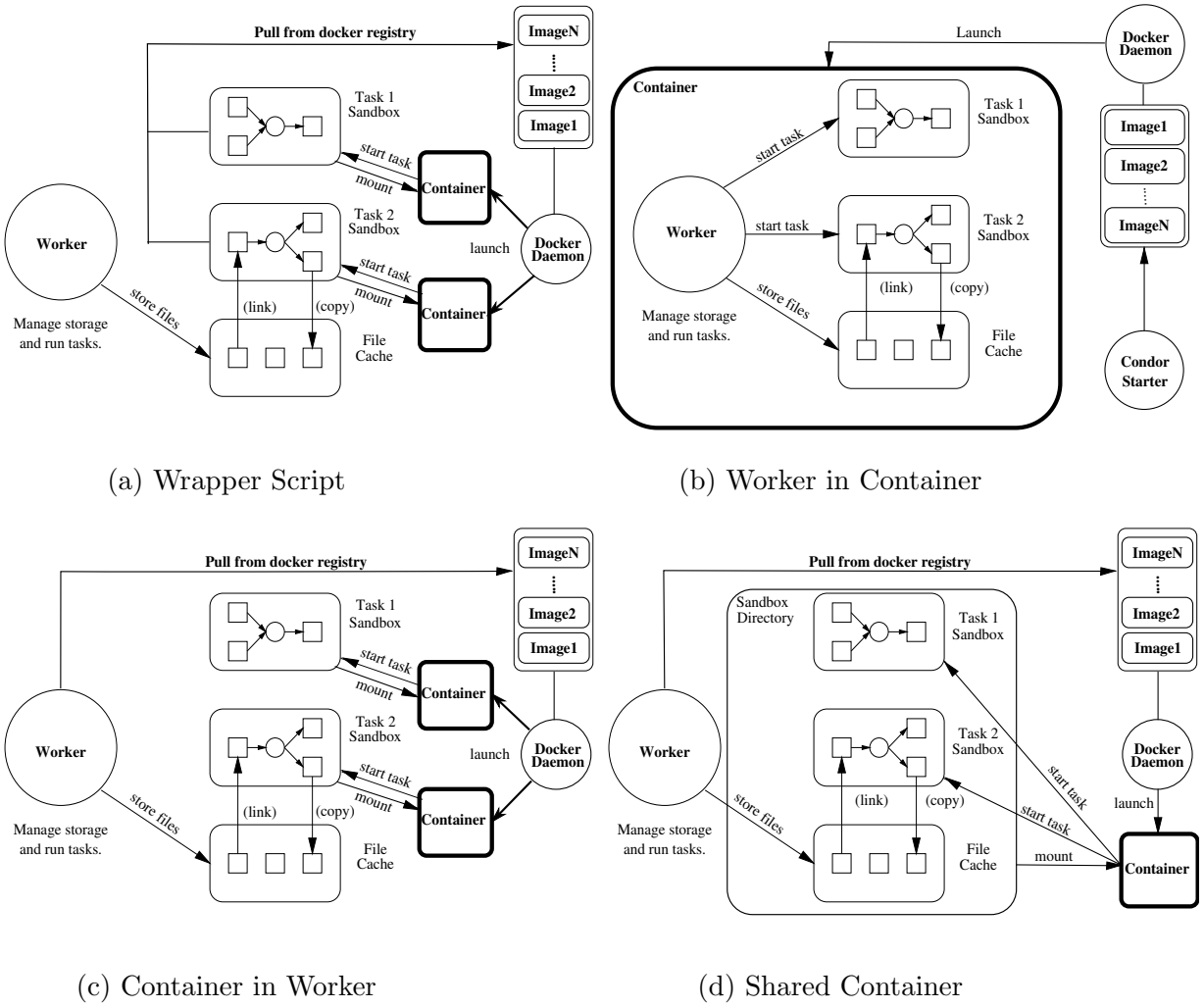
Generally speaking, the workflow system runs each task as a process or thread within a private sandbox (e.g., a filesystem directory) on a computing node, and container runtime starts container instances as new processes with isolated namespaces and exclusive computing resources but share same host OS kernel. To coordinate these two contexts, I break down the design considerations into three categories:

1. what level of isolation need to be achieved, i.e. task level or worker level.
2. How to manage the lifecycle of containers, i.e. when to initiate or delete containers.
3. How to distribute container image across computing nodes, i.e. rely on centralized registry or distribute as part of workflow data.

Specifically, I examine four different methods of integrating Docker with Makeflow and Work Queue.

1. Wrapping each task at the workflow level, which is simple but inefficient due to lack of information about image state (see figure 4.1a).
2. Running each worker inside a container, which eliminates startup overheads, but exposes more of the system to container overheads (see figure 4.1b).
3. Running each task inside a container, which limits the scope of the container, but increases startup/shutdown events (see figure 4.1c).
4. Running multiple compatible tasks inside one container, which minimizes startup/shutdown events, but decreases isolation (see figure 4.1d).

To evaluate these configurations, I execute a bioinformatics workflow which consists of a large number of relatively short tasks, which emphasizes the efficiency of container startup and shutdown. The final configuration achieves performance very close to that of a system without containers, if one is willing to sacrifice isolation between tasks. I observe that this technique requires that the execution system must be container-aware, rather than simply wrapping each task with the desired container operations. The lesson I learned is there exists tradeoff between granularity of isolation and performance, which leads to the first two design factor, i.e. **Isolation Granularity** – the granularity of isolation should be determined by characteristics of target workloads; and **Container Management** – container runtimes must be adapted to the distributed environment, and the under-lying distributed systems best does the management of containers.



(a) Wrapper Script

(b) Worker in Container

(c) Container in Worker

(d) Shared Container

Figure 4.1: Container Management Options

4.2 Execution Environments

As described so far, Makeflow and Work Queue accurately capture the data dependencies of a workload, but say nothing about the **environment** in which a task should be executed. Including the program as an input dependency is a good first start, but does not capture all of the shared libraries, script interpreters, configuration files, and other items on which it may depend. Ideally, the end user will provision worker nodes that have exactly the same operating system, installed applications, and so forth.

Unfortunately, this is easier said than done. My experience is that end users construct complex workflows in an environment not of their own creation, such as a professionally managed HTC facility. In this situation, the user's environment is a mix of a standard operating system, patches and adjustments made by the staff, locally installed applications, and items from the user's home directory. When moving the application to another site, it is not at all obvious what components should be included with the workflow. Should the user copy the executables, the libraries, the Perl or Python interpreters, or perhaps even the entire home directory?

An alternate approach is to **define the environment explicitly** when the workflow is created. Rather than accept the current environment as the default, the user would be required to explicitly name an image that contains the operating system, applications, and all other items needed by the application. This image could be constructed by hand by the user, but is more likely provided by administrators. Once explicitly named, the image can easily be transported along with the workflow. A single image might apply to all tasks in the workflow, or might vary between tasks.

However, the combination of these technologies is not as simple as putting `docker run ubuntu` in front of every command, because the workflow data dependencies must be connected to the executing image. To address this problem, there are two different design questions need to be answered. The first is **container management**: namely, which component of the system is responsible for configuring, deploying, and tearing down containers. The second is **image management**: namely, how the container images must be moved to the (possibly thousands) of workers that comprise the system in an efficient way. The remainder of this chapter considers each of these problems in detail.

4.3 Container Management

First, I consider four strategies for assigning the responsibility of creating and tearing down containers within the workflow system. Each of these methods has been implemented and is evaluated below.

Base-Architecture. (Figure 2.1) The basic architecture of Makeflow and Work Queue effectively uses a directory as a placeholder for a container. For each task to be executed, the input files are linked from the cache directory of the worker into the task sandbox directory, the task is run within the directory, and then the output files are copied back into the worker's cache for later use. This can be thought of as a (very) lightweight container in as much as each task has an assigned namespace, assuming each task is well behaved and stays within its current working directory.

This method has the advantage that it is simple, requires no special privileges, and imposes no overhead on the execution of the application. Of course, the only environment that can be provided to the application is the operating system in which the worker runs.

Wrapper-Script. (Figure 4.1a) The simplest step up from the base architecture is to use a wrapper script to provision a container for each task. A small script can be written which will contact the local `dockerd`, pull the desired image to the execution host, run the desired task in the container, then tear down the image. To simplify access to the task's files, the task sandbox directory is mounted into the container as the task's working directory, such that neither the task nor the worker must change their behavior.

This method has the advantage that no change is necessary to either Makeflow or Work Queue, and can be applied transparently by the end user. Each task will be isolated from all other concurrent tasks. If necessary, different tasks could execute in different environments. (In the common case that all tasks require the same

environment, the `--wrapper` command-line option to Makeflow can be used to easily apply a single wrapper globally without modifying the workflow itself.) However, as shown below, this method imposes a container startup and shutdown cost on each task, and does not give the distributed system visibility into the location and movement of the (possibly large) images.

Worker-in-Container. (Figure 4.1b) An alternative approach is to simply take the entire worker itself and place it into a container for the entire duration of the run. This requires a small modification to the provisioning of the worker itself, to pull the image and create the container. In fact, the same wrapper script as above can be used, if the worker itself is provisioned by an underlying system manager.

This approach succeeds in delivering the desired execution environment to each task and avoids paying the startup and shutdown costs for each task. In the common case where all tasks in the workflow require the same environment, one can easily imagine provisioning a number of workers in bulk before the workflow execution. However, it does not provide isolation between tasks, which all execute in a sandbox directory in the same container, so the same degree of trust must be assumed as in the base case. More subtly, the worker itself must pay the costs of executing within the container, such that there may be a penalty applied to the network communication between master and worker, as well as in the management of the local cache directory. Finally, this configuration relies on the union filesystem (e.g., aufs, overlayfs, etc.), which is reported to cause non-negligible overheads when using a container with multiple layers and deep-nested filesystem structure [50].

Containers-in-Worker (Figure 4.1c) Another approach is to modify the worker code itself to run each task within a specified container. The effect of this is very similar to that of the wrapper script, but with one important difference: the worker itself now has some knowledge of the state of the local `dockerd` and can execute more efficiently. Rather than attempting to pull and transform images for each container

invocation, the worker can prepare the image once, and then instantiate multiple containers from the same image.

This approach allows each task to provision a distinct container and allows the worker itself to avoid container overheads, while still providing isolation between tasks. Further the container image itself can becomes an input dependency of the task, which allows for the scheduler to explicitly take advantage of this information. For example, tasks can be scheduled preferentially to nodes where the image has already been transferred and cached.

Shared-Container. (Figure 4.1d) Finally, I may attempt to combine the benefits of multiple approaches by allowing tasks to share the same container where possible. In the Shared-Container approach, the worker is again responsible for creating and deleting containers, but will only create one container for each desired environment. Containers are not removed when tasks complete, but remain in place for the next (or concurrent) task to be placed inside.

This approach allows each task to run in different environments, where needed, avoids the overhead of multiple container creation, but does not provide isolation between tasks beyond the base case. As with the previous case, it also gives the scheduler visibility into image locations.

4.4 Image Management

Considering a large scale workflow application running on thousands of workers, the cost of moving the environment to each node can become a significant component of the overall cost, depending on the form in which it is transferred and the source of the transfer. There are three commonly used forms for communicating an executable environment in Docker:

A **dockerfile** describes the entire procedure by which an image is built, starting with the base operating system image, adding software packages, and running

arbitrary commands until the desired result is achieved. Building an image from a dockerfile is comparable to installing a new machine from scratch and could take anywhere from minutes to hours, depending on the number of layers involved. However, a dockerfile is quite small (1KB or less) and is easily transferred across the system.

A **binary image** is the result of executing a dockerfile, and is the executable form of a container ready for activation, with all files laid out into the form of a multi-layer directory tree that can be directly mounted and used. The image is a binary object required backend support of specific union filesystem and may not be appropriate for portability or long term preservation.

A **tarball** is a more portable form of a binary container, in that all the layers of the filesystem have been collapsed into a tree of files and directories encoded in the standard `tar` format. A tarball is much more suitable for sharing and preservation, but must be unpacked and encoded back into a binary image before it can be executed directly by Docker. A tarball can be used directly by other technologies, such as a `chroot` based sandbox.

By default, Docker encourages end users to create containers by pulling binary images from the global image registry (i.e. Docker Hub). This would be the result of using the Wrapper-Script method with reference to an image name. This method is quite useful for sharing frequent-used container images over limited number of computing nodes. But, when employing thousands of workers for a single workflow, this could result in extraordinary loads on the public network. Further, using the central hub may be inappropriate for images with security, copyright, or privacy concerns.

Another approach would be using the workflow system to distribute the dockerfile itself, relying on the workers to construct the desired images at runtime. This would dramatically reduce the network traffic from workers pulling images, but would result in all workers duplicating the same effort for minutes to hours to generate the same

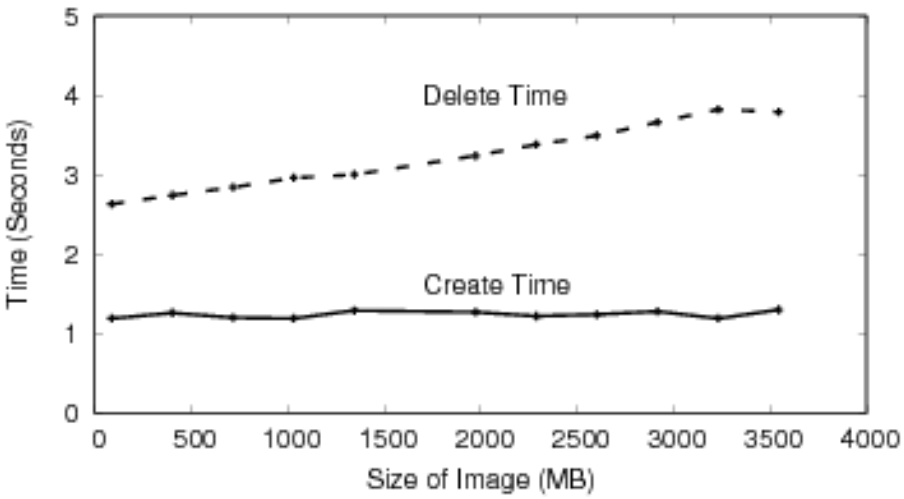


Figure 4.2. Container Creation and Deletion

resulting image. The effort would be better expended in one location before allocating workers.

In the context of a large workflow on thousands of nodes, the most appropriate solutions seems to be for the workflow manager itself to build the desired environment image on the submit machine, either by executing a dockerfile or by pulling an image from a repository. Once generated, that image can be exported from Docker as a portable tarball and then included as an input dependency for each task. In this way, if the submit machine is inside the network of the computing cluster, it can take advantage of the existing file transfer mechanisms on the data center network, which has much larger network bandwidth than the public network.

However, users of high-throughput workflows often create container images on private workstation outside of the data center network. To study the system behavior under this circumstance, following examples using the public registry for images management, where they are pulled and installed.

4.5 Experimental Evaluation

I evaluated each of the configurations above by implementing them as options within Makeflow and Work Queue, then observed the performance of a large bioinformatics workflow on a Docker-enabled cluster. The cluster consisted of twenty-four 8-core Intel Xeon E5620 CPUs each with 32GB RAM, 12 2TB disks, 1Gb Ethernet, running Red Hat Enterprise Linux 6.5 with Linux kernel 2.6.32-504.3.3.el6.x86_64 and Docker 1.4.1.

Before evaluating the workflow as a whole, I performed some basic micro-benchmarks on a single machine to evaluate the low level performance of the container technology. I employed `sysbench` to measure CPU performance and memory bandwidth, `netperf` to evaluate the performance of TCP throughput both on and off the machine and `bonnie++` to evaluate local disk performance. I expected (and confirmed) that the CPU, memory, and network benchmarks would all result in virtually indistinguishable performance between the host and container, since the container mechanism primarily affects the namespace of kernel objects and not the direct access to machine resources.

However, based on previous reports, I expected to see that the I/O performance within the container would take a significant penalty due to the use of union filesystem. For example, Felter [49] mention that `aufs` should be avoided due to the overhead of metadata lookup in each layer of the union filesystem. What I observed was more difficult to characterize: generally, reads from `aufs` would see performance similar to that of the host, while writes to `aufs` would be sometimes be *faster* than the host, as changes were simply queued up until a later `docker commit` which would flush changes out. The overall effect was very inconsistent performance, sometimes better and sometimes worse than the host filesystem.

That said, `aufs` performance is less relevant in this setting because three of the

configurations relies on mounting a sandbox directory from the host in order to access workflow data. The performance of this mount was indistinguishable from the host. The Worker-in-Container relies on `aufs`, but the size of the input file and output file for each task are around 10MB to 20MB. To access small files like this, the difference of I/O performance is negligible.

Where overheads were more clear was in the cost of creating and deleting containers at runtime. To evaluate this, I created a base operating system image, and then progressively increased the size of the image by adding files. Then, I measured the minimal container startup time by executing `docker run debian /bin/ls` and then the time to delete the image after the command completed. Figure 4.2 shows the average of ten startups and deletes at each size. I was surprised to see that the startup time was essentially constant with respect to the image size, but the deletion time increased linearly with the image size. These overheads have a significant effect on workflow executions, depending on the container management strategy chosen.

To evaluate the overall system performance, I executed the BWA workflow in each of the five container management configurations discussed above. Commonly, there are three kinds of workflow, workflows which consist mainly of long-running tasks, the ones that containes many short-running tasks, and the ones including both type of tasks. For long-running tasks, in comparison to the overall execution time, environment set up time is negligible. While the workflow has many short tasks gains concrete benefits from shorter environment setting up time. In order to present the maximum level of speedup gain from applying lightweight container technology, The selected BWA workload consists of 4082 short-running parallel tasks.

The workload was run on a cluster, each task a single-core process, such that up to 192 tasks (or containers) would run simultaneously. Workers were deployed onto the cluster in each of the five configurations described earlier, with Makeflow running on the head node to coordinate the computation.

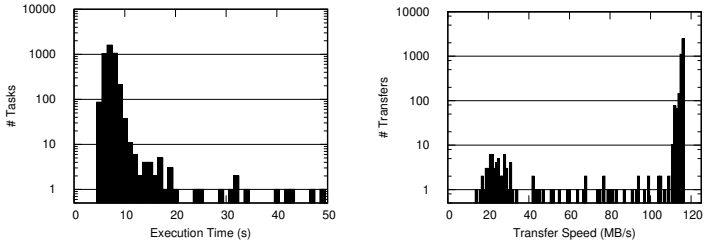
Figure 4.3 shows the results of each run, with each configuration in a row. The first column gives the key details of the total runtime, the average execution time of each task, and the average transfer performance between the master and the worker. The second column gives a histogram of individual task execution times for the 4000 tasks of the workflow, while the third column gives a histogram of transfer rate of individual file between the master and the worker to support each task.

As expected, the Base-Architecture has the fastest overall execution time (25 min) and a compact distribution of task execution times. The simplest extension, the Wrapper-Script, has the worst performance of the five (38 min), primarily because each task must independently pull an image, execute the task, and then clean up the image when done. The variation in task execution times is also much higher, due to the interference between tasks managing images and doing work. Worker-in-Container achieves faster and more compact task execution times but still pays a penalty due to overhead applied to the worker itself. Containers-in-Worker shows a slightly worse performance because each task must create and delete a container, but the worker handles the image management. Finally, the Shared-Container case achieves performance very close to that of the Base-Architecture, because the containers are created once per worker, and then shared among up to eight tasks simultaneously.

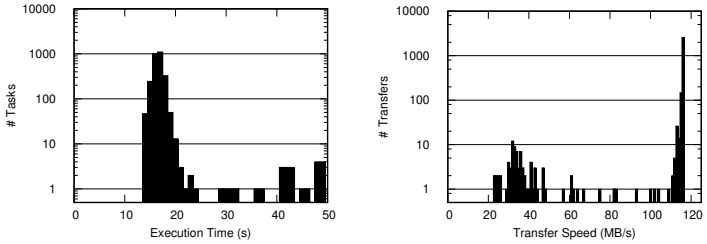
I included the file transfer histograms because I expected to see some variation in transfer performance, particularly in the case of the Worker-in-Container. However, I do not see any differences significant enough to draw conclusions.

As can be seen the selection of a strategy for managing containers within a workflow has a significant impact upon the bottom line, primarily due to the non-trivial expense of booting and removing containers. A tradeoff must be made between achieving complete isolation and maximum performance.

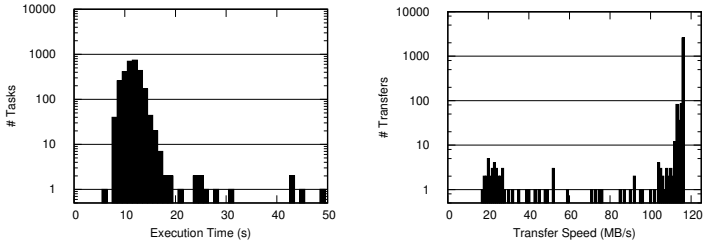
Basic Structure
Total exc time: 25mins
Task time: (8.28 +/- 2.52)secs
Transfer rate: (115 +/- 11)MB/s



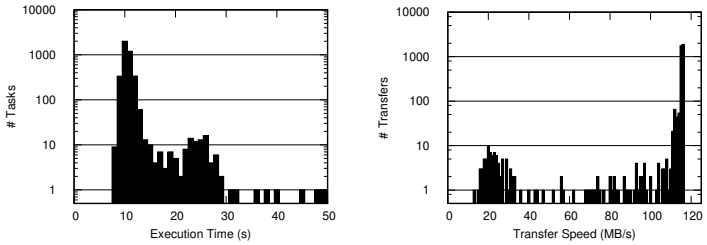
Wrapper Script
Total exc time: 38mins
Task time: (18.93 +/- 12.07)secs
Transfer rate: (114 +/- 13)MB/s



Worker in Container
Total exc time: 30mins
Task time: (10.72 +/- 4.79)secs
Transfer rate: (113 +/- 13)MB/s



Container in Worker
Total exc time: 33mins
Task time: (11.89 +/- 3.17)secs
Transfer rate: (114 +/- 13)MB/s



Shared Container
Total exc time: 26mins
Task time: (8.37 +/- 2.54)secs
Transfer rate: (115 +/- 11)MB/s

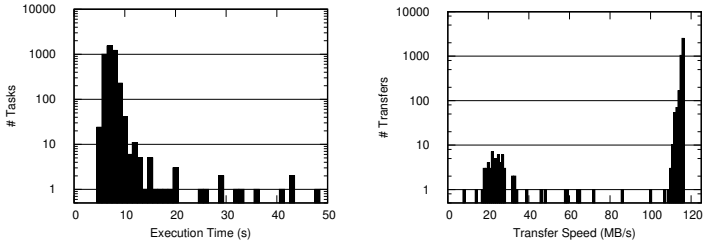


Figure 4.3: Workflow Performance for Each Configuration: In each row of the table, details of the configuration, task execution time histogram and task transfer rate histogram are presented. (1) for configuration details, total execution time, average execution time +/- standard deviation, average transfer rate +/- standard deviation are given (2) for task execution time histogram, The frequencies of tasks execution time in different time intervals are presented. The x-axis is the time intervals and the y-axis is the number of tasks in certain time interval. (3) for task transfer rate histogram, we show the frequencies of tasks' file transfer rate in different time intervals. The x-axis is the time lapse and the y-axis show the number of tasks.

4.6 Conclusions

With the advent of container technology, many of the benefits of traditional virtualization can be achieved at significantly lower cost. However, as I have shown, the costs of instantiating and managing a large number of containers can lead to significant system overheads, and efficiently using container technologies with large computational framework is still an open challenge. In this chapter, I compared four configurations of integrating container runtimes into different components of existing workflow system. I evaluate these configurations by running a large bioinformatics workflows with them and considering tradeoffs between performance, isolation, and consistency. The results show that for large workloads including thousands of tasks, launching and removing large amount of containers cause notable overheads. The overheads can be eliminated by sharing containers across multiple tasks in the cost of losing isolation for each task. More broadly, the design factor derive from this work is that **container runtimes must be adapted to the distributed environment, and the under-lying distributed systems best does the management of containers**, rather than simply leaving it to the user to invoke a container from within each task.

CHAPTER 5

USING CONTAINER ORCHESTRATOR

5.1 Introduction

With the prevalence of container technology, container orchestrators have been widely used as a new means for managing resources on the cloud. In the last chapter, I suggested that container management is better done by the lower layer distributed system. Therefore, rather than developing dedicated container scheduling systems for HTC workflows, a better solution might be running HTC workflows on the leading container orchestrators that have been optimized for managing containers at scale. The work described in this chapter has been published as paper "Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos" in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

Even though container orchestrators have been widely adopted to run the commercial workflow, there are few use cases about using them for HTC workflows. Therefore, there might exist incompatibilities between HTC and container orchestrators. To explore the possibility of running HTC workflows on container orchestrators, I considered four configurations of connecting Makeflow and Work Queue to Apache Mesos:

1. running workflows directly on Mesos with Makeflow;
2. running Makeflow on Mesos with Resource Monitor which measuring and update resource requirement of tasks at real-time;
3. setting up Work Queue framework on the Mesos cluster and running Makeflow on Work Queue;

4. setting up Work Queue framework on Mesos, launching Makeflow on Work Queue, and enabling Resouce monitor to update resource requirements of tasks at real-time.

To evaluate these configurations, I run the SHRiMP workflow with them, which consists of 5079 parallel tasks and transfers 10148 small files during the runtime. I observe that Work Queue can help to avoid resource starvation with fewer user interference and increase the average transfer throughput; and by using Resource Monitor, the resource usage has been greatly increased. Through this work, I summarize three design factors: **Garbage Collection** – timely garbage collection (i.e. removing intermediate data that is no longer needed) is necessary given the massive amount of intermediate data generated by HTC workflows; **Network Connection** – excessive network connections should be avoided considering the large amount of small transmissions during the runtime; and **Resource Management** – resource management mechanisms should be customized based on the characteristics of the objective workflow.

5.2 Challenges with Container Orchestrators

HTC workflows from different domains have several common features:

1. tasks of HTC workflows might have intricate dependencies, large workflows often group tightly coupled tasks into the same phase and execute each phase sequentially;
2. HTC workflows often consist of heterogeneous tasks that can be either short (i.e. seconds) or long (i.e. hours), which requires sophisticated resource management mechanism;
3. HTC workflows usually generate a hundred gigabytes of intermediate data, which can result in high I/O and storage pressure;
4. resource requirement of tasks are usually unknown in advance;

These features pose five challenges to users who plan to scale up HTC workflows on container orchestrators.

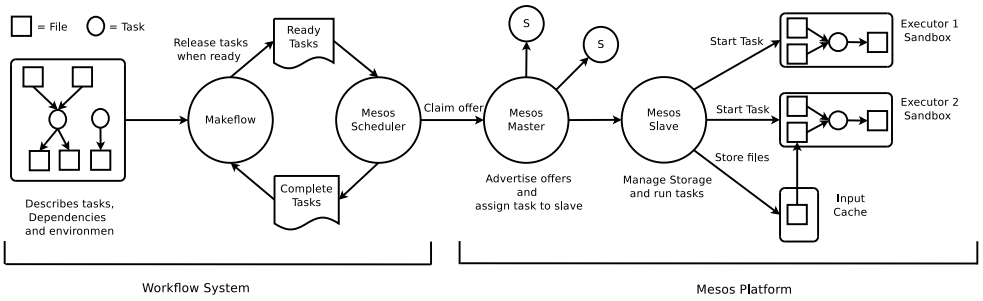
First, the default framework scheduler of Mesos cannot synchronize the workflow status between Makeflow and Mesos. Thus the workflow scheduler must be aware of the completion of tasks on Mesos side, and inform Makeflow to dispatch ready tasks of the next phase.

Second, as many tasks are running concurrently during runtime, an enormous amount of data fetching requests are generated simultaneously, which may results in network congestion on the client-side. When a Mesos agent starts running a new task, an internal fetcher process tries to fetch inputs of the task simultaneously. As many tasks are running concurrently, and all input files of a workflow are normally located in the same host, the host has to handle hundreds of fetch requests. Thus the fetching process may become the bottleneck of the whole system.

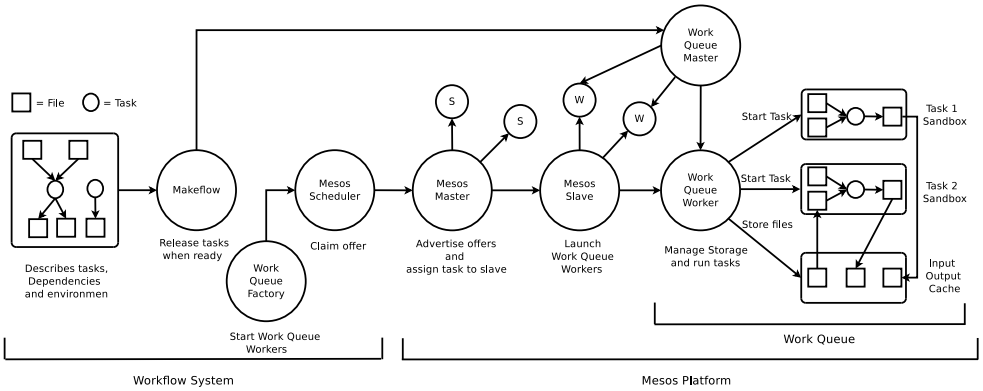
Third, delayed garbage collection can cause disks to be filled up quickly. By default, rather than sending results back to the user after the tasks complete, Mesos agents temporarily store results in executors' sandboxes and mark them as garbage. By default, Mesos agent collects garbage in one week. Even though delay time can be set as short as one day, it is still too long for HTC facilities. In my experience, five runs of SHRiMP workflows on Mesos cluster that has 26 nodes can produce 83GB to 119GB of intermediate data on each node in just one day, and there are usually far more than five workflows being launched on the cluster every day. Another option to reduce the garbage collection cycle is setting a threshold of maximum disk usage. However, the threshold can only be set on a cluster level, which affects other workloads, and some workloads might prefer to keep results on the cluster.

Fourth, as HTC workflows might contain long-running tasks that occupy multiple resource slots for a long time, other workloads running on the same cluster might suffer from resource starvation and the rules of fairness between workloads can be broken.

Fifth, the core idea of Mesos's resource scheduler is the two-level resource match-



(a) Makeflow on Mesos



(b) Makeflow and Work Queue on Mesos

Figure 5.1: System architectures

ing, which matches resource requests proposed by users with resource offers rendered by computing nodes. This mechanism requires users to specify the resource requirement of each container. However, users of HTC workflows usually do not know the resource requirement of tasks in advance. Even though experienced users might be able to provide a coarse-grained prediction, it is far from optimal.

5.3 Proposed Solution

To cope with above challenges, I consider four configurations for launching scientific workflows on Mesos with Makeflow and Work Queue. In order to make the best use of Mesos's two-level scheduler, in two of the configurations, I enable the Resource Monitor of Makeflow, which keeps monitoring the resource consumption of each task and provide a real-time resource usage evaluation.

5.3.1 Makeflow and Mesos

The first configuration is connecting Makeflow to Mesos directly. Shown in figure 5.1a, I develop the *Makeflow Mesos Scheduler (MMS)* to connect them. During the runtime, Mesos executes each task with an independent executor on an available agent. Next, two daemon threads are spawned by Mesos scheduler, one is responsible for polling the file that has information of ready tasks, and another one starts an HTTP server for handling file fetching requests from executors. Since some of the tasks share the same inputs, to mitigate the network pressure, I cache inputs on Mesos's agent.

Figure 5.1a shows how does *MMS* work: i) Makeflow writes the information of ready tasks to a file; ii) the task monitor poll information of ready tasks; iii) *MMS* sends resource requests to the Mesos master; iv) Mesos agents register resource offers with Mesos master; v) Mesos master advertises resource offers to the workflow; vi) scheduler assigns offers to proper tasks, and launches an executor on agents that provide offers; vii) executors retrieve inputs from client or cache directory if exist; viii) executor run tasks; ix) after tasks complete, executors send output URIs to scheduler; x) scheduler retrieves outputs from executor's sandbox and deletes the outputs; xi) scheduler writes the information of finished tasks to file; xii) Makeflow keeps checking whether there are new finished tasks and mark them as completed.

This configuration is straightforward and resolves the first three design challenges. It uses one daemon thread with scheduler and one sub-process with Makeflow to synchronize the workflow status between Makeflow and Mesos. For bandwidth issue, I implement an HTTP server with a thread pool on the scheduler side, which handles each fetching request by an independent thread. The HTTP server limits the number of threads to 30 and maintains a request queue to cache the incoming requests when no thread is available. To avoid resource starvation, there exist two options, i) running long tasks locally, or ii) limiting the resource quota for frameworks that have long

tasks. The first option will not work if target workflows contain many long tasks, while the second option requires users to extend the default resource allocator, which is not trivial. To prevent disks from being filled up quickly, I implement a customized executor, which deletes intermediate data after they are retrieved.

The remaining hurdle is, how to provide accurate task resource requirements. Imprecise resource requirements can result in long workflow execution time or low cluster resource usage.

5.3.2 Makeflow, Work Queue, and Mesos

The second configuration I tried is using Work Queue as the workflow execution layer between Makeflow and Mesos (see figure 5.1b). The main idea of this setting is relying on Work Queue factory, which sets up Work Queue framework on Mesos.

The configuration works as follows: i) the Work Queue factory creates a Work Queue master; ii) the Work Queue master is linked to the Makeflow through a catalog server; iii) the Work Queue master writes the ready workers' information to a file; iv) the *MMS* keep polling the text file, get information of ready workers; v) the Mesos scheduler submits tasks of launching workers for Mesos master; vi) Mesos agents render resource offers to the Mesos master; vii) the Mesos master advertises resource offers to the Mesos scheduler; viii) the Mesos scheduler match resource offers with proper workers, and then launches workers on agents provided offers, each worker is treated as a task and run in an executor; ix) Work Queue workers work for the master, execute tasks of the workflow, x) the Work Queue master informs Makeflow about the completion of tasks; x) after the workflow completes, workers are deactivated; xi) status monitor of the worker keeps checking the list of deactivated workers and informs the factory to remove them.

Compare to the first configuration; this configuration addresses the issue of resource starvation by limiting the resources quota of each workflow. Specifically, work-

ers are run with fix amount of resources. As Work Queue extends the semantics of Makeflow, Mesos is transparent to Makeflow and Work Queue can delete all intermediate data for Makeflow, the default executor can work just fine.

However, this approach creates a new problem: how many resources does each worker require? If the resource requirement of each task is unknown, how many tasks a worker can run parallelly? Considering two harmful use cases: few tasks are run on a worker, which can lead to inefficient resource usage and reduced throughput; too many tasks run on a worker, which can result in resource contention and reduced performance.

5.3.3 Enable Resource Monitor

To pair up resource offers with tasks, both of the above configurations rely on resource requirements provided by users. However, it is rarely possible for average users to provide accurate resource requirements. To remedy this problem, I run the Resource Monitor with the Makeflow to measure and update task resource requirements in real-time. The system works as before, except that each Makeflow task is run with a Resource Monitor thread. Initially, tasks use all available resources in a worker, and the peak resource consumption of tasks are collected. When resource consumption of a completed task is reported, task resource requirements in the same category are updated. With the known resource requirements, the Work Queue worker is able to run multiple tasks concurrently. Following is a Makeflow file includes rules of Resource Monitor.

```
.MAKEFLOW CATEGORY local_split

subseq.1 .. subseq.5079: seq.inp splitreads.py
    LOCAL python splitreads.py 5079 seq.inp

.MAKEFLOW CATEGORY remote_map
.MAKEFLOW MODE MIN_WASTE

output.1: seq.target subseq.1 rmapper
    ./rmapper subseq.1 seq.target > output.1
    ...
output.5079: seq.target subseq.5079 rmapper
    ./rmapper subseq.1 seq.target > output.1

.MAKEFLOW CATEGORY local_combine

output: output.1 .. output.5079 combine.sh
    LOCAL ./combine.sh
```

Above Makeflow file classify tasks into three categories: i) *local_split*, which contains one task that splits the input sequence into 5079 sub-sequences and run locally without resource reinforcement; ii) *remote_map*, which consists of 5079 tasks with each aligning a sub-sequences to a portion of the target sequence, tasks of this phase run on Mesos with the *MIN_WASTE* mode enabled, which starts a reinforcement loop that uses a small portion (100 by default) of tasks to collect resource consumption data and update resource requirements of waiting tasks dynamically; (3) *local_combine*, which combines all the results generated by tasks from the second

phase and generates the final output.

5.4 Experimental Evaluation

5.4.1 Experimental Setup

To measure the performance of the above four configurations – i.e. Makeflow and Mesos; Makeflow, Resource Monitor and Mesos; Makeflow, Work Queue and Mesos; Makeflow, Work Queue, Resource Monitor and Mesos. I run each configuration with the SHRiMP workflow on the cluster which consists of twenty-four 8 core Intel Xeon E5620 CPUs each with 32 GB RAM, 12 2TB disks, 1GB Ethernet, running Red Hat Enterprise Linux 6.8 with Linux kernel 2.6.32-642.6.1.el6.x86_64 and Mesos 0.26.

To stress the problem of low resource usage caused by inaccurate resource estimation. I assume that a greedy resource plan is provided initially, which requires 4 CPUs, 5120 MB of memory and 5120 MB disk for each task. For the same reason, when using the Work Queue, I assign 4 CPUs, 5120 MB of memory, and 5120 MB disk to each worker.

TABLE 5.1

PERFORMANCE SUMMARY OF EACH CONFIGURATION

	Makeflow, Mesos	Makeflow, Mesos, Resource Monitor	Makeflow, Mesos, Work Queue	Makeflow, Mesos, Work Queue, Resource Monitor
Total Exec Time (Hours)	11.17	6.7	8.97	5.37
Average Task Exec Time (Seconds)	408	445	327	355
Average Transfer Rate (MB/S)	43.11	26.88	106.87	104.66
Average CPU Usage (#used/#allocated)	0.500	0.976	0.501	0.975

5.4.2 Results and Analysis

Table 5.1 presents four system metrics: workflow execution time, average task execution time, average transfer rate between the Makeflow and the Mesos agent, and the average CPU usage. To illustrate how different data transmission mechanisms can affect the performance of the workflow. I include the data transmission time as part of the task execution time. Figure 5.2 presents the task execution time histogram and transfer throughput histogram with each row represents a configuration respectively. The first column gives a histogram of individual task execution times of 5079 tasks and the second column include histograms of transfer rate. Figure 5.3 shows the histograms of the number of cores being allocated/used. The first column gives the complete CPU usage, during the second column zooms in and illustrates the CPU usage during the first 30 minutes.

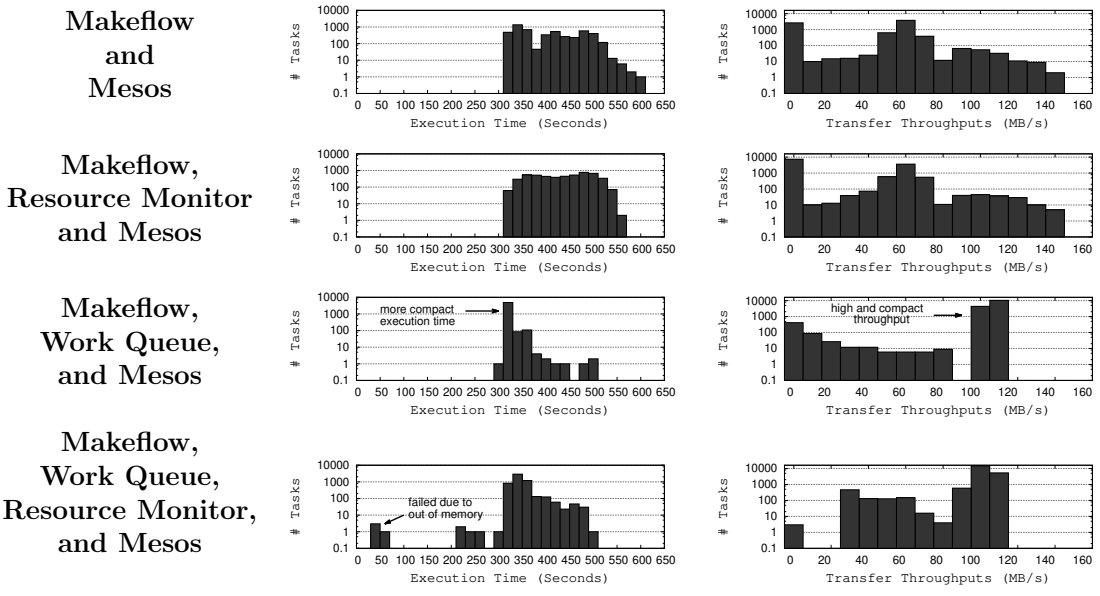


Figure 5.2: Task Execution Time and Transfer Rate, In each row of the table, task execution time histogram and transfer throughput histogram of each configuration are presented.

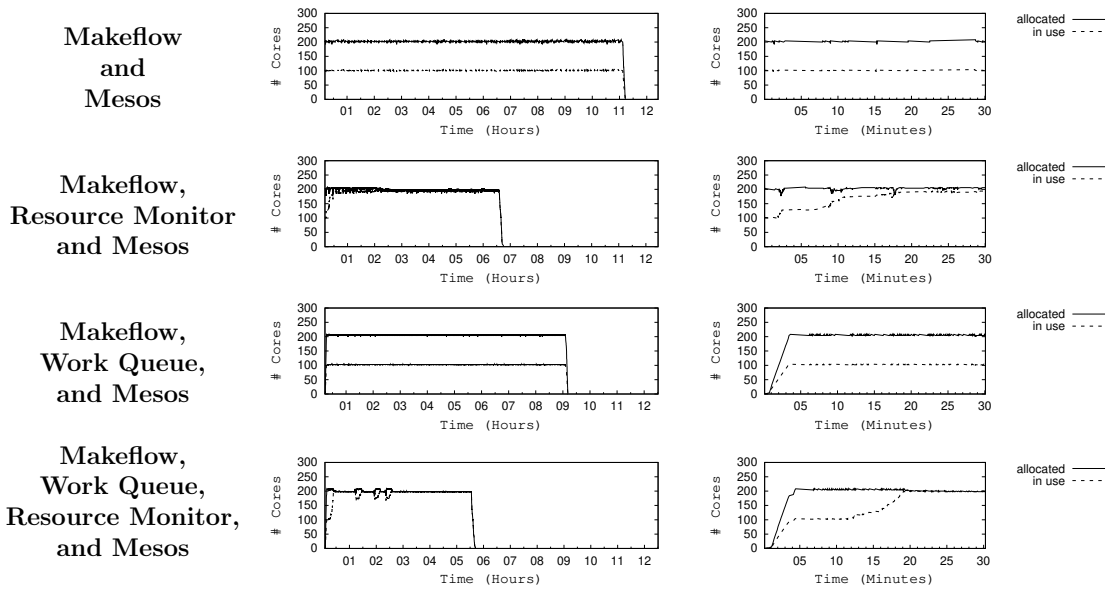


Figure 5.3: CPU Usage Rate, Each row shows the complete CPU usage timeline, and the CPU usage timeline of the first 30 minutes are presented. The total number of available cores is 208. The solid line depict the number of allocated cores, and the dashed line depict the number of cores in use.

As shown in table 5.1, running Makeflow directly on Mesos has the longest workflow execution time (11.17 hours) with a low average CPU usage (0.500). Shown in the first column of the first row in the figure 5.3, during the runtime, only half of the allocated cores are in use. The average task execution time is 408 seconds, which is longer than the configurations using Work Queue. This is due to the repetitive setting up of TCP connections. After Mesos master assigns a task to an executor, the file fetcher fetches each input file of the task with an independent HTTP request. Moreover, after the task is complete, the scheduler retrieves each output file with an HTTP GET request. As there are an enormous amount of small files (i.e. 10150 small files with each file has a size range from 70 bytes to 1 MB) transferred during the runtime, TCP connections are repetitively set up, which also results in lower average transfer rate (i.e. 43.11 MB/S) comparing to other configurations.

As expected, by using Resource Monitor, the resource usage is increased from 0.5000 to 0.976, which also shorten the workflow execution time (6.7 hours). Shown in the second column of the second row in the figure 5.3, the number of cores in use are gradually increased during the first 20 minutes, and finally reach the number of allocated cores. Therefore, even though the average task execution time is still long, the overall performance is improved by 1.67×. By using Work Queue, the average task execution time is shortened. Work Queue worker reuses a single TCP connection to handle all the data transmission. Even though this configuration achieves a better overall performance, it still pays the penalty due to the waste of resources. I remedy this problem by running the Work Queue with Resource Monitor, which further reduces the workflow execution time to 5.37 hours.

5.5 Conclusion

In this chapter, I exploit the possibilities of running workflow systems on container orchestrators. I list five design challenges and try to resolve them by using four configurations, which combine Makeflow, Work Queue, and Resource Monitor in different ways. To compare the four configurations, I run the SHRiMP workflow. The experimental results show that by using Work Queue and Resource Monitor, both performance and resource usage are improved by up to 2×. Consequently, I conclude that when running HTC workflows on container orchestrators, three important factors that need to be considered are: **Garbage Collection**, **Network Connection**, and **Resource Management**.

CHAPTER 6

DISTRIBUTED CONTAINER RUNTIME

6.1 Introduction

So far, I have attempted to integrate container runtime into workflow systems and bridge workflow systems to container orchestrators. In this chapter, I will show how to adapt popular container runtime to the distributed environment in general, which can then benefit workflow systems running in it. The work described in this chapter has been published as paper "Wharf: Sharing Docker Images in a Distributed File System" in *ACM Symposium on Cloud Computing*.

Docker as one of the most popular container runtime, has been widely deployed across public clouds and private clusters. Docker provides a set of interfaces, implemented by the Docker *daemon*, which allow users to conveniently package an application and all its dependencies in *images* and to start containers from these images. To facilitate image storing and sharing, Docker provides a *registry* service which acts as an image repository. Daemons can push/pull images to/from the registry and run them locally.

In Docker, each daemon process is *shared-nothing*, i.e. it pulls and stores images in local storage and starts containers from the local copies. This design introduces a tight coupling between the daemon and the node's local storage. However, considering the rate at which containers start in cloud environments [44] and the low startup latencies required for micro services [90], this tight coupling leads to four major problems: i) if containers start from the same image on different nodes, the image

exists on all of the nodes, this wastes storage space; ii) large-scale applications often consist of thousands of identical tasks, which are distributed across a large number of nodes, to run such an application, each node has to pull the same image from a Docker registry to its local storage, this wastes network bandwidth and increases the application startup time; **(iii) while only** 6.4% **of the image data is read by containers on average [57], each node pulls the *complete* image from a registry, which further wastes storage space and network bandwidth;** (iv) some clusters, e.g., in HTC environments, only offer shared storage while compute nodes themselves are diskless [54, 104], which prohibits running containers in such environments.

To solve these problems and **exploit the benefits of a highly scalable shared storage layer [78]**, I suggest that the architecture of the container runtime should be able to provide a way of storing images in a shared file system, and each daemon should only maintain the minimum necessary private state. In addition, I present Wharf, a system for efficiently serving container images from a shared storage layer such as NFS [80] or IBM Spectrum Scale [19]. Wharf enables distributed Docker daemons to *collaboratively* retrieve and store container images in shared storage and create containers from the shared images. **Wharf significantly decreases network and storage overheads by holding only one copy of an image in central storage for all daemons.** Designing Wharf requires careful attention to the semantics of operations on container images to ensure consistency, scalability, and performance.

Wharf exploits the structure of Docker images to reduce the synchronization overhead. Images consist of several *layers* which can be downloaded in parallel. Wharf implements a fine-grained *layer lock* to coordinate access to the shared image store. This allows daemons to pull different images and different layers of the same image in parallel and therefore increase network utilization and avoid excessive blocking. Wharf splits the data and metadata of each daemon into *global* and *local* state to min-

imize the necessary synchronization. **Additionally, using the layer lock, Wharf ensures that only consistent layer state can be seen by each daemon and prevents the entire cluster from failing when single daemons fail.**

Wharf is designed to be transparent to users and allows to reuse existing Docker images without any changes. Wharf is independent of the distributed storage layer and can be deployed on any storage system, which provides POSIX semantics. Wharf currently supports two common Docker copy-on-write storage drivers – i.e. `aufs` [7] and `overlay2` [32]. Furthermore, a generic principle I learned from Wharf is that when using container runtime in distributed environment, one should consider optimizaing **Image Management** mechanism by using distributed storage.

6.2 Design

In this section, I discuss how to run Docker on distributed storage. I first describe the naive approach, which is inherently supported by Docker, and discuss its shortcomings (subsection 6.2.1). I then introduce the design goals for a native, efficient integration of Docker with distributed storage (subsection 6.2.2).

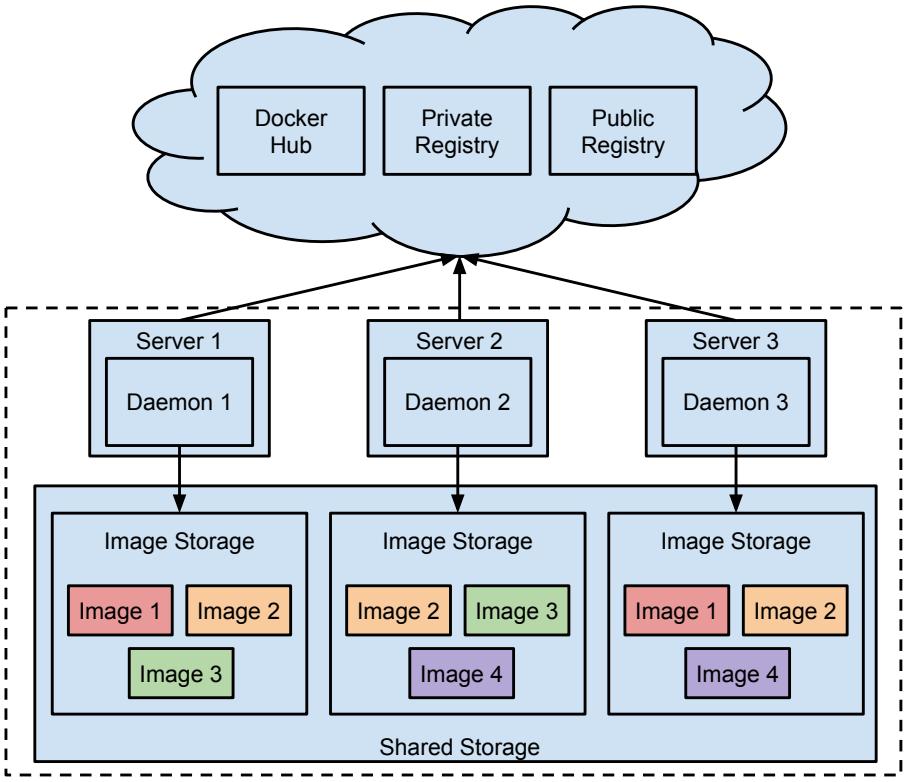


Figure 6.1. Docker on distributed storage, naive solution

6.2.1 Naive Solution

In its current deployment mode, Docker assumes that local storage is used exclusively by a single daemon. The daemon uses this storage to store image and layer data, and metadata on the state of the daemon and its running containers. Hence, two daemons cannot operate on the same storage as they would override each other's state. This does not only hold for multiple daemons accessing the same shared storage but also for multiple daemons on the same host, accessing the same local storage.

The simplest way of enabling Docker to run on distributed storage is to partition the distributed storage. Each daemon receives its own partition where it can store all of its state and image data (see figure 6.1). The partitioning can be physically

done as part of the storage or simply by assigning each daemon a different directory. This approach is supported by Docker today without any additional implementation effort, as long as the underlying storage exposes a POSIX interface.

While this design is simple, it comes with several shortcomings: i) it does not solve the problem of redundant pulls during a large-scale workload, each daemon still has to pull an image separately and store it in its partition, which leads to both network and storage I/O overhead and increased container start-up latencies; ii) it over-utilizes storage space because a copy of the image has to be stored for each daemon, as image garbage collection is still challenging, it is a common problem for Docker users to run out of disk space due to unused images/containers not being removed, which can quickly lead to high storage utilization; iii) image pull latencies can be much higher – up to 4× longer in our experiments – compared to Docker on local storage, due to extra data transfers between the daemon and the distributed storage.

6.2.2 Design Goals

The above issues suggest that the naive approach is not sufficient to successfully integrate Docker with a distributed storage layer. Based on the shortcomings of the naive solution, I identify five main design goals for a native integration approach.

1) Avoid redundancy. To efficiently run on distributed storage, a native solution needs to avoid redundant data transfers. If an image is required by multiple daemons, it should only be retrieved and stored once. Additionally, all daemons should be aware of the existing layers. If an image requires layers that are already present in the distributed storage, only the missing layers should be pulled.

2) Collaboration. Docker daemons should work together collaboratively to ensure correct and efficient operation. For example, images can be pulled in parallel by multiple daemons to speed up pulling as more resources are available. Furthermore,

coordination is necessary to prevent individual daemons from deleting images if they are still in use by other daemons.

3) Efficient synchronization. When multiple daemons access the same storage, locking and synchronization between the daemons is necessary to avoid race conditions. To minimize the impact on container startup times, synchronization should be lightweight and exploit Docker's layered image structure, i.e. accesses should be synchronized at layer rather than image granularity.

4) Avoid remote accesses. As data is now accessed remotely from the shared storage layer, additional latency is induced for read/write operations. This can impact both Docker client calls and the workload running inside a container. In the worst case, when connectivity drops due to network failures, the entire container can stall until the connection is restored. Hence, the amount of necessary remote accesses should be minimized.

5) Fault Tolerance. The system as a whole must stay operational even if one or several individual daemons fail, i.e. a failing daemon should not corrupt the global state and pending operations should be finished by the remaining daemons.

6.3 Wharf

Based on above goals, I design a distributed version of Docker, i.e. Wharf, , which meets the above described design goals. Wharf allows Docker daemons based on the same distributed file system to collaboratively download and share container layers and thereby, reduce storage consumption and network overhead. First, I discuss the overall architecture of Wharf (figure 6.3.1) which addresses design goal 1) and then explain fine-grained layer locking (figure 6.3.2) which implements design goals 2) and 3). Next, I describe Wharf's local write optimization (figure 6.3.3) to address design goal 4) and finally discuss Wharf's approach to fault tolerance (design goal 5) in figure 6.3.4.

6.3.1 System Architecture

The core idea of Wharf is to split the graph driver contents into *global* and *local* state and synchronize accesses to the global state. Global state contains data that needs to be shared across daemons, i.e. image and layer data, and static metadata like layer hierarchies and image manifests. It also includes runtime state, e.g., the progress of currently transferred layers, relationships between running containers, and pulled images and layers. Global state is stored in the distributed file system to be accessible by every daemon.

Local state is related to the containers running under a single daemon, such as network configuration, information on attached volumes, and container plugins, such as the Nvidia GPU plugin, which simplifies the process of deploying GPU-aware containers. This data only needs to be accessed by its corresponding daemon and hence, is stored separately for each daemon. Local state can either be stored on a daemon's local storage or on a separate location in the distributed file system.

Splitting the graph driver content into global and local state ensures that all image-related information is stored once and not duplicated across different daemons. This addresses design goal 1) as no redundant information is retrieved or stored.

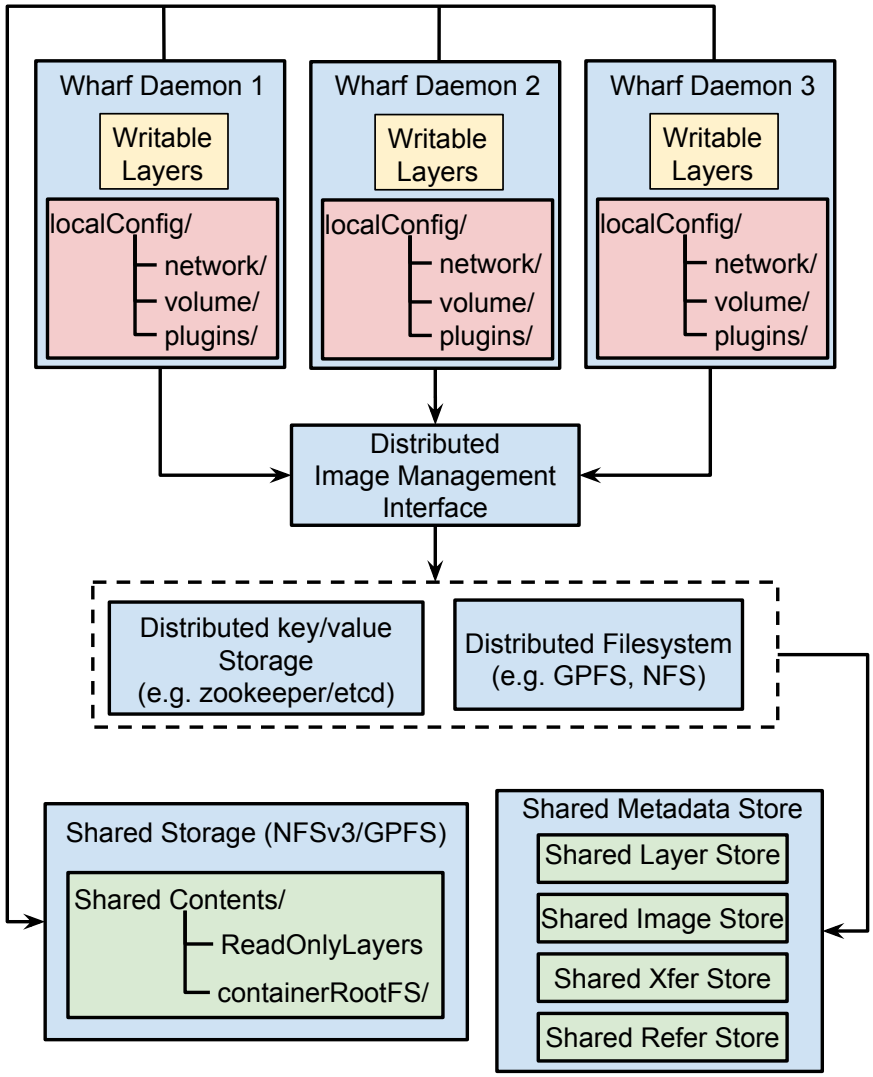


Figure 6.2. Wharf architecture

The architecture of Wharf is shown in figure 6.2. It consists of three main components: Wharf daemons, an image management interface, and a shared store for data and metadata.

1) Wharf daemons. Wharf daemons run on the individual Docker hosts and manage their own local state for their local containers. When a Wharf daemon receives a request to create a container, it sets up the container root file system such that the writable layer is stored at a private, non-shared location and the read-only

layers are read from the distributed file system. Wharf daemons exchange information via updating the global state on the distributed file system but do not communicate with each other directly.

2) Image management interface. The image management interface is the gateway through which Wharf daemons access the shared global state. It ensures that concurrent accesses are synchronized by locking the parts of the global state which are updated by a Wharf daemon. This keeps the global state consistent. The image management interface offers different ways of implementing a distributed lock, e.g., using zookeeper [60] or etcd [17], if these systems are available. If supported by the underlying file system, Wharf can also rely on file locking (`fcntl()` interface) for access synchronization. This approach is highly portable as it does not require any additional external services.

3) Shared store. The shared store hosts all global state and is split into two parts. The *Shared Content Store* contains the data of the readonly layers and the root file systems of the running containers. The *Shared Metadata Store* holds the metadata on which layers and images exist (Shared Layer Store and Shared Image Store), are currently being pulled (Shared Transfer Store), and currently being referenced by containers (Shared Reference Store). Each of these metadata stores can be locked individually. The metadata is concurrently readable by multiple daemons, but can only be updated by one daemon at a time. Any attempt by a daemon to access a layer from the Shared Content Store requires it to first read the Shared Metadata Store and check the status of the layer, i.e. whether it already exists, is currently being pulled, or is referenced by a running container. As write access to the metadata is protected, daemons are guaranteed to have a consistent view of the Shared Content Store.

6.3.2 Layer-based Locking

Design goals 2) and 3) state that locking should be efficient and daemons should collaborate with each other when retrieving or deleting images. Wharf achieves this by exploiting the layered structure of Docker images and implements *layer-based locking* as part of the image management interface. Layer-based locking allows Wharf daemons to write to the shared store by adding single layers rather than an entire image. Read accesses do not have to be synchronized. This means that multiple Wharf daemons

1. can collaboratively pull different layers of an image in parallel;
2. only lock a small portion of the shared store such that unrelated, parallel operations are unaffected;
3. do not pull or remove the same layer at the same time to avoid redundant work;
4. and can read data from the same layer in parallel.

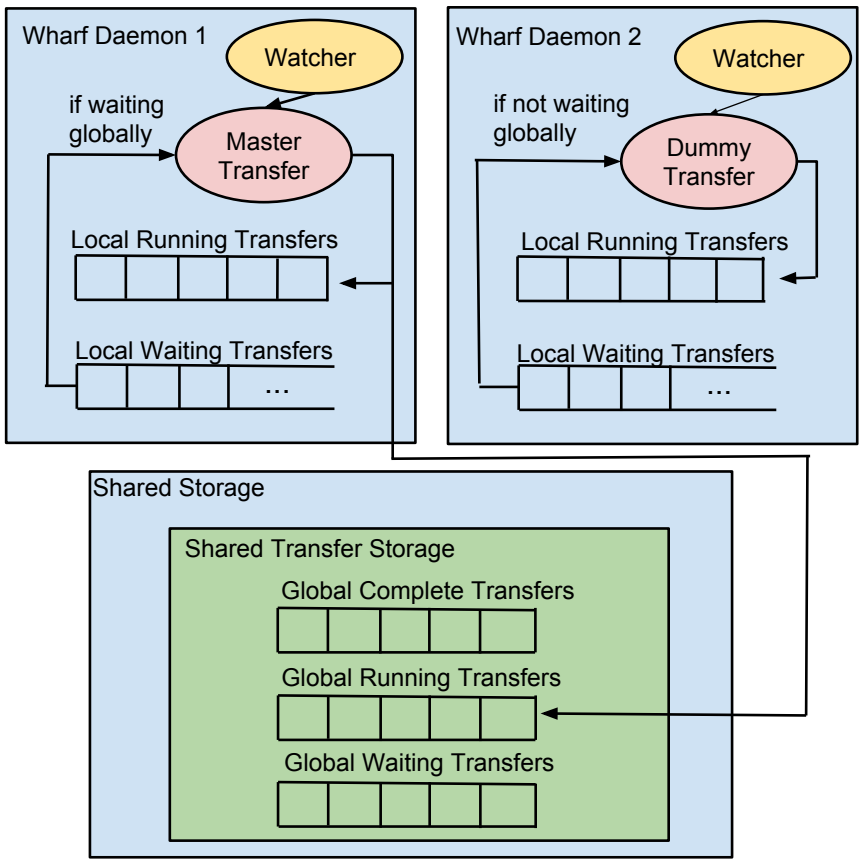


Figure 6.3. Implementation of concurrent layer pulling

Figure 6.3 shows how an image is pulled in parallel under layer-based locking. In Docker, a daemon can start multiple transfers and each transfer is responsible for pulling one layer. Each transfer has two states: running and waiting. By default, each daemon has 3 threads for pulling layers in parallel.

When a daemon receives a layer pulling request, it will create a *Master Transfer* and register it in a local map data structure (*Local Running Transfers*). If all pulling threads are busy, the daemon will add the request to the *Local Waiting Transfers* queue for later processing, once a thread becomes available. If another client tries to pull a layer that is already being pulled by the daemon, the daemon will create a *Watcher* to report the pulling progress back to the client.

In Wharf, pulls not only have to be coordinated across multiple local clients but also across multiple remote daemons (and their clients). Therefore, Wharf uses three additional map data structures, *Global Running Transfers*, *Global Waiting Transfers*, and *Global Complete Transfers*, which are serialized and stored in the *Shared Transfer Store* on the shared storage. Write access to the Shared Transfer Store is protected by a lock.

Each time a Wharf daemon tries to pull a layer, it will first check the Global Running Transfers map for whether the requested layer is already being pulled by another daemon. If not, it will create the Master Transfer and add it to both the Local and Global Running Transfers. Waiting transfers are also registered in both the local and global data structures. Daemons are responsible for removing waiting transfers from the global map once they start retrieving the corresponding layer.

If a Wharf daemon finds that a layer is already being pulled by another daemon, it will create a *Dummy Transfer* and add it to the list of running transfers. The Dummy Transfer is a placeholder to transparently signal the client that the layer is being pulled without actually occupying a thread. Each daemon runs one background thread to periodically check, whether a Master Transfer for a corresponding Dummy Transfer has finished. Therefor, it polls the Global Complete Transfers map where finished Master Transfers are registered.

By using layer-based locking, Wharf accelerates image transfers for large images, which consist of multiple layers and are required by many daemons. It also avoids redundant layer pulling and increases the layer usage as it can now be shared by containers across different daemons. To avoid potential conflicts caused by deleting layers that are currently in use by other daemons, Wharf uses a global reference counter for each layer. A layer can only be deleted from the Global Layer Store if its reference counter is 0.

6.3.3 Local Ephemeral Writes

In a typical Docker deployment the root file system of a container is *ephemeral*, i.e. the changes written to the file system while the container is running are discarded after the container stops. This translates to the removal of the writable layer by the underlying storage driver. User applications frequently create substantial amounts of intermediate data in the root file system which inflates the writable layer. Furthermore, overlay storage drivers, like `overlay2` and `aufs`, copy the whole file to the writable layer even when only a small portion of the file is modified. This leads to significant write amplification at the file system-level compared to the user writes.

Docker typically stores writable and readable layers in the *same* file system. In case of a distributed file system, this translates to a large quantity of ephemeral container data being transferred across the storage network, adding network and remote storage server overhead. Unlike Docker, Wharf stores writable and readable layers in two *separate* locations. Specifically, Wharf puts the writable layers of running containers on locally attached storage while readonly layers are stored in the shared storage so that any daemon can access it. This design addresses design goal 4) by decreasing the amount of writes to the distributed file system.

6.3.4 Consistency and Fault Tolerance

To achieve design goal 5), Wharf requires a *strong consistency model* for its global state and needs to deal with daemon crashes.

Consistency. As mentioned in section 6.3.2, a fine-grained layer lock is used to synchronize the global image and layer state between daemons. While the global state can be read by multiple daemons simultaneously, it can only be written by one daemon at a time. Additionally, global metadata cannot be cached locally at daemons to prevent them from operating on stale data. Exclusive writes combined

with no caching of state provides the necessary strong consistency of the global state across daemons and makes sure that daemons always operate on the same metadata view.

When a daemon performs an operation that requires updating image data, e.g., pulling a new image or layer, it proceeds in three steps: i) It goes through a *metadata phase* in which it locks the necessary part of the global state and registers its actions, e.g., pull layer l_1, if no other daemon is already performing the same action. It then releases the lock; ii) it then continues with a *data phase* during which the actual data is retrieved; iii) finally, if the operation was successful, it again acquires the necessary metadata lock and updates the metadata. As the data phase is always preceded by a metadata phase, no two daemon can perform the same action twice.

Fault Tolerance. Wharf needs to deal with two types of failures: First, Wharf needs to handle the case of a daemon crash while that daemon is still holding a lock. In such a case, the entire system can be stalled if the lock is not released correctly. To avoid such a situation, Wharf uses lock timeouts after which any lock will be released automatically. As daemons only need to acquire a lock to access metadata, the periods during which a lock is required are short and hence, timeouts can be set to low values (e.g., 1 s).

Second, Wharf needs to handle daemon crashes during data phases. If a daemon crashes while pulling a layer, the corresponding image (and all other images that depend on that layer), will never finish pulling. To continue downloading an image after a daemon crashes, Wharf daemons use heartbeats. Heartbeats are periodically sent and the last heartbeat timestamp is stored with the transfer in the Global Running Transfers map. This allows other daemons to check, whether a daemon is still pulling its layer or has crashed. In case of a crash, a new daemon can continue the transfer.

6.4 Implementation

Next, I describe implementation details of Wharf. I first describe how Wharf is able to share global state between the individual daemons (section 6.4.1) and then explain how images can be pulled collaboratively in parallel (section 6.4.2). The described implementation adds approximately 2,800 lines of code to the Docker code base. To run Wharf, the user only has to run the Docker daemon with the `shared-root` parameter set. All other commands can be used without any further changes for the user.

6.4.1 Sharing State

To efficiently share the global state between daemons, Wharf has to deal with two main problems: distributed synchronization for access to the global state and in-memory caching of state in individual daemons.

Docker uses three main `structs` which store the global state:

1. `LayerStore` for information on available layers (readonly and writable layers);
2. `ImageStore` for information on local images (storage backend and image meta);
3. `ReferenceStore`, which includes the references to all available images.

Each daemon keeps an in-memory copy of these data structures and can generate them by reloading the directory that holds the persistent state of the daemon. To accommodate multiple concurrent clients, a Docker daemon protects its in-memory state via a mutex.

Wharf extends the design of Docker by serializing the above data structures and to make them available to all daemons, it stores them in the shared storage. It uses a distributed locking mechanism to synchronize accesses. Read accesses can happen concurrently and only require a read-only lock whereas write accesses require an exclusive lock on the part of the state that should be updated.

By default, Wharf uses the `fcntl` system call to implement the locking. As `fcntl` is supported by most distributed file systems, this approach can be used out-of-the-box without extra software and library dependencies other than what Docker requires. Via Wharf's Image Management Interface, it also allows users to replace the default file-based locking with, e.g., an in-memory key/value store.

As multiple daemons can now update the global state, the in-memory state of an individual daemon can become invalid. Hence, before reading from their in-memory state, daemons have to check whether the global state has been updated. To ensure daemons always have the latest version of metadata before processing any operation, Wharf applies a lazy update mechanism, which only updates the cache of an individual daemon if the operation requires metadata access. Wharf updates its in-memory data structures by deserializing the binary files from the distributed storage and overwrite its in-memory data with the retrieved data.

6.4.2 Concurrent Image Retrieval

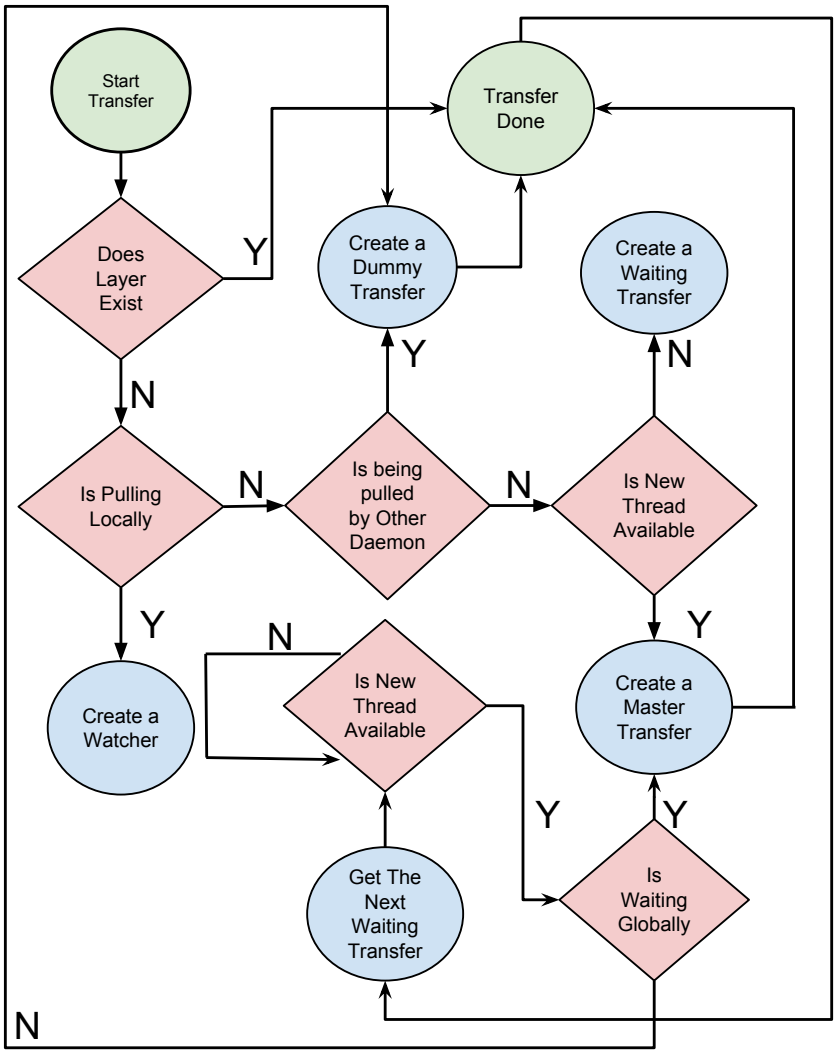


Figure 6.4. Layer pull procedure in Wharf

When pulling an image concurrently, Wharf daemons need to collaborate such that no redundant data is pulled. To enable multiple daemons to pull layers concurrently, Wharf extends the layer transfer model of Docker by adding two new components: the `SharedTransferManager` and the `SharedTransferStore`, to com-

municate between daemons. The `SharedTransferManager` is the distributed version of Docker's `TransferManager` struct while the `SharedTransferStore` is part of the global state and also serialized to the shared storage. Figure 6.4 describes in detail how an image is pulled in parallel by multiple Wharf daemons if they require the same image at the same time.

Each daemon starts by fetching the image manifest and extracting the layer information. The daemons will then dispatch a configurable number of threads to pull the image layers in parallel. A daemon will first check if the layer already exists. If not, it will check whether it is already being pulled by one of its local threads.

In case another thread is already pulling the layer, the daemon will generate a watcher to monitor the progress of the transfer. Otherwise, the daemon will check if the layer is currently being pulled by another daemon on a different host. This information can be obtained via checking the `SharedTransferStore`. If another daemon is found, a *dummy transfer* is generated to monitor the *master transfer*.

If no other daemon is pulling the layer and not all of the daemon's local threads are busy, the daemon will generate the master transfer and dispatch a thread to start downloading the layer. Once the master transfer is complete, the daemon will update the `SharedTransferStore`. The `SharedTransferStore` is accessible by all daemons, thus other daemons with allocated dummy transfers can learn about the completion of the matching master transfer.

In case all local threads are busy pulling other layers, the daemon creates a waiting transfer and pushes it to a local and global waiting transfer queue. Once a local transfer finishes, the daemon will take the next waiting transfer from the local queue and if it is still on the global queue, start the download. Otherwise, that layer is already being pulled by another daemon.

6.4.3 Graph Drivers and File Systems

As Wharf does not have direct access to the underlying block storage, its applicability is limited to the category of overlay drivers. Currently, Docker supports two overlay drivers: `aufs` and `overlay2`[1]. Conceptually, Wharf is compatible with both graph drivers and any POSIX distributed file system. However, running `overlay2` drivers with a distributed file system is less common and, therefore, not well tested. I found that some combinations are currently not operational. I tried two different file systems with Wharf: NFS and IBM Spectrum Scale [19].

NFS. Wharf can work with NFS and both the `aufs` and `overlay2` graph drivers. However, I observed a problem when using NFSv4 and `overlay2` due to the `system.nfs4_acl` extended attribute, which is set on NFS files. OverlayFS is not able to copy the extended attribute to the *upper file system*, i.e. the file system which stores the changed files (ext4 in our case). I believe this is due to incompatibilities between NFSv4 ACLs and the ACLs of the upper file system. While it is possible to mount NFSv4 with a `noacl` option, we found that this is not supported in our Linux distribution.

Spectrum Scale. I also were able to run Wharf on top of IBM's Spectrum Scale parallel file system. While the `aufs` driver was working correctly, I again experienced problems using `overlay2`. I observed that Docker tries to create the upper file system for OverlayFS on Spectrum Scale, even though, local writes to ext4 are configured. This leads to an error ("`filesystem on '/path' not supported as upperdir`"). I currently do not know the exact reason for this behavior, especially because I was able to manually create an OverlayFS mountpoint on Spectrum Scale, but are planning to investigate the problem in the future.

As OverlayFS is part of the mainline Linux kernel and hence, offers better portability, I used the combination of *overlay2* and NFSv3 for our experiments with Wharf.

[1]While there is an older driver based on OverlayFS, Docker recommends to use the new `overlay2` driver.

6.5 Evaluation

To evaluate Wharf, I compare it to two other setups: (i) **DockerLocal**, which uses local disks to store the container images, the corresponding layers, and the writable layers; and (ii) **DockerNFS**, which uses NFS as a storage backend but separate directories to store the data for each daemon.

I run the experiments on an Amazon EC2 cluster using 5 to 20 t2.medium instances. Each instance has 2 vCPUs, 4 GB RAM, and 32 GB EBS disks and runs Ubuntu Linux 16.04 with kernel version 4.4.0-1048-aws. Wharf is based on Docker Community Edition 17.05 and I use this version in all of the experiments.

To avoid network speed variations between a public Docker registry and the daemons, I set up a private registryt2.medium instance.

6.5.1 Pull Latency and Network Overhead

To measure the impact of Wharf on image pull latencies, I run a set of microbenchmarks. I consider five dimensions: (i) the number of layers; (ii) the size of each layer; (iii) the number of files per layer; (iv) the network bandwidth between the registry and the daemons; and (v) the number of daemons. I set up five experiments, and in each, vary one of the above dimensions while fixing the others. I use the default number of 3 concurrent pulling threads for each daemon.

To investigate how the granularity of images influences the pulling latencies for the three different setups, I pull an image that has a **varying number of layers** ranging from 2 to 40. Each layer contains one file. I use 10 Docker hosts and fix the image size to 2 GB. I measure the average pull latencies of daemons. The first row of figure 6.5 presents the results. Wharf shows the shortest average pull latencies (73 s) and lowest overheads for both external (to the registry) and internal (to NFS) network traffic. During each pull operation, all Wharf daemons combined receive on

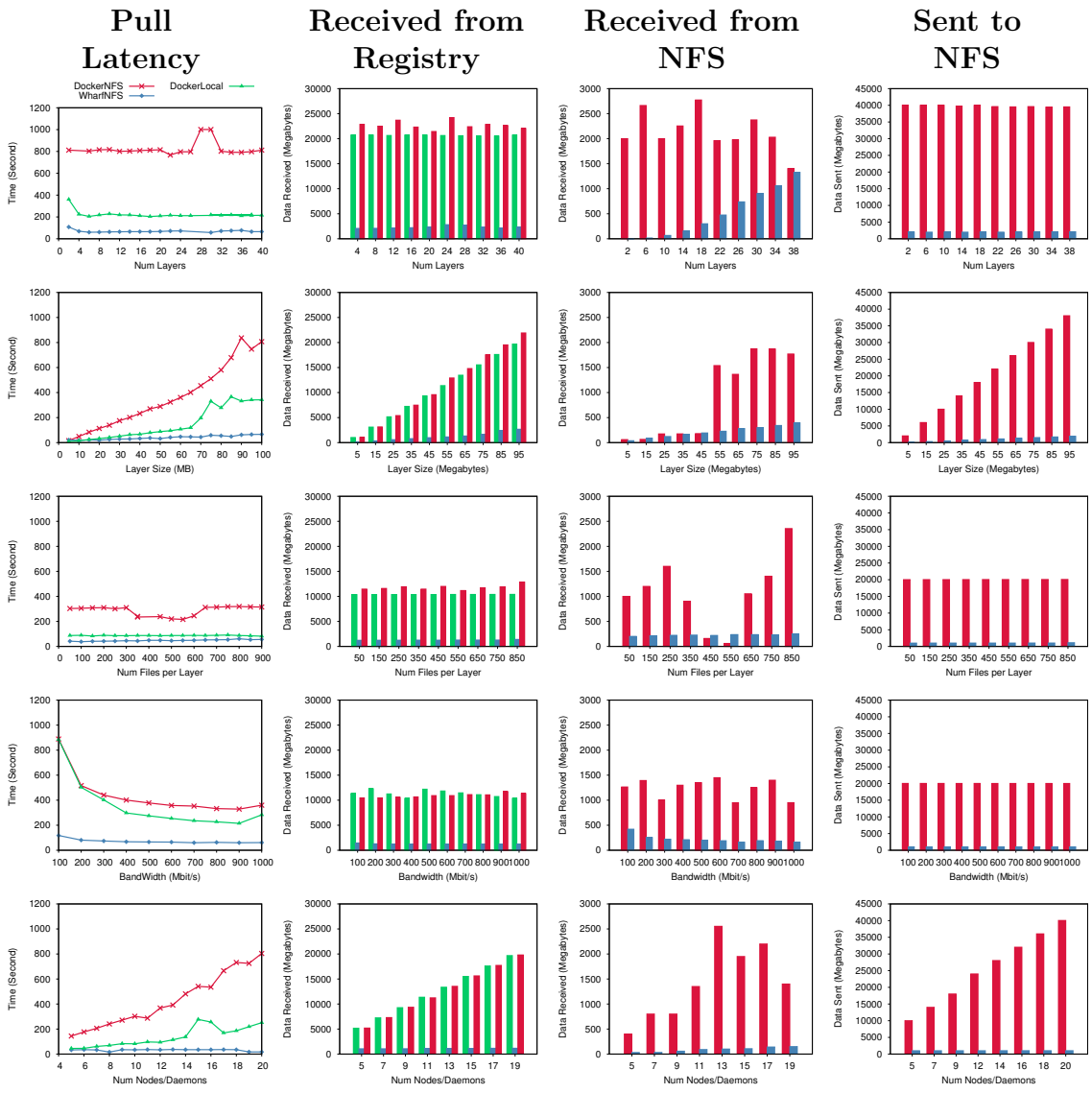
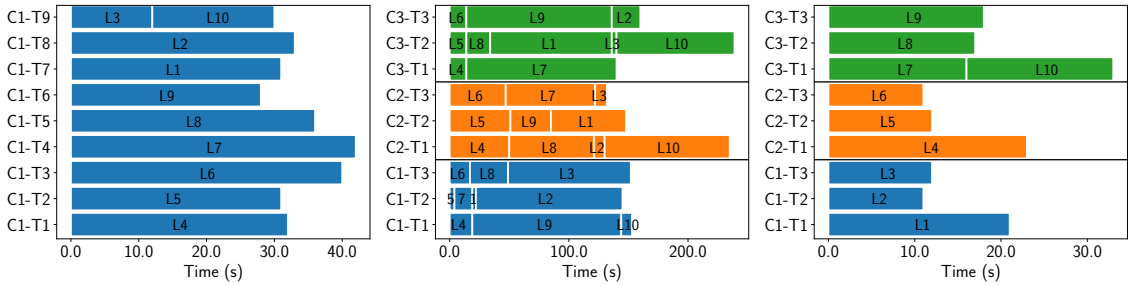


Figure 6.5: Pull latencies and network performance, The figures show image pull latencies and network utilization for different configurations. From the first row to the last row: (i) 10 daemons pull 20 images each, each image size is 2 GB and layers per image number vary from 2 to 40; (ii) 10 daemons pull 20 images with each image containing 20 layers and the size of each layer varies from 5 MB to 100 MB; (iii) 10 daemons pull 18 images and each image has 20 layers with 50 MB per layer and 50–900 files per layer; (iv) 10 daemons pull 20 images from a local registry with the egress network bandwidth varying from 100 Mbps to 1000 Mbps; (v) 20 images are pulled by a cluster with a varying number of nodes, ranging from 5 to 20.



(a) Single Docker daemon with 9 pulling threads

(b) Three Docker daemons with 3 pulling threads each

(c) Three Wharf daemons with 3 pulling threads each

Figure 6.6: Time diagram of layer pulls per client per thread, every rectangle represents a pull of a single layer. *C* stands for *client* and *T* for *thread*. The layer identifiers *L* are placed on top of their corresponding rectangles.

average 2128 MB from the registry, 532 MB from the NFS server, and sent 2049 MB to the NFS server. For a small number of layers, I observe a slight increase in pulling times as in those cases Wharf cannot exploit the available parallelism.

The performance of DockerNFS is limited due to the data transfer between each daemon and the NFS server. It has the longest average pull latencies (953 s) and highest external and internal network traffic, receiving/sending an order of magnitude more data from the registry/to the NFS server compared to Wharf.

DockerLocal generally performs worse than Wharf, due to the fact that when all daemons are redundantly pulling the same image, the network link to the registry becomes a bottleneck. On the other hand, DockerLocal performs significantly better compared to DockerNFS as the retrieved image data does not have to be sent over the network again. Similar to Wharf, I also observe a slight increase in pulling times for DockerLocal when the number of layers is low.

Next, I **vary the size of each layer** from 5 to 100 MB to analyze the impact of different image sizes. I fix the number of layers for the test image at 20 and again use 10 Docker hosts to pull the image in parallel. The results are shown in the second row of Figure 6.5.

For all three setups, I observe an increase in pull latency for larger layer sizes.

This is expected as more data needs to be pulled as the image size increases. Because Wharf only has to pull the image once, its pull latency only increases by a factor of 1.1× overall while DockerNFS and DockerLocal increase by 16× and 7×, respectively. Due to the same reason, the network traffic for DockerNFS and DockerLocal grows significantly faster compared to Wharf.

When looking at small image sizes, I observe that Wharf adds a small overhead compared to DockerLocal and DockerNFS. For a layer size of 5 MB, Wharf takes 21 s to pull the image while DockerLocal takes 9 s and DockerNFS takes 16 s. This is due to the remote accesses and the additional synchronization in Wharf. When moving to a layer size of 10 MB, the overhead disappears and Wharf performs similarly to DockerLocal (18 s and 15 s respectively) while outperforming DockerNFS (49 s).

In early tests with NFS, I observed large pulling latencies due to unpacking the compressed layer tarballs. This is because NFS was using *synchronous* communication by default, i.e. the NFS server replies to clients only after the data has been written to the stable storage. To mitigate this effect, I change the communication mechanism of NFS to *asynchronous*. To see if the number of files in the tarball can affect pulling times even in asynchronous mode, I **vary the number of files per layer** from 50 to 900 while fixing the number of layers at 20 and the total image size at 2 GB.

As shown in the third row of Figure 6.5, the pull latencies of DockerLocal and Wharf are close and are unaffected by the file size. Consistent with previous results, DockerNFS performs worse than the two but also does not show any significant variation due to the number of files per layer.

The default bandwidth between our private registry and daemons is between 1 Gbps and 10 Gbps (as per AWS specification). However, in practice, when connecting to a public registry via a wide area link, bandwidths can vary and throughput can drop significantly. To explore how network bandwidth affects the system, I use

TABLE 6.1

RUNTIME PERFORMANCE SUMMARY

Container Runtime	Total Exec Time	Avg Exec Time (s)	Min Exec Time (s)	Max Exec Time (s)	Data Rev (MB)	Data Sent (MB)
Docker	7 m 26 s	158	31	252	3227	50
Wharf	7 m 47 s	154	46	263	354	768

the Linux tool `tc` to **vary the egress bandwidth of the registry node** from 100 Mbps to 1 Gbps and measure the pull latencies. The pulled image consists of 20 layers of 50 MB each and I use 10 daemons to retrieve the image in parallel. I fix the bandwidth to the NFS server to simulate a realistic scenario in which distributed storage is cluster-local and available via a fast interconnect. The fourth row of Figure 6.5 shows the results.

For the lowest bandwidth of 100 Mbps, the average pull latencies are 118 s, 885 s, and 889 s for Wharf, DockerLocal and DockerNFS, respectively. As bandwidth increases, the pull latencies of DockerNFS and DockerLocal drop quickly, while pull latency of Wharf stays almost constant. As Wharf minimizes the network traffic to the registry, it offers stable performance and is independent of bandwidth variations on the registry connection.

In order to analyze how the cluster size affects system performance, I **vary the number of nodes, and hence the number of daemons**, from 5 to 20. The same 20-layer image is used as in the above experiment for pulling and again measure the average pull latencies of daemons and accumulated network traffic. The results are presented in the fifth row of Figure 6.5.

As the number of nodes increases, the image pull latencies of DockerNFS grow

significantly from 129 s with 5 nodes to 826 s with 20 nodes. While the growth for DockerLocal is less, it is visible and spans from 40 s with 5 nodes to 215 s for 20. In contrast, Wharf shows stable pull latencies regardless of the number of nodes.

The reason is that DockerNFS and DockerLocal pull the image to each node, i.e. as the number of nodes increases, the load increases. Wharf only pulls an image once and is hence independent of the number of nodes. This is also reflected in the network traffic which stays constant for Wharf but increases for the other two setups.

To take a closer look at an image pull operation, I zoom into the pull operation at the layer level and consider three microbenchmarks with **varying pull parallelism**: (i) one Docker daemon with 9 concurrent pulling threads; (ii) three Docker daemons with 3 pulling threads each; and (iii) three Wharf daemons, with 3 pulling threads each. The image pulled consists of 10 layer with each layer ranging from 100 MB to 120 MB.

As shown in Figure **??**, the single Docker daemon with 9 threads completes the task in 40 s. This is similar to Wharf, which completes the task in 33 s. In both cases, the image is only pulled once with 9 parallel threads but as Wharf combines the resources from three machines to retrieve the image, it can improve the pull latency.

The three Docker daemons take 151 s to 256 s to complete the task. As each daemon pulls the image separately, every layer needs to be pulled 3 times which causes network contention and slows down the individual daemons.

6.5.2 Runtime Performance

While sharing images from distributed storage can reduce storage and network overhead, containers have to now access data remotely. This could add additional latency to individual tasks in a workload.

To analyze any potential runtime performance degradation in Wharf, I run the

BWA workflow and compare its performance to DockerLocal. I use Makeflow and WorkQueue to and run them on 10 nodes with 1,000 parallel tasks with DockerLocal and Wharf.

Table 6.1 presents the summary of execution times. On average, Wharf only adds an absolute overhead of 4 s which is equal to 2.6%. Looking at the total execution time, the 5 workflow executions run 21 s slower in Wharf compared to DockerLocal, translating to an overhead of 4.7%. This shows that the remote accesses incurred by Wharf only add a small overhead. As showed in microbenchmarks, the benefit of Wharf also increases with larger cluster sizes and hence, I expect Wharf to perform better compared to DockerLocal at scale.

Additionally, Wharf requires significantly less network resources, retrieving 9.1× less data from the external network. While Wharf introduces more sent network traffic (15× larger compared to Docker) due to NFS traffic, the interconnect to the distributed storage is usually faster compared to an external, wide area network and hence, I do not expect it to become a bottleneck.

Figure 6.7 shows a histogram of the task execution times for one of the runs of the workload. In general, Wharf produces more short running tasks compared to Docker which is due to the shorter image pull latencies. In general, the two system setups have a similar distribution of task execution time, therefore, I conclude that using Wharf does not introduce significant runtime performance degradation.

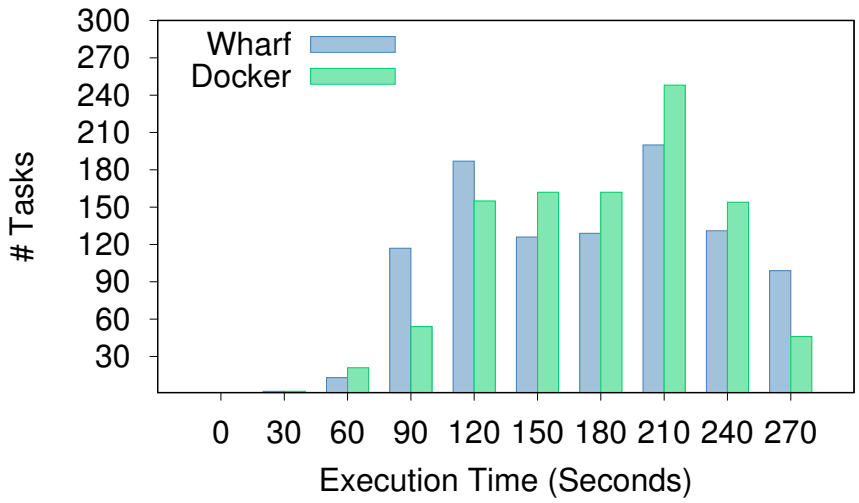


Figure 6.7. Effect of Wharf on task execution time

6.6 Conclusion

In this chapte, I attempt to optimize the container runtime in distributed environment by managing container images through distributed file system. I discuss the drawbacks of a naive solution and the design goals for a native and more efficient approach. I develop Wharf, a shared Docker image store, which fulfills the design goals and allows Docker daemons to collaboratively retrieve, store, and run images. Wharf uses layer-based locking to support efficient concurrent image retrieval and allows to write data to local storage to reduce remote I/Os. The experimenal results show that Wharf can reduce image pull times by a factor of up to 12× and network traffic by up to an order of magnitude for large images while only introducing a small overhead of less than 3% when running container workloads. An important design factor derived from this work is that **Image Management** mechanism can be highly optimized through distributed storage.

CHAPTER 7

AUTOSCALING HTC WORKFLOW

7.1 Introduction

By now, I have exploited how to use container technologies in the HTC environment. However, new virtual machine technologies, like TinyX [70] and unikernel [109], can deliver virtualized environment with reduced overhead, even lower than the container, then why should users choose container over virtual machine? Is there an obvious advantage that makes container out-compete other options? In this chapter, I develop a resource autoscaler for HTC workflows on Kubernetes, which dramatically improves the resource usage of the cluster and cannot be accomplished by using other platforms.

HTC workloads are resource-intensive and have resource usage vary substantially during runtime, making autoscaling essential for efficient resource use. Figure 7.1 shows the three essential components of an HTC system – a workflow manager, a task scheduler, and a cluster manager – and their corresponding autoscaling strategy.

First, A generic autoscaling strategy that can be performed by the cluster manager is scaling cluster via a feedback loop that observes resource load or response time. When any metric is too high, the autoscaler increases the resource pool, which has the effect of reducing the metric. However, such strategy does not work for HTC workflows: high resource use is the normal case, and increasing the resource pool simply allows more jobs to run.

The other two strategies are dedicated to HTC workflows: i) **analyzing the workflow structure** through workflow manager and preserving resources in advance;

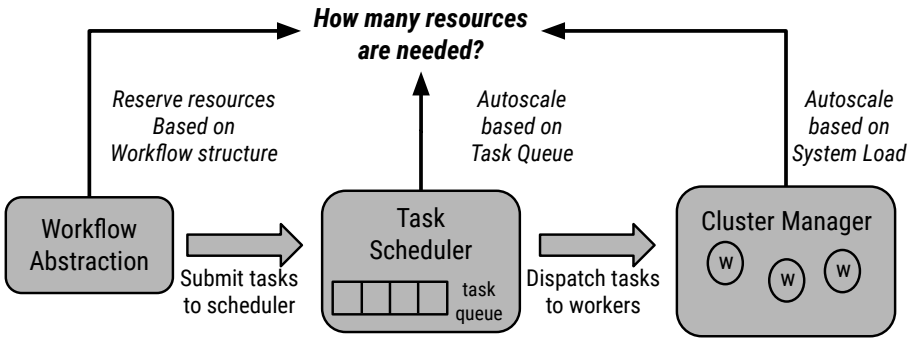


Figure 7.1: Resource Provisioning for HTC Workflows

ii) scaling the resource pool based on the **length of the task queue** reported by task schedulers. However, these two strategies have their own pitfalls which make them sub-optimal. When going with strategy i), resource requirements of tasks are also required in advance, which is not available in most cases. If choosing strategy ii), the resource preparing latency needs to be informed as well, which can only be achieved from cluster manger. As a result, an open challenge in the HTC community is **how to autoscale resource pools accurately**.

To resolve this problem, I refine the autoscaling problem into two sub-problems, i) what is the size of an essential resource unit? And, ii) how many resource units are required by the target workflow? I resolve the first problem by comparing various system settings and observe that if runtime resource monitoring is enabled, and I align each resource unit with an independent node, workflows will achieve the max degree of parallelism with the largest network bandwidth. For the second problem, I suggest that a better strategy that can accurately resize resource pool for HTC workflows require information from all three components – the resource consumption of tasks from the workflow manager, the real-time status of the task queue from a task scheduler, and also the resource preparing latency of the cluster manager. **However, for the virtual machine, to extract application-level knowledge, extra components such as paravirtualization drivers and in-guest controller agents are**

required, which makes it difficult to transcend the boundary between components and collect necessary information.

In contrast, the container orchestrator, like Kuberetens, provides API that allows applications to monitor and control resources by using the container as the basic unit, which blurs the barrier between applications and the infrastructure. Consequently, I extend my idea and develop a middleware called High-Throughput Autoscaler (HTA), which sets up the Work Queue framework on Kubernetes and controls the lifecycles of deployment units based on the progress of running workloads. The experimental results show that compared to the default acutoscaler of Kubernetes, HTA improves resource utilization by 5.6× and shortens the workflow execution time by up to 3.65×. Also, a generic design factor distilled from this work is that when implementing advanced features, one should consider **Cross-layer Cooperation**.

7.2 A Use Case

Before developing a new autoscaling approach, I migrate an existing workflow system to Kubernetes. The new software stack contains three components (figure 7.2).

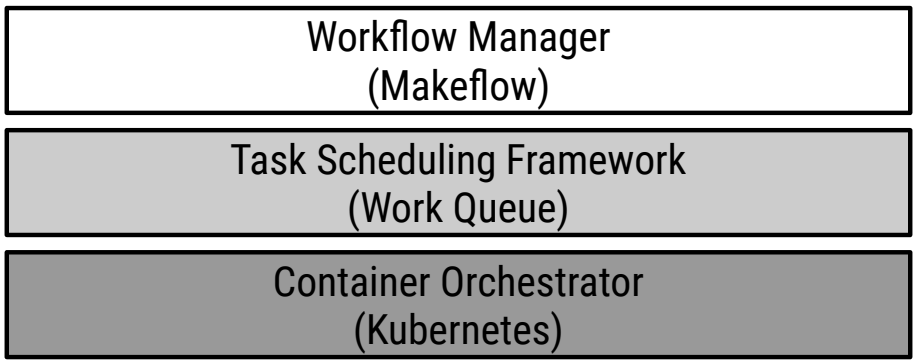


Figure 7.2: The Software Stack of Running HTC Workflows on Kubernetes

1) Makeflow, a workflow manager for defining large and complex workloads. Makeflow's syntax is similar to that of GNU Make, which allows users to describe any workload expressible in a Directed Acyclic Graph (DAG) structure. After the user creates the workload description, Makeflow parses the description and generates an in-memory representation of the workload's DAG structure, which it uses to distribute tasks to different execution framework.

2) Work Queue, a framework for building large-scale master-worker applications that spawn workers across different cloud platforms. Each master program has a worker pool consisting of a set of connected workers. The size of the worker pool varies dynamically with the available computing resources. During runtime, the master finds available workers and assigns tasks to them, and then the worker will arrange data transfer and execute each task it receives.

3) Kubernetes, a container orchestration tool developed by Google which gives the developer the ability to manage distributed applications hosted in containers. Kubernetes allows users to describe resources using different objects. I focus on three of them, i) Pod, which is the primary deployment unit and a mortal object which might fail or restart; ii) StatefulSet, which contains a set of pods and each of them has a unique and sticky identity; iii) Service, which defines the network protocol for accessing the micro-services hosted on a set of pods.

For deploying Work Queue on Kubernetes, there exist several configurations depending on which deployment unit is chosen to manage worker container. As workers will be created and deleted frequently during the runtime, removing workers by deleting the deployment unit that is wrapping them will result in worker containers and tasks running on them be interrupted with the risk of losing intermediate data. Rather than using advanced deployment unit to control the lifecycle of worker container, I intend to control them directly. To this end, I align each worker container with an independent Pod and manage the lifecycle of each worker container through

Work Queue.

7.3 Problems

After determining to align each worker container with an independent pod, I refine the problem of **how to autoscale resource pools for HTC workflows** into two sub-problems: what is the size of each worker-pod (§7.3.1), and what is the amount of worker-pods (§7.3.2).

7.3.1 Size of a Worker-Pod

Typically, HTC workflows consist of many parallel tasks. Without knowing the resource requirements of individual tasks, assigning multiple resource-intensive tasks to a single worker and running them simultaneously may lead to resource starvation. To prevent this, Work Queue assigns only one task to a worker at a time. This setting makes the size of worker critical to the performance of the individual task. Thus, when setting up the Work Queue on Kubernetes, the resource capacity of each worker-pod should be carefully determined.

Assume that the size of the resource pool is fixed, then a fine-grained configuration that has many, small workers will have the ability to run more tasks concurrently, while a coarse-grained configuration that has few, large workers will have a lower degree of parallelism. However, since the outgress network bandwidth of a master is constant, the fine-grained configuration has to share limited bandwidth across more workers with more data movements. This imposes extra network overheads and might cause longer workload execution time. Therefore, which configuration is better depends on whether the target workload is data-intensive or compute-intensive. However, this information is difficult to obtain without running the workload multiple times.

7.3.2 Number of Worker-Pods

Even though the size of worker-pod is determined; the number of worker-pods still need to be figured out. The resource demands of different workloads vary dramatically. Even for a single workload, resource demands change substantially during runtime.

To decide the quantity of worker-pods, one option is using the Horizontal Pod Autoscaler (HPA) of Kubernetes [26]. HPA changes the number of pods based on the ratio between a desired metric value and the current metric. For example, the desired amount of CPU can be calculated through equation (7.1), with $CurrentCPU$ and $CurrentCPUUse$ reported by Kubernetes and the $DesiredCPUUse$ set by users.

$$DesiredCPU = CurrentCPU \times \frac{CurrentCPUUse}{DesiredCPUUse} \tag{7.1}$$

However, the nature of HPA only allows it to make belated responses to potential changes in resource demand. Although this mechanism works well with latency-sensitive micro-services, it does not work for HTC workflows. Applying HPA to HTC workflows may lead to three negative results, i) the cluster scales slowly which misses the peak resource demand; ii) resources are over-provisioned when they are no longer needed; or iii) workloads never scale up to the desired degree.

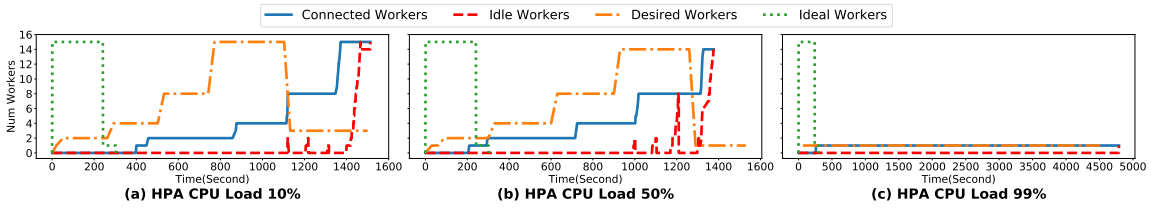


Figure 7.3: Workload Runtime Statistics with Different HPA Target CPU Load

To show the negative results, I run the BLAST bioinformatics workload [8] on

GKE with different HPA target CPU loads that are 10%, 50% and 99% (hereinafter referred as Config-10, Config-50, and Config-99) and the cluster can be scaled up to 15 nodes. The input is split into 200 parts which give us a parallel workload consists of 200 tasks. I assume that the resource requirements of individual tasks are known in advance. Shown in figure 7.5, four dimensions are considered: i) the number of worker-pods connected, ii) the number of idle worker pods, iii) the desired number of worker-pods calculated by HPA, and iv) the number of worker-pods required in an ideal scenario. Config-10 and Config-50 have similar workload execution time, (1294 versus 1304 seconds) and close CPU usage (68.3% versus 65.2%). And both of them have cluster size increased to the capacity limit, i.e. 15 nodes. The main difference is that Config-10 has a more extensive cluster upscaling variation with latency – the latency of transferring from the old scale to the desired scale – than Config-50. This is due to the more considerable disparity between current and target CPU load. By contrast, Config-99 never scales up and results in the workload execution time almost four times longer (4682 seconds) than the other two configurations. In summary, though Config-10 and Config-50 finally scale up to the desired degree, they are still far from the ideal, which is to have the workload complete in 240 seconds. Therefore, the autoscaler reacting to system indicators does not always work for HTC Workflows.

7.4 Proposed Solution

7.4.1 Large Pod with Resource Monitoring

As discussed in section 7.3.1, for an arbitrary workload, it is challenging to decide which configuration to go with. However, if tasks' resource requirements are foreseen, Work Queue will be able to run multiple tasks on a single worker concurrently. The coarse-grained configuration will not only benefit from larger network bandwidth but

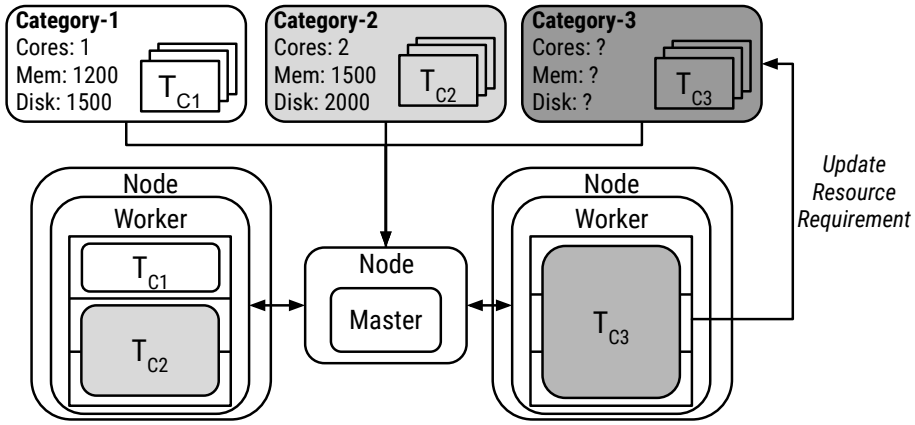


Figure 7.4: One Worker-Pod per Node with Runtime Resource Monitoring

also have a high degree of parallelism as in the fine-grained configuration. To prove this, I run a bioinformatics workload (i.e. BLAST workload [8]), which is made up of 100 parallel tasks with each task having 1.4GB input and 600KB output.

Individually, I consider three setups on a GKE cluster consisting of 5 physical nodes with each node having 3vCPUs, and 12GB RAM. Configuration (a) has 15 workers with each worker occupying 1 vCPU, 4 GB RAM, configuration (b) has 5 workers with each worker occupying an entire physical node and configuration (c) has a similar worker setup as (b) however, resource requirements are known for all tasks in this case.

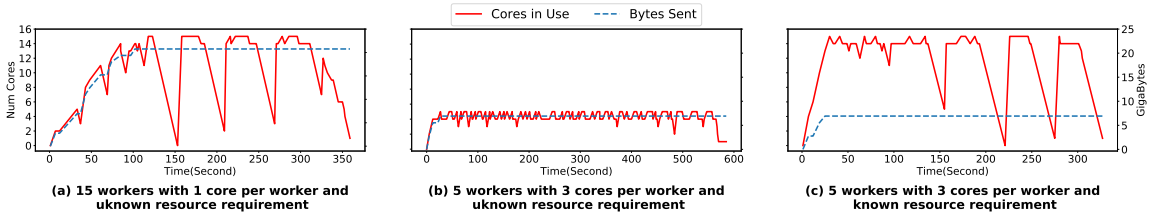


Figure 7.5: Runtime Statistics of Workload with Unknown Resource Requirements

Shown in figure 7.5, when using the fine-grained configuration, the workflow completes in 6 minutes 51 seconds with 278.382 MB/S average bandwidth and 87.21%

CPU usage. If the coarse-grained configuration is used, the workflow completes in 10 minutes 32 seconds with 452.138 MB/S average bandwidth and 32.43% average CPU use. If the resource requirements of tasks are known in advance, the coarse-grained configuration completes in 5 minutes 30 seconds with 466.173 MB/S average bandwidth and 85.73% average CPU usage, which outperforms the other two configurations in terms of both resource utilization and network bandwidth. Therefore, I convert the problem of deciding the size of worker-pods to how to obtain the tasks' resource requirements for arbitrary workloads.

As HTC workflows often consist of parallel tasks that fall into sub-categories with tasks in the same category sharing similar resource requirements, I propose an approach that obtains the resource requirements of tasks by referring to the resource consumption of completed tasks belonging to the same category. Specifically, it contains three steps (figure 7.4), i) tag tasks of the workload by category before execution; ii) for a task with unknown resource requirements, it uses a worker exclusively, has resource consumption measured, and updates the resource requirement of its category; iii) for tasks that belong to categories with known resource requirements, Work Queue will run multiple tasks concurrently on a single worker which has enough resources.

7.4.2 Well-informed Autoscaling

Most HTC workflows are resource-intensive, which often results in a fixed, high system load during runtime despite the number of available resources. Therefore, rather than using a reactive autoscaler that relies on system indicators, a better-informed autoscaling approach that considers system runtime metrics, as well as workload status, is more applicable.

In a queue-based submission model, jobs are queued up and wait for appropriate resource slots. Therefore, the basic idea of autoscaling is adding new resources to

make up for the resource shortage when the job queue length is increasing and removing idle resources vice versa. For any given time point, the resource shortage can be estimated by considering resource requirements and quantity of waiting and running tasks. However, a critical problem of this approach is that the length of the task queue might keep changing while the cluster manager is adding new resources. This can result in resource over-provisioning or under-provisioning. Therefore, an efficient autoscaling approach needs to consider the variation of resource shortage when the cluster manager is preparing new resources.

To illustrate my approach, I define five terminologies, i) Resource In-use (RIU), the amount of resources currently being used by running tasks; ii) Resource Shortage (RSH), the amount of resources desired by the waiting tasks; iii) Resource Demand (RD), the sum of resources in-use and resource shortage; iv) Resource Supply (RS), the amount of available resources supplied by the cluster manager; v) Resource Waste (RW), the amount of ideal resources. During runtime, resource demand is uncontrollable, and there often exists a maximum resource quota depending on user budget. Despite the above factors, an efficient autoscaling approach should **maximize resources in-use and workload throughput; meanwhile, minimize resource waste, resource shortage, and workload execution time**.

Formally, I define a time interval between the time point of submitting new resources request (t_{nr}) and the time point when all new resources are ready (t_{rr}) as a resource preparing cycle. Then resource shortage of the system can be calculated by equation (7.2), specifically, $\Delta RSH(t)$ is the resource amount of newly enqueued tasks waiting for available resources at t, and $\Delta RIU(t)$ is the resource amount of finished tasks at t

$$RSH(t_{rr}) = RSH(t_{nr}) + \sum_{t=t_{nr}}^{t_{rr}} (\Delta RSH(t) - \Delta RIU(t)) \tag{7.2}$$

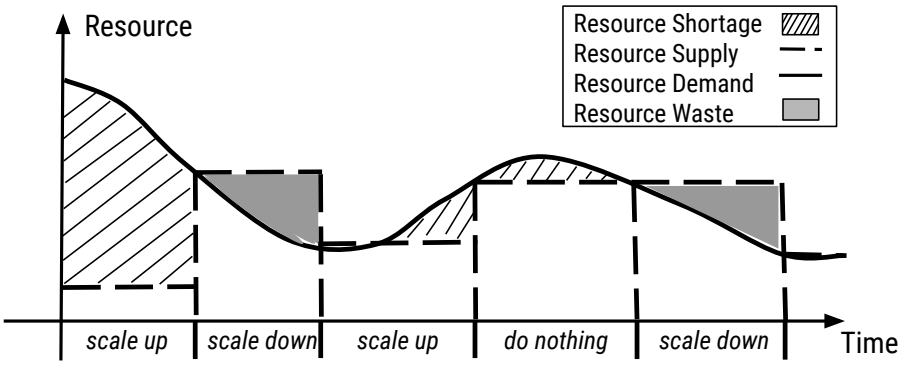


Figure 7.6: An Example of workloads resource relationship on cluster

A potential problem of this approach is, if the variation rate of the resource pool – when and how many resources the cluster manager will add – is unknown, it will be challenging to estimate $\Delta RSH(t)$ and $\Delta RIU(t)$. However, I notice that cluster managers usually process reservation requests in batches thus request submitted in the same batch that ask for the same machine types and container images in the same geographical region should experience similar resource preparing latencies. In other words, resources reserved at the same time should be ready almost at the same time.

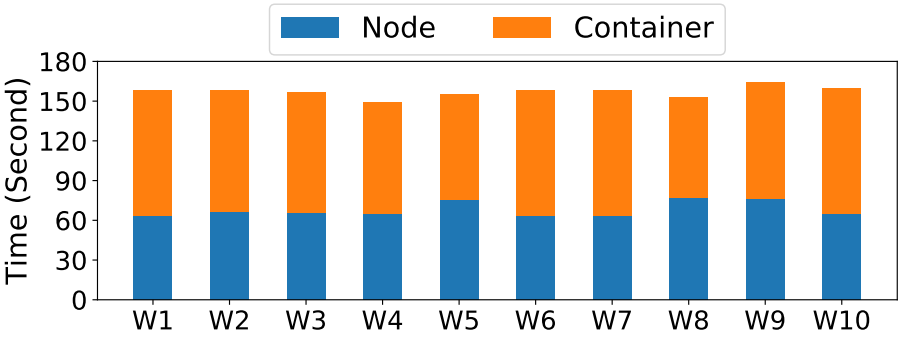


Figure 7.7: An Example of GKE Resource Preparing Latency

To verify this, I measure the resource preparing latency (including machine reservation and container pulling time, see figure 7.7) by creating pods that have resource

requirements that cannot be met by existing nodes. I ran the benchmark 10 times on GKE and found that the resource preparing latency changes little (mean: 157.4 seconds, standard deviation: 4.2 seconds). Therefore, I assume that the size of the resource pool is constant during a resource preparing cycle and simplify the problem of estimating $\Delta RSH(t)$ and $\Delta RIU(t)$ to the problem of simulating the execution of a workload with a fixed resource pool during a given resource preparing cycle.

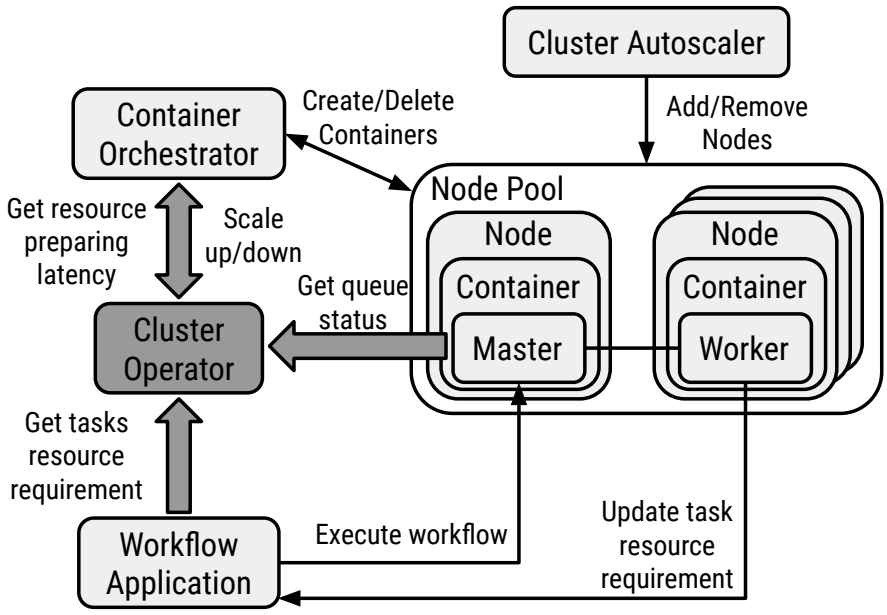


Figure 7.8: A Well-Informed Autoscaling Approach

From this observation, I present a well-informed autoscaling approach (Figure 7.8) which **autoscales the resource pool for HTC workflows by considering resource consumption of completed tasks, the real-time status of the task scheduling framework, as well as the resource preparing time of the cluster manager**. Specifically, this approach contains three steps, i) obtaining the latest resource preparing time from the cluster manager, the resource consumption of com-

pleted tasks from the workflow manager, and the real-time status of the task queue from the task scheduling framework; ii) simulating the workload execution with the current resource pool in a resource preparing cycle and determining the resize action corresponding to different simulation results. (i.e., scale down – there is resource waste, scale up – there is resource shortage, and do nothing – resource supply and demand are in balance) iii) proceeding corresponding action (i.e., scale up/down or do nothing) there might exist pending pods that cannot find nodes which have met their resource requirements or idle nodes that are underutilized. The cluster autoscaler of Kubernetes will add new nodes for pending pods or delete idle nodes, waiting for a resource preparing cycle and repeat i) to iii);

If the above approach is applied, the system should get a resource relationship that looks like Figure 7.7, which has three states corresponding to each resize action (i.e. scale up, scale down, and do nothing).

7.5 Implementation

7.5.1 System Components

I implement the above approach in a new middleware called High-throughput Autoscaler (HTA). HTA deploys Work Queue task scheduling framework on Kubernetes, helps workflow application to submit tasks, and manages the container cluster through the Kubernetes master during runtime. Figure 7.9 shows related system components.

1) Makeflow Kubernetes Operator contains four sub-components, i) *Kube Informer* which will receive notice when registered deployment units (i.e., Pod, Service, and Statefulset) are created, updated, or deleted. It keeps tracking new nodes adding latency and the pulling latency of container image; ii) *Resource Provisioner* which evaluates resource usage and resizes clusters based on the status of task queue,

statistics of completed tasks, and resource preparing latency; iii) a *TCP Client* which sets up a network connection to the Work Queue master, submitting ready tasks, receiving completed tasks to and from the Work Queue master, and getting the task queue status for the resource provisioner; iv) and a *TCP Server* which is responsible for all communication, including task inputs/outputs movement and task information transmission, between HTA and Makeflow. HTC workflows often produce a large number of data transmission during runtime. To decrease network overheads, I keep the connections between Makeflow, HTA, and Work Queue alive during the entire lifecycle of workloads.

2) Container Cluster On GKE, HTA sets up a container cluster to run the Work Queue framework. As discussed in section 7.4.1, I use the configuration of large worker-pods with each pod occupying an entire physical node. Initially, the cluster has 3 nodes [1] and will scale up on-demand later. To avoid loss of intermediate data and ensure a restarted master pod can run on the same physical node with the same identity, I encapsulate the master pod inside a StatefulSet and dump intermediate data into a persistent volume. Since users often start workflow applications locally or from a network namespace different from container cluster, I set up dedicated services for HTA and worker-pods to access the master pod from outside and inside of the cluster.

7.5.2 Resource Preparing Latency

To estimate resource usage, I simulate the workload execution of a period of time equal to latest resource preparing cycle. This requires three pieces of information: i) resource requirement of individual tasks, which can be estimated by referring to complete tasks belong to the same categories; ii) number of waiting and running tasks,

[1] A GKE cluster with a size smaller than 3 nodes might be unreachable during the Kubernetes master node upgrade. To avoid unnecessary disruption, I start with 3 nodes.

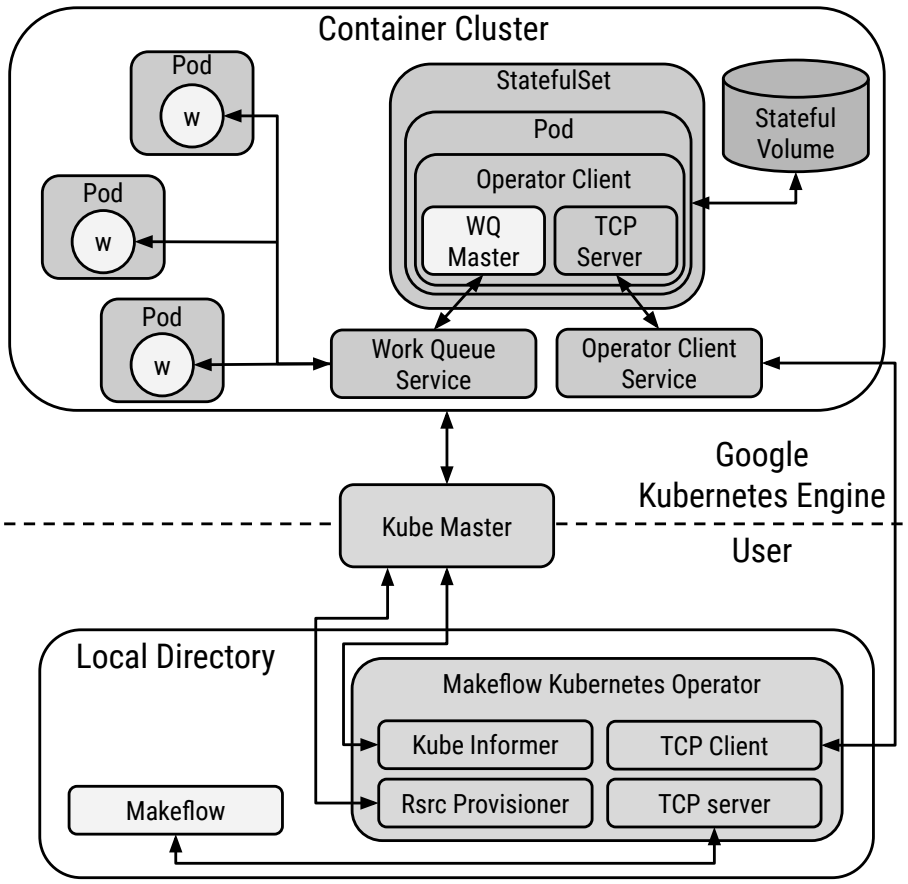


Figure 7.9: Makeflow Kubernetes Operator Architecture

which can be obtained by inquiring Work Queue; iii) resource preparing latency of the cluster manager.

To obtain the latest resource preparing cycle, I use the informer API to track the lifecycle of each worker-pod, which includes four states (see Figure 7.10):

1) No Available Node This state happens when Kubernetes receives requests for creating new worker-pods, but no existing physical node can meet the resource requirement. The worker-pod will stay in `Pending` phase with event `Insufficient Resource`. Then Cluster autoscaler of Kubernetes will detect the pending pods and reserve new physical nodes that can hold all pending pods.

2) No Container Image In this state, the worker-pod will turn into `Pending`

phase with event `Pulling Image`. Meanwhile, Kubelet pulls container images of worker-pods to new nodes. The current version allows users to specify a container image for workloads that have Work Queue worker executable included.

3) Worker-Pod Running This state happens after Kubelet pulling container image. In this state, Kubelet starts worker-pods.

4) Worker-Pod Stopped When the worker process in worker-pod completes, the worker-pod will stop and be removed. In this state, Worker-pod turns into `Succeeded` phase. If, Kubernetes receives a request for worker-pod creation between two synchronization cycle of cluster autoscaler, new worker-pod will be created, and worker-pod will go back to state **3)**, else the node will be marked as underutilized and removed by cluster autoscaler.

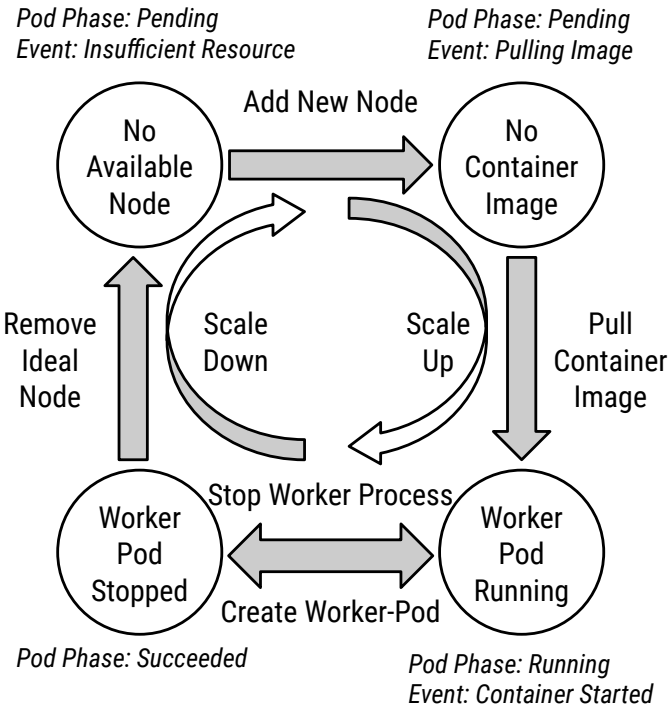


Figure 7.10: Lifecycle of a Worker-Pod

Kubenretes provides powerful API for tracking status of the different deployment units. I implement an informer module, which keeps tracking the lifecycle of each worker-pod and its event. If the creation process of a worker-pod experiences three states – No Available Node, No Container Image, Worker-Pod Running – I will use the time interval between HTA generating creation request for worker-pod and the worker-pod is up as the latest resource preparing latency.

7.5.3 Resource Autoscaling

The autoscaling process includes three stages:

1) Warm-Up stage In this stage, HTA sets up the Work Queue framework on Kubernetes with 3 nodes and start collecting the resource preparing latency. Makeflow submits the first batch of tasks to HTA. Instead of fanning out all tasks at once, HTA sends out only a portion of tasks with one task per category to collect resource statistics of each category.

2) Runtime stage During runtime, HTA periodically estimates resource needs and resizes the cluster on-demand. As shown in algorithm 1, when proceeding resource estimation, the latest resource preparing latency, information about running/waiting tasks, category information that contains tasks runtime data (i.e. resource requirement and execution time) grouped by categories and information of active workers are passed to the function. The estimation function checks the current resource balance, simulates the task dispatching process, and returns the desired scale variation.

If the scale variation is larger than zero, a scaling up action will be applied which creates new worker-pods, and scaling down action will be taken if the scale variation is smaller than zero, which drains worker pods, i.e. stop the worker once all running tasks on it are finished. To avoid system thrashing caused by frequently resizing, time intervals between two resizing actions is always set as the latest resource preparing

cycle.

3) Clean-Up stage Finally, when there are no more tasks that need to run, HTA will receive a notification from Makeflow and drain all workers. Once all tasks are complete, HTA will erase intermediate data, delete all deployment units left on Kubernetes, and send out a notification to the user.

7.6 Evaluation

Resource Autoscaler	**Workflow Runtime** ($second$)	**Accumulate Waste** ($core \times s$)	**Accumulate Shortage** ($core \times s$)
HPA(20% CPU)	2656	51324	34813
HPA(50% CPU)	2480	39353	66611
HTA	3060	9146	40680

(a) Blast Workflow Performance Summary

Resource Autoscaler	**Workflow Runtime** ($second$)	**Accumulate Waste** ($core \times s$)	**Accumulate Shortage** ($core \times s$)
HPA(20% CPU)	6670	159	337737
HPA(50% CPU)	7230	82	357640
HTA	1823	2028	31840

(b) I/O Bound Workflow Performance Summary

Figure 7.11: Workflow Performance Summary

In section (§7.3), I show that with CPU load higher than 50%, HPA would rarely scale up the cluster. Therefore, I compare HTA to two setups: (i) HPA-20%, which use HPA of Kubernetes with target CPU load 20%; and (ii) HPA-50%, which use

Algorithm 1: Resource Estimation Algorithm

Data: $newRsrcLatency$, $runningTasks$, $waitingTasks$, $taskCategoryInfo$, $activeWorkers$
Result: $scaleChanged$, $timeToNextAction$

```
/* simulate the execution of workflow */
rsrcCap = TotalRsrc(activeWorkers)
avaRsrc = AvaRsrc(activeWorkers, runningTasks)
for t = 1; t < newRsrcLatency; t++ do
    completeTasks = TasksCompleteAt(t, runningTasks)
    /* return resource used by complete tasks */
    foreach task in completeTasks do
        avaRsrc = avaRsrc + task.rsrc
    end
    /* simulate task dispatching */
    foreach task in waitingTasks do
        /* no resource available, moving on */
        if AvaRsrc == 0 then
            break
        end
        /* dispatch waiting tasks */
        if task.rsrc < avaRsrc then
            avaRsrc = avaRsrc - task.rsrc
            runningTasks = append(runningTasks, task)
            delete(waitingTasks, task)
        end
    end
end
/* resources are enough, do nothing */
if Len(waitingTasks) == 0 then
    return 0, DefaultCycle
end
/* scale down if there is spare resources */
if avaRsrc ¿ 0 then
    return -NumIdleWorkers(avaRsrc), MaxRuntime(runningTasks)
end
/* scale up, otherwise */
return WorkerRequired(waitingTasks), newRsrcLatency
```

HPA of Kubernetes with target CPU load 50%.

All experiments are run on a Google Kubernetes Engine (GKE) with Kubernetes version 1.13 using 20 n1-standard-4 instances. Each instance has 4 vCPUs, 15 GB RAM, and 100 GB SSD with Container-Optimized OS from Google. To avoid network speed variations between a public Docker registry and the daemons, I set up a private container registry on Google cloud. As discussed in section (§7.4.1), I set up Work Queue framework with one worker per pod. To monitor the resource consumption of tasks, I enable the resource monitor [97] of Work Queue.

7.6.1 Multistage Workload

I start by considering a multistage BLAST workload, which contains three stages and each stage involves three steps, i.e. . splitting an input data, aligning subsequences, and reducing intermediate results. Five dimensions are considered: i) workload execution time; ii) resource shortage; iii) resource supply; iv) accumulate resource shortage; and v) accumulate resource waste. Particularly, accumulate resource shortage/waste is the definite integral of resource shortage/waste in the time range of the workload lifecycle.

As shown in Figure 7.12a, the first and last stages of the workload contains tasks more than the second stage (200, 164 compared to 34). I expect to see a decrease in resource demand in the middle of the lifecycle, and a bump up once the workload entering into the third stage. And an ideal autoscaler should resize the cluster according to the same pattern.

However, as shown in Figure 7.12b, if HPA is applied, the cluster size will gradually increase and stay at the capacity limit (i.e. 20 nodes, 60 cores) until the workload completes. This is because, to avoid pods from thrashing, there is a stabilization interval between two downscale operations and the default value is 5 minutes. Even though the frequency of downscale can be increased by tuning this value, different

workloads have various resource changing rate, without running a workload multiple times, it is challenging to pick the right value.

In contrast, HTA autoscales the cluster as expected. As shown in table 7.11a, even though there is a slight increase in workload execution time (12.5% compare to HPA-20%, 16.6% compared to HPA-50%), HTA reduces the resource waste dramatically (5.6× compare to HPA-20%, 4.30× compare to HPA-50%).

In general, when resizing resource pool for workload with fluctuant resource demands, HTA can make a more accurate autoscaling plan compare to HPA as HTA considering information from every component of the software stack.

7.6.2 I/O Intensive Workload

While CPU load is a good indicator for system load, applications' performance might be bound by other resources. Choosing a wrong indicator might cause HPA scaling cluster to an inappropriate degree. To reveal how will autoscaler behave for workload bounded by resources other than CPU, I create a synthetic workload that contains 200 I/O intensive parallel tasks. Each task of them run `dd` commands to read/write data from a disk device. Same dimensions as previous benchmark are considered 7.6.1.

As shown in sub-figures (i) and (ii) of Figure 7.13b, while tasks are queueing up on Work Queue, the cluster size maintain in 1. The reason is that each task is busy at reading/writing data, and the CPU load is rarely over 20%. In contrast, HTA is able to scale up the cluster to the desired size as it considering CPU load as well as other resources (e.g., max number of processor required by task) usage when making an autoscaling plan. As a result, by using HTA, the cluster is successfully scaled up, and the workload execution time is shortened by around 3.66×.

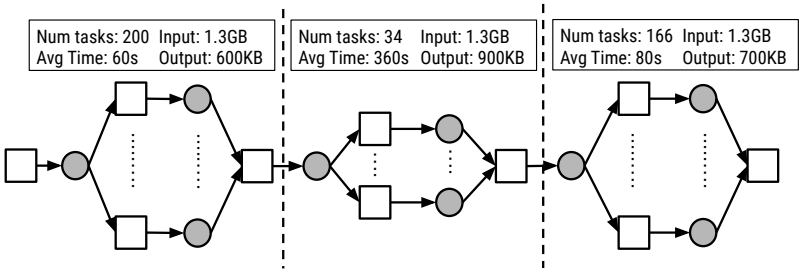
In terms of resource waste, even though configuration using HPA does not have resource waste, the significant resource shortage and small cluster scale results in

unacceptable throughput and execution time. In contrast, when running with HTA, even though there is a small amount of resource waste at the beginning as Work Queue master assigning tasks to workers, once the cluster upscaled to the desired degree, this is no resource waste during the entire lifecycle of workload.

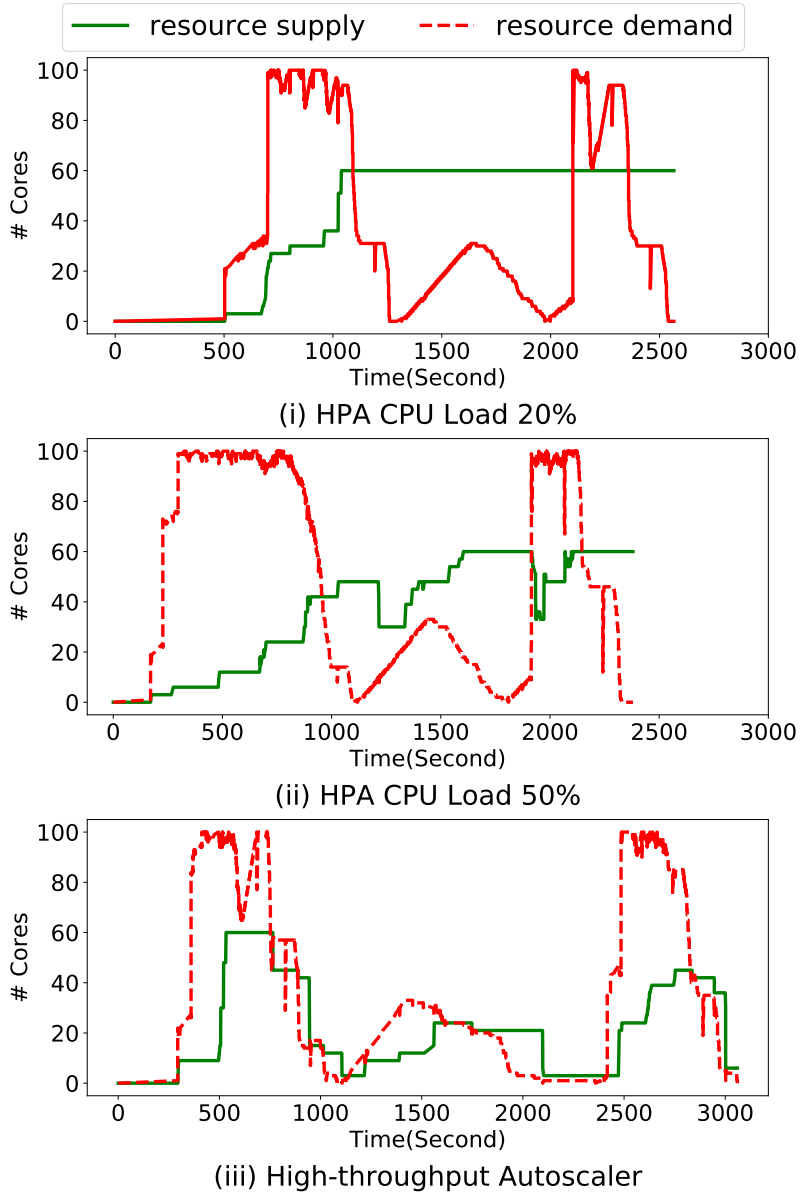
In general, using HPA require users to know the workload well and pick the correct resource indicator. Moreover, to scale the cluster to the desired degree, users need to fine-tune multiple system options. However, without running workloads multiple times, it is challenging for regular users to choose appropriate parameters for each option. By contrast, HTA estimates the resource shortage based on the real-time status of different system components, and dynamically adjust the stabilization cycle by considering the latest resource preparing latency.

7.7 Conclusion

In this chapter, I develop and implement HTA, a better-informed resource autoscaler for HTC workflows on Kubernetes. I show that HTA outperforms the default autoscaler of Kubernetes – which improves the resource usage by 5.6× and shortens the execution time of workflows by up to 3.66× – by synthesizing resource consumption of complete task, real-time status of task queue and resource preparing latency. An important lesson I learned is that the **Cross-layer Cooperation** provided by the emerging container orchestrator can help us to improve the system performance that can't be done by using conventional technologies.

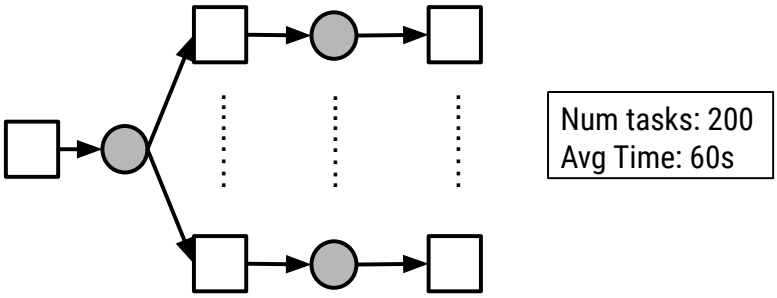


(a) Blast Workflow Structure

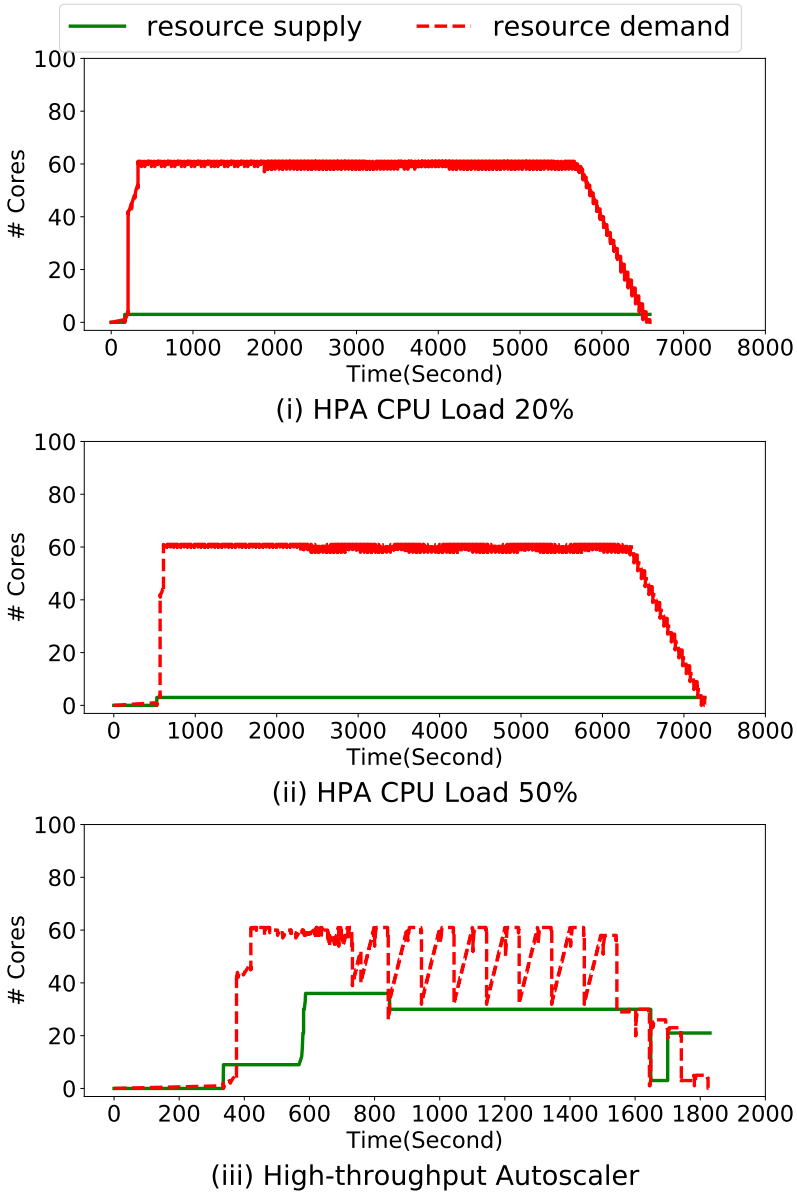


(b) Resource Supply and Demand of Blast

Figure 7.12: Blast Workflow



(a) I/O Bound Workflow Structure



(b) Resource Supply and Demand of I/O Bound Workflow

Figure 7.13: I/O Bound Workflow

CHAPTER 8

CROSS-CHECKING WITH ONLINE WORKLOADS

8.1 Introduction

Even though online workloads (i.e. latency-sensitive workload) have optimization goal different from HTC workflows, when their scale is increased enough (e.g., starting hundreds of services per second), they share common characteristics with HTC workflows: task schedulers are busy most of the time; resource supply cannot meet resource demand; also, a tremendous amount of data is generated. Consequently, a methodology that works to HTC workflows should be applicable to large scale online workloads as well. In this chapter, I validate the seven-element methodology by applying it to my work in Alibaba which standardizes the resource provisioning process for extreme-scale online workloads.

Large organizations often run millions of tasks over cloud-scale computing clusters [40]. To efficiently manage resource at scale, many efforts have been made [45, 111, 38]. These works mainly focus on improving system scalability and efficiency of handling stable daily workloads. However, bursting workloads that spike CPU usage, network traffic happen occasionally. For example, during major sporting events, millions of audiences watch games through online streaming services; rideshare platforms receive overwhelming requests from sites around large events; during holidays and promotion events, online marketplaces need to handle a substantial amount of transactions that can be 100 $\times$ more than usual. Inefficiently handling such workloads can lead to customer unsatisfaction and immediate economic loss [42, 86, 52].

Back in summer 2018, I work with the team of container platform in Alibaba as a research intern. During that time, I cooperate with engineers and scientists from Alibaba to develop a new generation of container orchestrator for cross datacenter resource management. A major challenge we faced is how to manage resources for extreme-scale online workloads efficiently. To overcome this challenge, I design new resource scheduling policies for the cluster manager as well as standardize the overall resource provisioning process consisting of existing resource management mechanisms.

In Alibaba, there are millions of online services running across datacenters worldwide daily. Besides, the annual sales event (ASE) on November 11th has almost all active users involved, which generates a peak throughput of 100 $\times$ higher than daily average and 10 $\times$ higher than daily peaks. Thus, the critical challenge of handling ASE is how to handle the higher volume of requests without using excessive extra resources. Specifically, three facts that make it insurmountable are: (i) purchasing dedicated servers just for handling this event is undesired, since new resources will be underutilized the rest of the year; (ii) given that tremendous resources are required, renting resources from public cloud services is unrealistic; (iii) current autoscaling mechanisms are not agile enough, as the bursting workload can hit the throughput peak in minutes while existing autoscaling mechanisms have comparatively long feedback loops [72, 71]

To address these problems, I analyze the characteristics of bursting workloads, synthesize existing mechanisms, and standardize the resource provisioning process. Throughout the work, I observe that the seven-element methodology developed for HTC workflow applies to extreme-scale online workloads as well.

8.2 Extreme Load Event

In this section, I give the details of the bursting online workload happened by *ASE* (section 8.2.1). I divide the problem of how to efficiently handle bursting online workload into three subproblems and resolve them by extending three heuristic approaches (section 8.2.2).

8.2.1 Bursting Online Workload

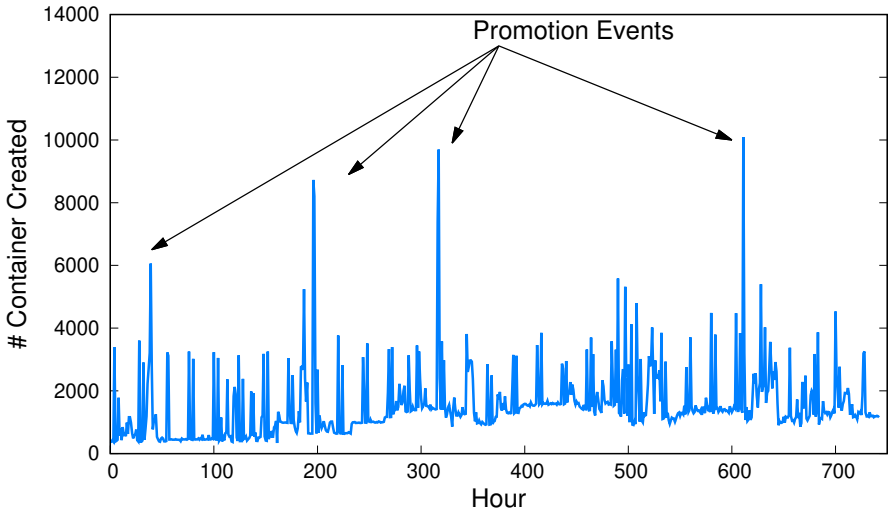


Figure 8.1: Container Created Hourly

In Alibaba, the eCommerce services consume 40% of the total resources every day. As most offline marketplaces, the eCommerce platform has promotion events happening year-round, which results in throughput several times higher than usual. As shown in figure 8.1, the number of requests for creating new containers are 2.5 to 4 times more than average.

Among all promotion events, the *ASE*has the largest peak throughput that is

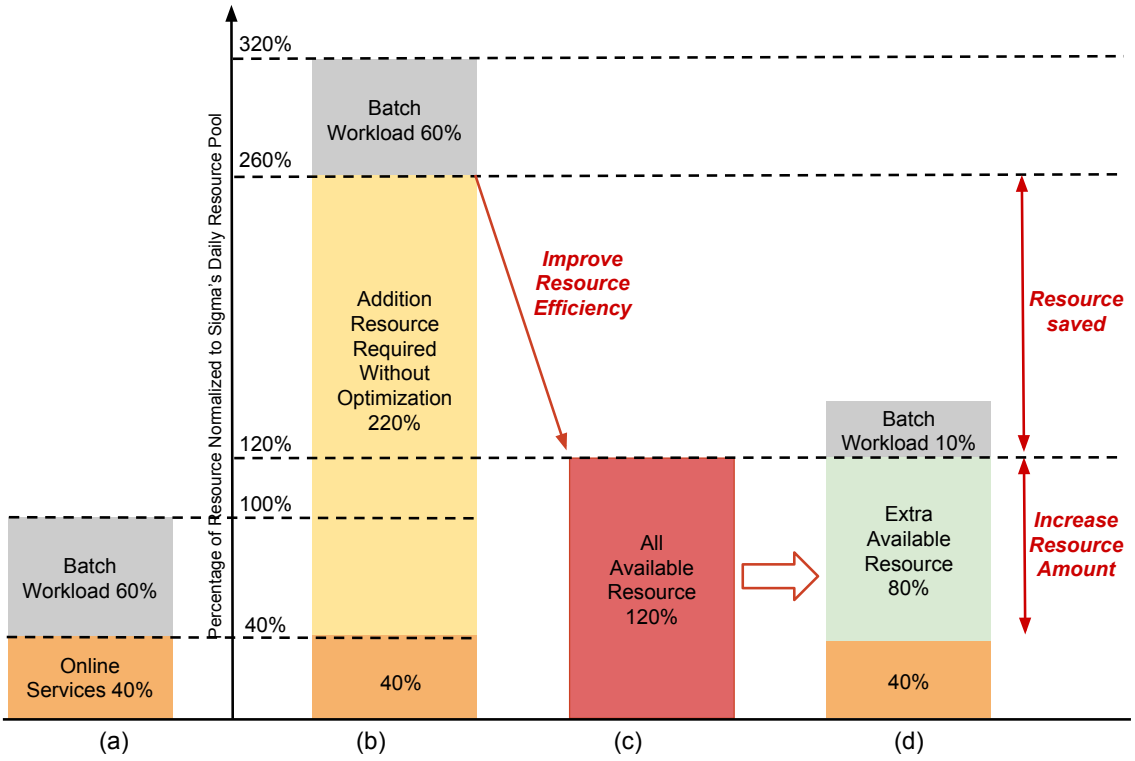


Figure 8.2. Problem Definition

hundreds of times higher than the daily average. For example, the *ASE*of 2017 had gross sales larger than the sum of Black Friday and Cyber Monday ($25.3 billion vs. $19 billion dollars), which produced 325, 000 of peak TPS with 256, 000 orders placed per second (see figure 8.3 and figure 8.4).

A vital characteristic of the *ASE*is that customers intend to place orders in the first few minutes due to the limited stock, which is due to the nature of promotion event, i.e. low price but limited stocks. **This behavior spikes the throughput which reaches the peak within seconds**. Considering the products' stock, logistics capacity, advertising, and sales strategies, another essential feature of promotion events is that **the starting time, ending time and the max trading volume are predictable**

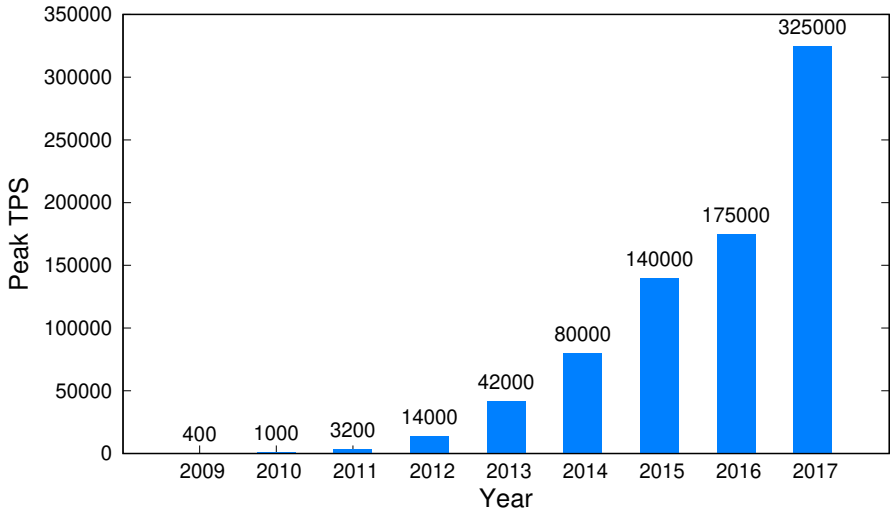


Figure 8.3: Peak TPS of since 2009, the growth of peak TPS since 2009 across 18 datacenters.

8.2.2 Redefine the Problem

A common way to handle increasing throughput is reactive autoscaling, which monitors particular system metrics and scales the resource pool on-demand. Autoscaling works well for workloads with steadily growing throughput, but it put forward two requirements that prevent it from being used for bursting online workloads. First, target workloads must be latency tolerant. Even under ideal conditions, it can take minutes for autoscaling mechanisms [72, 71] to scale up the resource pool, while bursting workloads often reach the peak throughput in seconds. Second, there must be enough available resource for scaling up, while bursting workloads have throughput hundred times higher than usual, which can quickly exhaust the resource buffer.

To efficiently handle bursting workloads, first, I redefine the problem based on the data from *ASE*of 2017. As shown in figure 8.2, in regular times, online and batch workloads own 60% and 40% of the resources pool respectively. During the *ASE*, online workloads will consume 220% of additional resources. In line with the budgetary constraint, only 30% of extra resources are available. Also, to maintain key production workloads, the resources owned by batch workloads can not be lower than 10%.

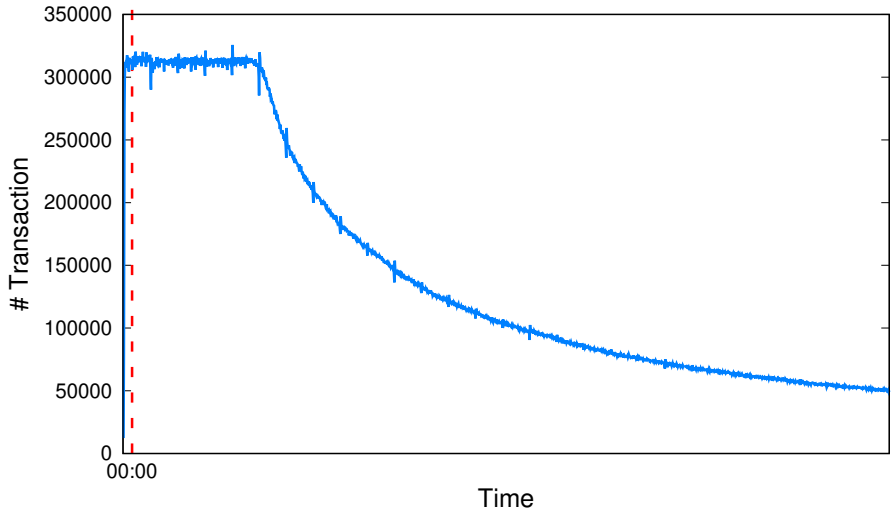


Figure 8.4: TPS of *ASE* from midnight of 11/11, Transaction per second since the midnight of November 11, 2017

Therefore, the maximum amount of resources available to online workloads is 120%, and a better-defined problem is: **how to successfully handle bursting online workloads that require** 5.5× **more resources than daily average with** 2× **more available resources?** To resolve this problem, I synthesize existing mechanisms developed by engineers in Alibaba and standardize the resource provisioning process by extending three heuristic approaches:

1) Increasing Resource Amount, which includes dynamically changing the resource shares between online and batch workloads, and more agile scheduling with finer-grained resource isolation. This approach fuses two factors of the seven-factors methodology, i.e. *Isolation Granularity, and Resource Management.*

2) Improving Resource Efficiency, which contains properly assessing the trade-offs between availability and reliability, scaling up/down selective services, and establishing an optimal allocation plan in advance. The development of this approach conforms to four factors of the *SFM*, i.e. *Container Management, Image Management, Resource Management, and Network Connection.*

3) Ensuring the Reliability, which includes iteratively validating the resource

allocation plan with the gradually increased workload. This approach requires cooperation between components and correlates to the factor *Cross-Layer Cooperation*.

8.3 Increasing Resource Amount

First, to increase the resource amount, I explore i) how to acquire maximum resource share for online workloads (section 8.3.1); ii) how to construct an on-demand resource buffer (section 8.3.2), which conforms the factor, *Resource Management*; and iii) how to develop hyper-threads(HT) level isolation for agile scheduling (section 8.3.3), which correlates to the factor, *Isolation Granularity*.

8.3.1 Dynamic Quota Change

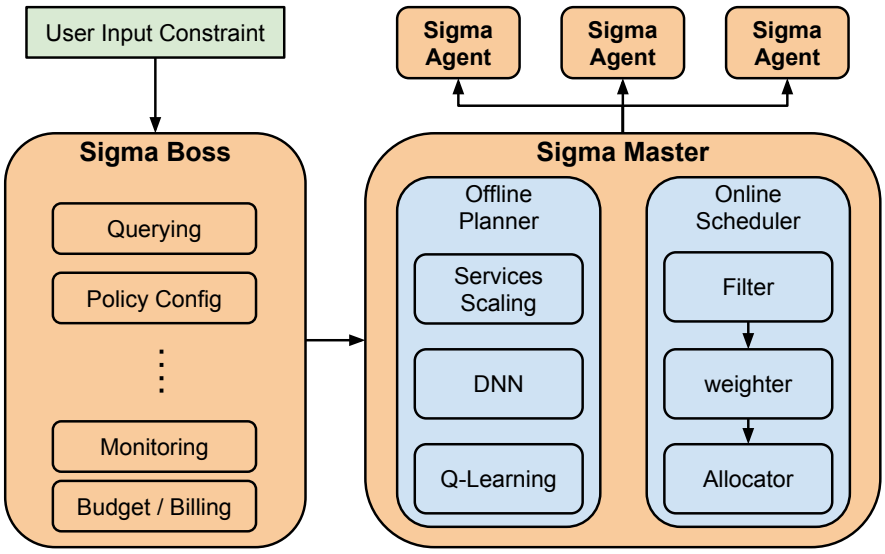


Figure 8.5. Sigma Architecture

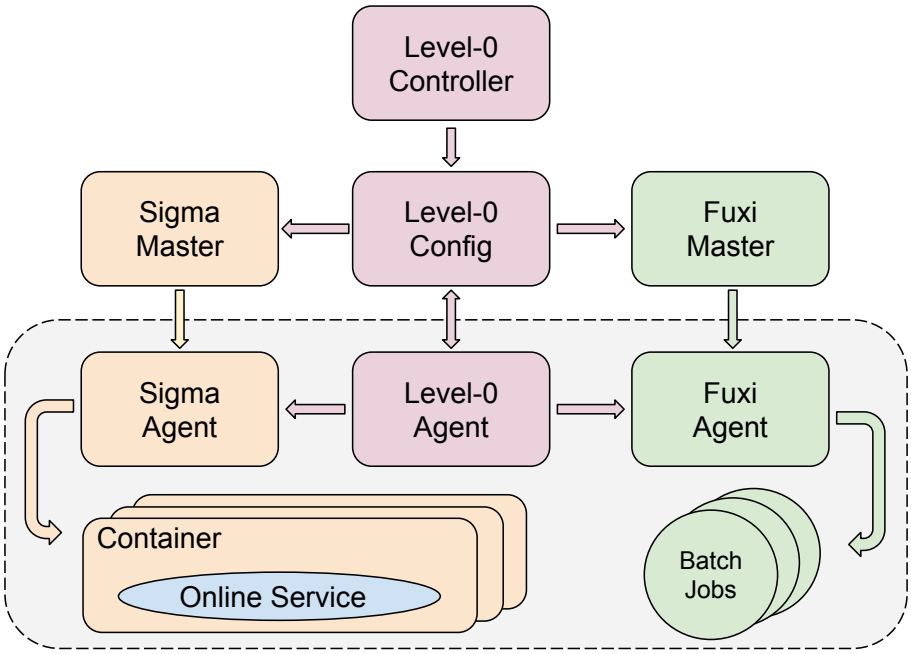


Figure 8.6. Colocation with Level-0

In Alibaba, there exist two resource manager allocating resources for online and batch workloads with a shared resource pool, i.e. *Sigma* for online workloads and *Fuxi* for batch workloads. As the allocation plan for the *ASE*is kept evolving before the event (details of how to enact the allocation plan will be introduced in later sections), resource quota between the *Sigma* and the *Fuxi* is changed more frequently than usual (e.g., ten times per day vs. one time per day), which poses two challenges, i) how to quickly spread new resource quotas across thousands of machines across datacenters, also, ii) how to improve the overall resource utilization without downgrading the QoS of online services.

To resolve the first challenge, *Level-0* collocation coordinator is developed, which vertically integrates into every stack of the infrastructure. As shown in figure 8.6, to change the resource quota, *Level-0* configurator first reads the latest resource quota, then pass it to *Level-0* controller, which enforces *Sigma* master and *Fuxi* master to

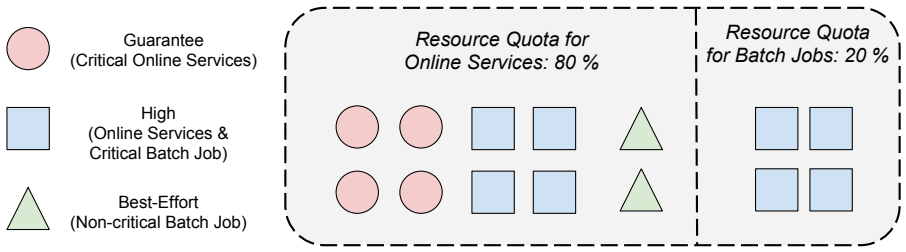


Figure 8.7. Resource Utilization of Different Tasks

allocate tasks based on the new quota. Meanwhile, a *Level-0* agent is running on each node, which will gracefully terminate jobs that are using resources not belonging to its corresponding scheduler. In addition to the function of quota assignment, *Level-0* is also responsible for collecting and aggregating resource statistics.

The top priority during the *ASE*is ensuring the QoS of online services, thus, a portion of batch workloads are temporarily paused and spared their resource share to online services. As Shown in figure 8.7, during the *ASE*, the resource shares owned by online and batch workloads are 80% and 20% respectively. At the same time, the boundary of resource pools of two schedulers are blurred (i.e. a portion of resources are available to both *Sigma* and *Fuxi*), and three tasks levels are defined and applied to both online and batch workloads: (i) *Guarantee* – this level of tasks are always able to obtain enough resources, critical online services belong to this level; (i) *High* – tasks of this level can access resources assigned to corresponding scheduler, non-critical online services, and critical batch jobs are marked as *High*; (iii) *Best Effort* – tasks of this level are allowed to access spare resources belong to the other scheduler and will be interrupted when there is no enough resources for higher-priority tasks, non-critical batch jobs are marked as *Best Effort*.

By assigning different priority levels to tasks, batch jobs can exploit any idle resources that not being used by online services, which increases the resource utilization. As shown in figure 8.8, through dynamic resource sharing and three-level

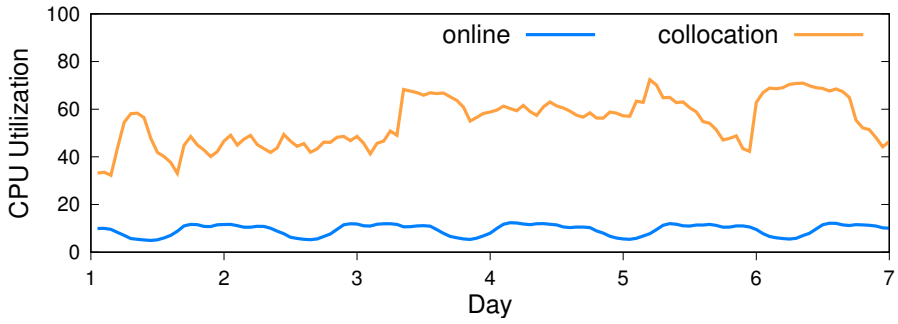


Figure 8.8. CPU Saved through Collocation, CPU utilization with and without collocation. The data is collected from server that contains 96 physical core and 460G Memory

priority, the CPU and memory utilization have been improved by 1.7× and 1.2× respectively.

8.3.2 On-demand Resource Buffer

To handle unexpected events – servers down, network fault, and power outage, etc. – a standby resource buffer is set up on the cloud. Considering the massive amount of required resources, entirely relying on the public cloud providers may affect the performance of currently running services. Consequently, to ensure there is no adverse effect on existing services on the cloud, the QoS of the existing services are closely monitored.

Recalling the *Resource Management* factor from SFM for HTC workflows, the above mechanisms share the same principles with it, i.e. high resource utilization requires customized resource management mechanisms that thoroughly considered the characteristics of target workloads.

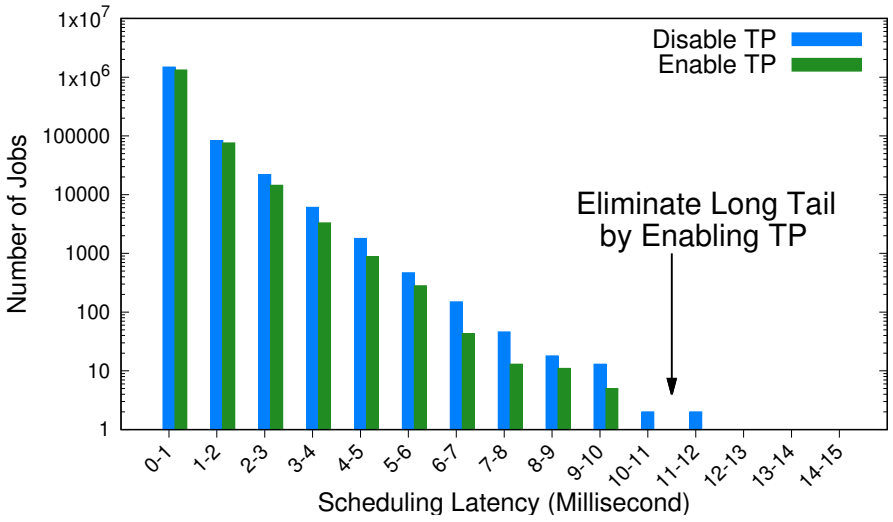


Figure 8.9: Task Preemption, For testing Task Preemption, we run two configurations with/without task preemption. Both configurations have 1×10^5 online tasks and 5×10^{10} batch. Configuration that schedules online tasks in shorter period, has better performance.

8.3.3 Fine-grained Resource Isolation

To share resources between online workloads and batch workloads without interference with each other, finer-grained isolation is required. When collocating online and batch workloads on the same machine, batch jobs might affect the decision made by process scheduler, which results in online services suffering from resource starvation. Besides, two jobs running with different hyper-threads in the same core will interfere with each other due to cache and data pipeline competition.

To avoid resource starvation, two mechanisms are integrated into *Complete Fair Scheduler (CFS)* [76], The first one is **Task Preemption**, which is based on the Borrowed-Virtual-Time (BVT) scheduling [64, 48]. To measure the performance, I run hackbench - a benchmark tool for the Linux scheduler. As shown in figure 8.9, by eliminating the scheduling long tail, the total scheduling time has been decreased from 21.182 seconds to 18.278 seconds.

The second mechanism is **Optimised Queue Balancing (OQB)**, which con-

	run	sleep	wait	total
receiver	2.8s	4.3s	6.1s	13.2s
sender	4.0s	4.2s	4.8s	13.2s
OQB rec	2.9s	3.0s	5.5s	11.5s
OQB sndr	4.2s	3.4s	3.6s	11.1s

(a) Optimized Queue Balancing

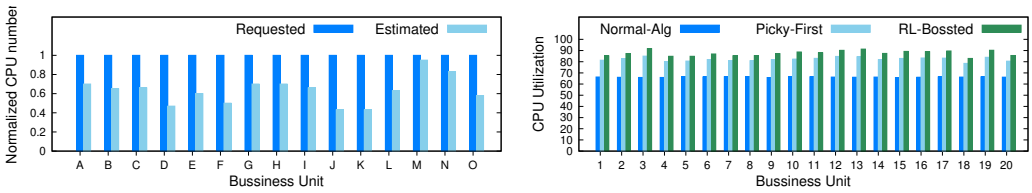
Setting	TPS
No neighbor noise	22629
Disable noise clean	19063
Enable noise clean	21541

(b) Noise Clean

Figure 8.10: Performance Gain through Optimizing CFS, (i) For testing Optimised Queue Balancing (OQB), two setups are compared, transfer time between a receiver and a sender, and transfer time between an OQB receiver and an OQB sender. (ii) For testing Noise Clean, we run 16 threads PostgreSQL and y-cruncher with eight tasks and more than 100 threads on one node, and execute pgbench with eight threads without neighbor on the other node.

siders historical task execution pattern, i.e. task run time and sleep time and avoid allocating multiple online services on the same core. By applying *OQB*, a task with short execution time will not be migrated to the core with less waiting task. To measure the perfoamance, I execute hackbench with customized receiver/sender setting. Shown in figure 8.10a, *OQB* shorten the scheduling latency by 20.5% compared to default CFS.

There are previous studies focused on avoiding the cache contention [89], but Hyper-Threads (HT) running on the same core might also compete with each other for the instruction pipeline. To avoid not only cache but also pipeline competition, a list of competitive jobs is maintained globally. To prevent competitive jobs from stacking on the same core or HT, four **core/HT sharing** labels are developed (core refer to physical core, HT or CPU refer to a Hyper-threading) on datacenter scheduler level. They are (i) *SameCoreFirst* – containers with this label prefer being allocated on the same physical core, e.g., to increase resource utilization and energy efficiency, the process scheduler prefer scheduling batch jobs with small resource requirement on busy CPUs than idle CPUs; (ii) *SpreadFirst* – jobs marked by this label won't be scheduled on a busy core, (iii) *Exclude apps* – jobs with this label reject to be scheduled with containers belong to the given *apps* on the same core, for example,



(a) CPU saved by Realtime Resource Estimation

(b) CPU Usage of Different Algorithms

Figure 8.11. Improve Cluster Resource Efficiency, (i) the y-axis is Normalized by the number of CPU required by each businese unit. (ii) the CPU usage is provided by 20 businese units, with each holds thousands of machines.

two compute-intensive batch jobs can not share the same core; (iv) *Exclusive* – if a core is occupied by an exclusive job, no other job will be allocated on it, e.g., a critical online service need to monopolize a physical core.

In addition, **Noise Clean** – a new mode for CFS – is developed as well, which prevent a batch job from being executed on a core that has online jobs running on it. As shown in table 8.10b, after activating noise clean mode, online services can handle 1.3 × more transactions per second.

The above mechanisms improve the granularity of isolation provided by containers, which allow programs to monopolize vCPU as well as hardwares like cache and instruction pipeline. These mechanisms share a similar principle with the factor *isolation granularity* – target workloads should codetermine isolation granularity and the distributed environments they are run in.

8.4 Improving Resource Efficiency

Next, I describe how to improve the resorce efficiency (i.e. the ratio of resource actually being used) without affecting QoS. To achieve this, five approaches are taken:

1. container resource usage are measured in real time (section 8.4.1);
2. selective services, like online model training and streaming data analytics, are

scaled down (section 8.4.2);

3. tightly coupled services are allocated as groups, e.g., cooperative services that operate as a single component (section 8.4.3);
4. an optimal scheduling plan is developed in advance (section 8.4.4).
5. an optimized container runtime and image distribution platform is developed (section 8.4.5)

Among the above approaches, the first, the second and the third approach share the same principle with the factor *Resource Management* of *SFM*, and the fifth approach correlates to the factors *Container Management* and *Image Management*.

8.4.1 Realtime Resource Estimation

To receive resources from a shared pool, developers or systems from upper layer need to submit resource requests for container/service to cluster managers, then the managers will make best effort to fulfill these requests. However, developers usually don't have a global view of the cluster (e.g., the capacity of the machine is affected by the workloads running on it [47]), and intend to request resources that are more than enough, which results in low resource efficiency.

As the computing resources are under pressure during the *ASE*, low resource efficiency can only exacerbate this problem. To remedy this, resource requirements proposed by developer are used as references and a resource allocation plan is made by cluster manager based on runtime resource utilization. To achieve more accurate resource estimation, resource utilization from past weeks to past hours are taken into consideration. Shown in figure 8.11a, using the resource estimation model help Alibaba to save 35% of CPU resources (i.e. 1×10^5 cores),

8.4.2 Selective Services Scaling

Besides providing the resource requirement of each container, developers also need to specify the maximum and minimum replication number for each service. During

the runtime, the cluster manager will then scale services in the range based on the ingressing throughput and the response time (RT) of each service. This process is sufficed for daily workload, while resources demands of bursting workloads often spike in minutes. Consequently, the replication number of each service must be precisely defined in advance.

In Alibaba, online services fall into two categories: core services which can be a performance bottleneck of the entire platform and require high stability and safety; and non-core services that provide ancillary functions, like beta features testing.

Particularly, services have the following characteristics will be classified as core services and won't be scaled down during the *ASE*.

- **Statful** – services with state are usually non-deterministic and rely on ephemeral data that are nonrecoverable.
- **Stable** – services that has not been updated for months. They are usually APIs that interact with a bank or other financial institutions, which require high stability and safety level.
- **Memory-Intensive** – services that have memory usage higher than 90%. They normally require short read/write latency, which needs to reload large dataset into memory on failure.
- **Long Startup Time** – services have startup time longer than 600 seconds. As the bursting workload only lasts for minutes, the failure of these services can be critical.
- **High Startup Failure Rate** – services that have a startup failure rate higher than 5%.

To evaluate the risk of scaling down non-core services, the *nice* value is calculated and assigned to each service. Services with larger *nice* value have lower risk to be scaled down. There exists various attributes that can affect the *nice* value. An attribute will be considered, if it, i) implies the service's significance level, like the service type; ii) indicates the service's resource utilization, like the desired resource type; iii) suggests the ability to recover from fault, like the application startup

time and success rate. The *nice* value is the sum of the product of each attribute's estimated cost and weight.

To determine the weight of each attribute of N attributes, N significance levels are defined and weight of each attribute are assigned through paired comparison. For example, considering three attributes, Resource Type ($ResType$), Services Type ($ServType$) and Application Start Time ($StartTime$), as shown in the first row of table 8.1, $ServType$ is twice as important as $ResType$, $StartTime$ is threefold as important as $ResType$. The reset of the table are filled out based on same idea.

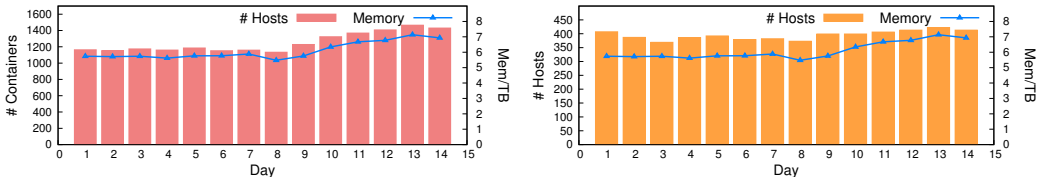
TABLE 8.1

WEIGHTS OF ATTRIBUTES

	ResType	ServType	StartTime	Weight
ResType	1	1/2	1/3	0.149
ServType	2	1	1/3	0.273
StartTime	3	3	1	0.578

For a given service with $ResType = 2, ServiceType = 1$, and $StartupTime = 1$, the *nice* value is, $2 \times 0.149 + 1 \times 0.273 + 1 \times 0.578 = 1.149$. This method thoroughly compares every pair of characteristics, and makes it easier to add new attributes in the future.

Before the *ASE*, marketing department synthetically considers the produce stock, data from previous years, advertising and sale strategy, and give an estimation of the maximum transactions. As a result, the maximum TPS and the demand capacity of



(a) Reduction of Containers and Freed Memory

(b) Involved Hosts and Freed Memory

Figure 8.12. Resources Saved by Scaling Down, Above figures show the amount of memory in terabyte saved on a 1900 nodes' cluster by scaling down the selective services in seven consecutive days. (i) the relationship between decreasing number of containers and amount of freed memory. (ii) the relationship between the number of involved nodes and amount of freed memory.

core services is known in advance, and the scale of core services is determined based on these data.

As shown in figure 8.12, after performing *selective scaling down* on a cluster contains 1900 nodes, the average number of running containers has been reduced by 1250, which involves 359 nodes and free 6.1 Terabytes of memory on average.

8.4.3 Gang Scheduling

As shown in figure 8.13, a complete business transaction consists of more than 200 services and 60 database operations. Due to power and footprint limitation, without enforcement, services can be spread over datacenters. As the network latency between datacenters varies from tens of seconds to minutes, the QoS can be primarily affected. Therefore, when making a resource allocation plan for services, a non-negligible factor is the latency between API calls. Figure 8.14 shows the latency of some API calls, if two consecutive API calls are located in different datacenters, the latency between them can range from 27 to 34 milliseconds, while the shortest latency inside a datacenter is only 3 milliseconds.

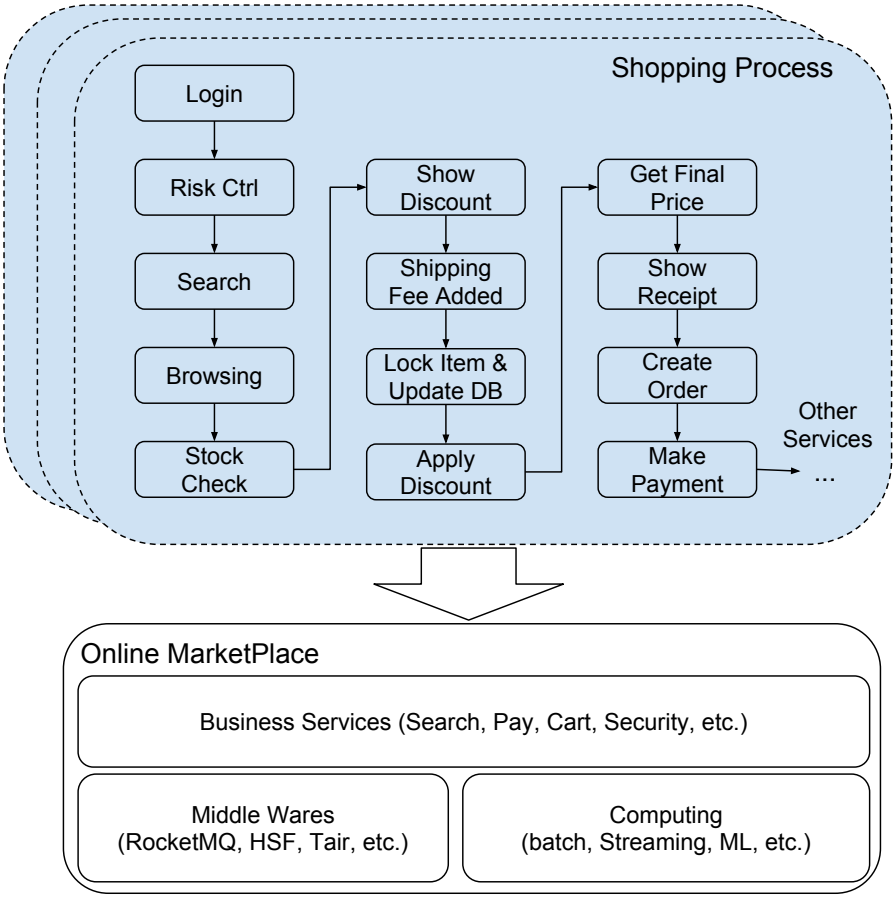


Figure 8.13. A Complete Business Workflow

To decrease the latency, tightly coupled services are bundled and scheduled as a group. As shown in figure 8.15, services and their related data are divided into two categories, i.e. *Unit* and *Central Unit*. Each *Unit* is responsible for a geographic region, hold services and data for customers from the region. Also, it also keeps a copy of all commodities and sellers' data. On the other hand, the *Central Unit* is responsible for running all trade-related services, holding information of buyers, sellers and commodities. During the promotion event, each datacenter will contain several *Units*, and all *Units* are connected to the *Central Unit* with several standby replicas. The *Unit* can handle almost all buyers' operations, including stock checking, product browsing and choosing. Only when an order is placed, the *Central Unit* will

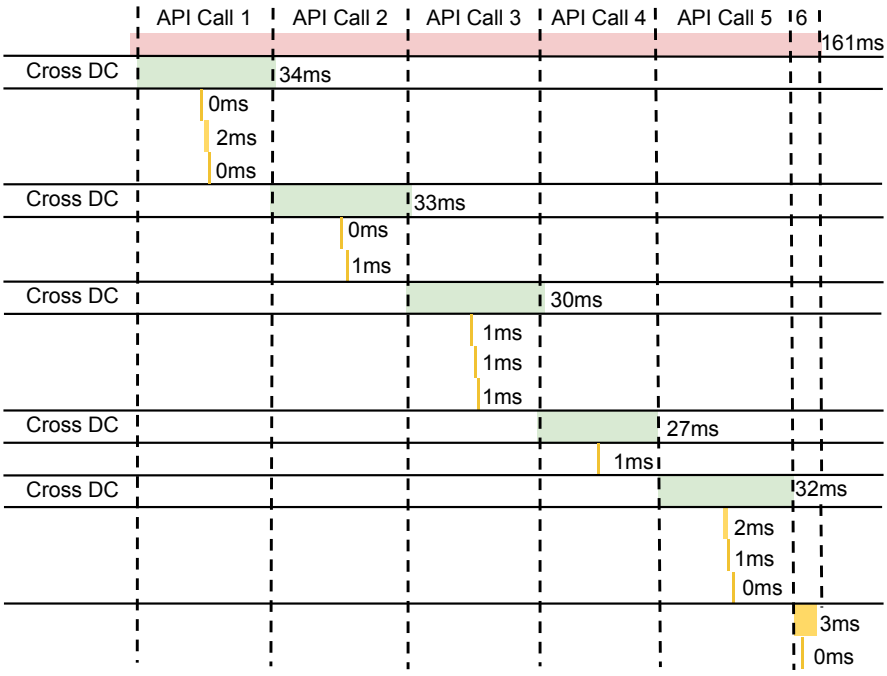


Figure 8.14. Latencies between API Calls

get involved.

Inside the *Central Unit*, the strong consistency is ensured, i.e. once the information of products or customers is updated, all following read requests sent by customers will return the latest state. In contrast, the eventual consistency model is enforced between *Unit* and *Central Unit*. Products and buyers' data in each *Unit* is updated asynchronously. Even though inconsistency might happen between *Unit* and *Central Unit*, each transaction will end up with updating global state in *Central Unit*, which ensure faults caused by inconsistency can be detected.

Through services gang scheduling, unnecessary network transfers that cross datacenters are avoided, which also validates the factor, *network connection*.

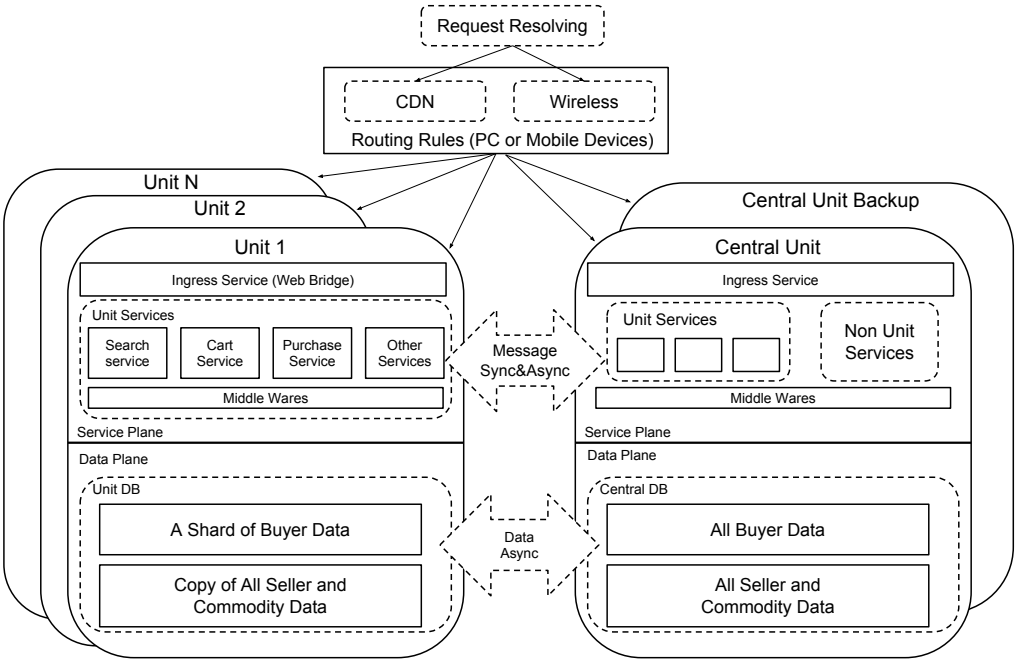


Figure 8.15. Gang Schedule Tightly Coupled Services

8.4.4 Scheduling under Constraints

As mentioned in section 8.3.3, programs might be incompatible to each other on CPU and thread-level. Thus new scheduling policies are enforced on process scheduler. Likewise, programs can interfere with each other on the machine level, for example, running multiple memory-intensive programs on the same machine might result in memory starvation. To prevent these from happening, advanced scheduling policies are developed for cluster manager, which labels services by scheduling constraints (i.e. services (anti-) affinity, CPU/Server monopolization, etc.). Establishing an optimal allocation plan without violating constraints is equivalent to solving a high dimensional bin-packing problem, which can be transferred into the Mixed Integer Linear Programming (MILP) problem and solved by algorithms like Q-learning [106].

The Q-learning algorithm explores the action space and receives rewards from the cluster state. An expected accumulated discount reward is learned for each state-

action pair, given by equation 8.1

$$q_*(s_t, a_t) = \mathbb{E}\left\{\sum_{k=0}^{\infty} \gamma^k r(s_{t+k}, a_{t+k})\right\}. \tag{8.1}$$

In equation 8.1, reward r is the increment of objective value based on cluster state s and action a, and it consists of weighted sum of factors including container allocation cost, host usage cost, balance of resource utilization, requirements on mutex, etc. $Q(s_t, a_t)$ represents the estimation of $q_*(s_t, a_t)$, which is updated according to the following equation,

$$Q^t(s, a) \leftarrow r(s, a) + \gamma \max_{a^*} Q(s', a^*) \tag{8.2}$$

as shown in equation 8.2, $Q^t(s, a)$ is the target value of $Q(s, a)$, and s' is the state immediately following s. The target value is the sum of rewards of current cluster state and its estimated successor's states based on the action. To implement $Q(s, a)$, a deep learning network (DNN) is set up with experiences (s, a, r, s') stored and randomly picked by mini-batches during the training process.

Figure 8.11b shows the CPU utilization after using different scheduling mechanisms. Comparing to the average CPU usage of online scheduling algorithm (66%), the Q-learning algorithm increase the utilization to 87%.

8.4.5 Container Deployment

Rather than extending popular container runtimes, like Docker and LXC, engineers in Alibaba developed a new container runtime, i.e. PouchContainer [68], PouchContainer is developed for satisfying special demands of large-scale distributed environment, including: i) Strong isolation – supporting security features, like hypervisor-based container technology and packed Linux kernel; ii) Fast image distribution – PouchContainer runtime speeds up image distribution by using Dragonfly [67], a P2P container image distribution system; iii) Backward Compatibility – PouchContainer

can work on Linux Kernel before 2.7; iv) Standard Compatibility – PouchContainer conforms to popular industry-standard, e.g., CNI [11] and CSI [13]

By using PouchContainer and Dragonfly image distribution platform, a large number of containers are able to be deployed in a short time. This also suggests that *container and image management* are two essential factors need to be considered when executing large workloads in container environment.

8.5 Ensuring the Reliability

Most extreme load events are short but critical, thus system error is not tolerated during the runtime. Through above technologies, an optimal resource allocation plan can be established, but without extended testing, its reliability can not be ensured. When designing an applicable testing framework, there exist three problems: i) services are tightly coupled – some core functions cannot be covered by unit tests, for example, the success of a function may depends on callback functions from other services; ii) bugs can not be revealed with small workloads – service can behave differently depend on the scale of target workloads, e.g., with increasing throughput, the performance of individual instance may decrease to maintain high availability; iii) large-scale testing is expensive - clusters dedicated for testing usually is not extensive enough, while repeatedly running large-scale testing on production clusters is expensive and can affect the performance of production workloads.

To solve the above problems, *Full-Load-Testing (FLT)* is developed, which tests all related services as a whole and incrementally validate the allocation plan using gradually increased workloads. *FLT* emulates the scenario of the *ASE* by running simulation workflows which include every step of the real business transactions except for the payment step.

Figure 8.16 describes the workflow of *FLT*, Starting with a small workload (e.g., max workload generated by a Unit), only a small amount of resources are required (e.g.,

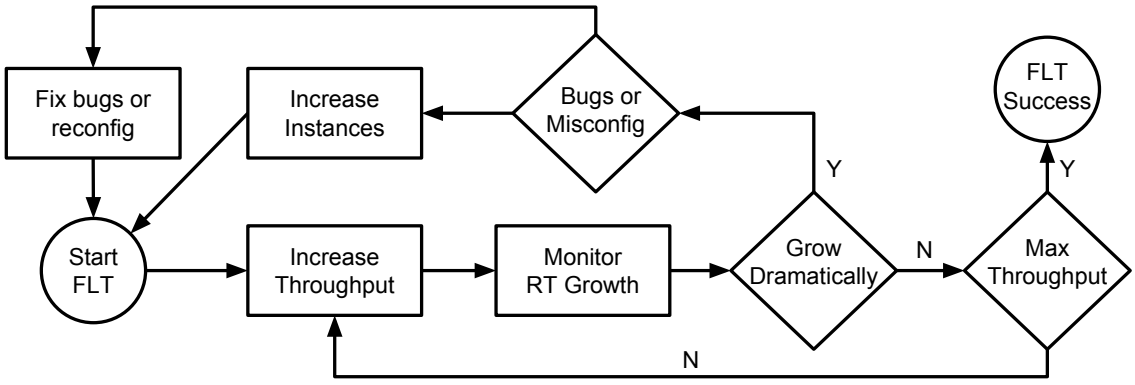


Figure 8.16. The Workflow of Full Load Testing

10000 instances). Next, the scale of the workload is increased, and the QoS (i.e. growth of response time, CPU, memory utilization, etc.) of involved services are being monitored. If any of the parameters grow dramatically, the system log and configuration will be rechecked, and potential software defects will be fixed. Then the *FLT* will be restarted, if the alternation of system parameters is stable, the workload will be increased. Otherwise, the resource pool will be scaled up. When reaching the max throughput, the *FLT* is successfully finished. By this, a solid resource allocation plan is established. To overload the platform without affecting the production workloads, *FLT* is run during the trough of daily business.

Reminiscent of the design factor of *Cross-layer Cooperation*, which is essential for implementing advanced features for HTC workflows. When developing complicated features for large scale online workloads, this factor is also critical, as systems used to handle online workloads usually contain multiple layers.

8.6 Standardize the Process

Synthesize above mechanisms, I standardize the resource provisioning process as follows (see figure 8.17):

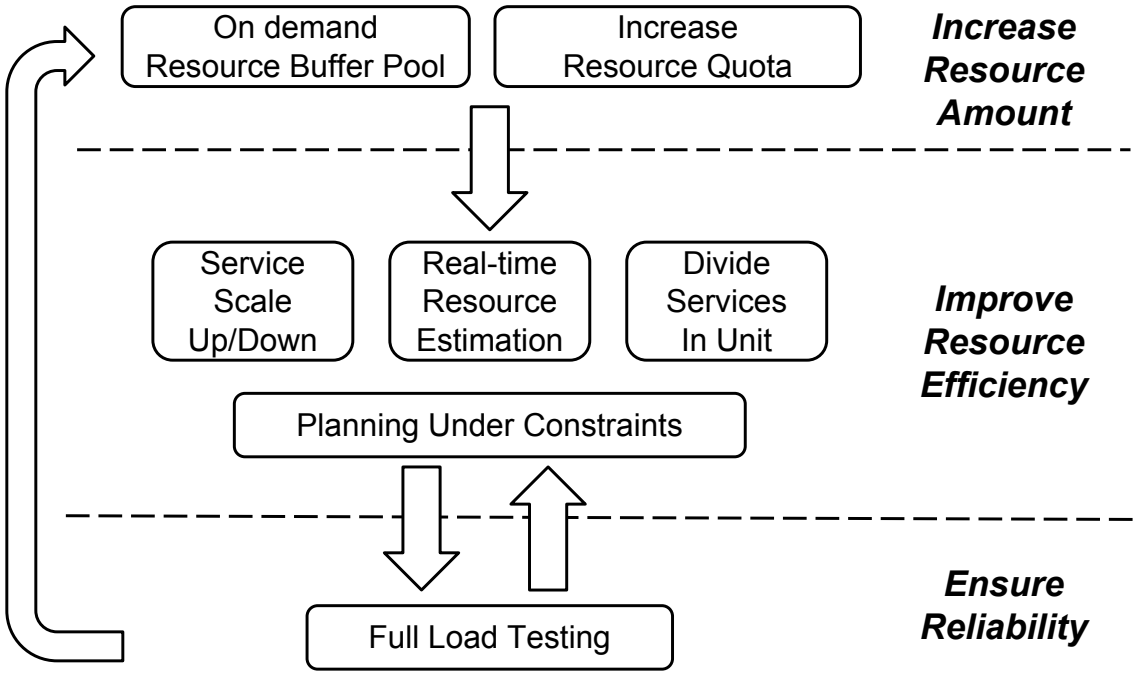


Figure 8.17. Resource Provision Workflow

1. collecting the resource statistics of online services and batch workloads;
2. choosing necessary batch workloads;
3. estimating the resource capacity of these workloads;
4. reassigning resource quota for online and batch workloads;
5. calculating the peak *TPS* and estimating total amount of resources required;
6. dividing online services into core and non-core services;
7. scaling up/down services based on their *nice* value;
8. grouping services in *Unit* and *Central Unit*;
9. converting the resource allocation problem to *MILP* problem;
10. solving the *MILP* problem through *Q-learning* algorithm;
11. validating resource allocation plan through *FLT*;
12. constructing standby buffer on-demand.

8.7 Conclusion

In this chapter, I introduce my work of standardizing the resource provisioning process for extreme-scale online workloads. Through this work, I observed essential design factors that share the same principles with the *SFM*, and I conclude that the *SFM* transcends the boundary between the two contexts, i.e. online workload, and HTC workflow, and is equally applicable to both.

CHAPTER 9

CONCLUSION

9.1 Recapitulation

The emergence of container technology opens up a new possibility to HTC workflows: scaling up on a public cloud with infinite computing resources. However, past studies rarely look into how to containerize HTC workflows. In this dissertation, I explore the possibility of integrating container technology into HTC systems, optimize the system from different aspects, and summarize seven design factors that are generally applicable to container-based system design.

I compare four configurations of integrating container runtime into different layers of existing workflow systems, and summarize two factors: **Isolation Granularity** can be varied and should be determined according to the characteristics of objective workloads; **Container Management** should be done by underlying distributed system without user intervention.

I develop a customized scheduler for connecting workflow systems to Apache Mesos, which improves resource usage as well as shorten the workflow execution time by up to 2× comparing to a primitive setting. Three factors I distilled from this work are: **Garbage Collection** should be done timely considering the tremendous amount of intermediate data generated; excessive **Network Connection** should be avoided given the quantity of data transmission is enormous; **Resource Management** mechanisms should be customized based on the characteristics of objective workloads.

I design and develop a distributed version of Docker, i.e. Wharf, which improves the efficiency of image storage and speeds up the distribution of the images by using a distributed file system. I conclude that the **Image Management** mechanism should be optimized in a distributed environment considering a large number of redundant images across nodes.

I propose an autoscaling strategy that scales HTC workflows according to resource usage of complete tasks, length of task queue, and response time of cluster manager. I implement this strategy by developing HTA – a middleware follows the cloud-native way and resizing HTC workflows on-demand. The general design factor I distill from this work is that **Cross-layer Cooperation** can be used to implement advanced features.

Finally, I standardize the resource provisioning process for extreme-scale online workloads in Alibaba. During this work, I validate the seven design factors by applying them into different steps of the resource provisioning process. I conclude that the seven design factors are equally applicable to designing systems for both HTC workflows and large scale online workloads.

9.2 Future Work

9.2.1 Scheduling Workloads from Different Categories

Due to the diversity of workload categories and the variability of workload behaviors, resource scheduling on the cloud has been a challenging topic for a long time. On the one hand, generic schedulers that aim at serving various workloads can only provide standard functions far from ideal. On the other hand, a domain-specific scheduler that works well for a category of workloads might result in side effects on other workloads residing in the same cloud environment.

In this dissertation, I focuses on improving resource usage for HTC workflows

within their quota. However, this might hurt other workloads to reside on the same cluster. For example, a computing-intensive machine learning task and an HTC task might be assigned to two separate threads located on the same physical core through two schedulers, which can result in cache contention.

A valuable scheduler should be hierarchical as well as cross-framework aware. Specifically, it can extend the conventional two-level architecture, which has a global scheduler to be responsible for allocating resource the quota for different frameworks; and the framework schedulers to assign resources to individual tasks. Besides, to achieve cross-framework awareness, framework schedulers can report unaccountable failures to the central scheduler, and then the central scheduler can rearrange the incompatible frameworks to different physical machines or informing framework schedulers to reallocate the failed tasks.

However, implementing such a system is not trivial. It requires i) sorting out failures that are caused by conflicts between frameworks, ii) supporting container live migration for framework relocation and ii) developing an API that allows framework schedulers to reschedule failed tasks caused by framework conflicts but not interfere with other schedulers. Each of them can be extended to an independent research topic.

9.2.2 Resource Management in Heterogeneous Environment

With the incoming wave of AI, the requirement for computing resources have been dramatically increased. At the same time, different categories of computing tasks have their own best suitable hardware. For example, It is better to use GPU for tasks fitting the single-instruction, multiple-data execution model (e.g., deep learning workloads), while for tasks not large enough to require a GPU, and not small enough to be done in a core (e.g., calculating a midsize polynomial function), FPGA can be a good choice. Therefore, the heterogeneous computing cluster that consists of

different processing units (e.g., GPU, FPGA, AP [103], etc.) connected through the high-speed network have attracted tremendous attention in recent years.

So how to manage resources in a heterogeneous environment? Resource management in a distributed environment is hard, resource management in a heterogeneous distributed environment is even harder as the dimension of the scheduling model will increase dramatically. I see possibilities of mitigating this problem from two aspects.

First, from users' perspective, rather than composing workloads that require rare and expensive hardware, one can create portability workloads that compatible with different hardware, which increases their chances to be scheduled and the possibility of being scaled up. However, this approach requires users to write a modified version of the same program by using different programming languages (e.g., CUDA for GPU; VHDL, Verilog for FPGA), which can largely prolong the development cycle. So what is the best strategy? Should we invent a new programming paradigm that hides certain aspects of the hardware from the programmer? Alternatively, should we develop new interfaces with a mainstream programming language?

Second, from the cluster manager's perspective, virtualizing scarce and expensive hardware can be a good option. However, how to achieve this is still an open question. Should we develop dedicated frameworks [105] for managing these virtualized hardware? Alternatively, should we implement new drivers [20] for attaching them to existing platforms? All the above questions can be developed into research projects worth studying.

BIBLIOGRAPHY

1. CoreOS is Linux for Massive Server Deployments. https://coreos.com/, 2015.
2. Docker, 2018. https://www.docker.com/.
3. Linux containers, 2019. https://linuxcontainers.org/.
4. Project Atomic. http://www.projectatomic.io/, 2015.
5. Amazon Elastic Container Service, 2018. https://aws.amazon.com/ecs/.
6. Argoproj - get stuff done with kubernetes. 2019. https://github.com/argoproj/argo.
7. AUFS - Another Union Filesystem, 2018. http://aufs.sourceforge.net. Accessed March 2018.
8. Basic local alignment search tool. 2019. https://blast.ncbi.nlm.nih.gov/Blast.cgi.
9. cgroups – Linux control groups, 2019. http://man7.org/linux/man-pages/man7/cgroups.7.html.
10. Cloud native computing foundation. 2019. https://www.cncf.io/.
11. Container network interface. 2019. https://github.com/containernetworking/cni.
12. Lightweight container runtime for kubernetes. cri-o authors, 2019. https://cri-o.io/.
13. Container storage interface. 2019. https://github.com/container-storage-interface.
14. Swarm mode overview. Docker Inc, 2017. https://docs.docker.com/engine/swarm/.
15. Docker project official website. https://www.docker.com/, 2015.
16. Amazon ec2 auto scaling. 2019. https://aws.amazon.com/ec2/autoscaling/.
17. Etcd: A distributed, reliable key-value store for the most critical data of a distributed system, 2018. https://coreos.com/etcd/.

18. Google Kubernetes Engine, 2018. https://cloud.google.com/kubernetes-engine/.

19. IBM Spectrum Scale, 2018. https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage.

20. Nvidia container toolkit. 2019. https://github.com/NVIDIA/nvidia-docker.

21. Deploy and manage any containerized, legacy, or batch application. 2019. https://www.nomadproject.io/.

22. IBM Cloud Container Service, 2018. https://www.ibm.com/cloud/container-service.

23. Ibm spectrum lsf v10.1 documentation. IBM Knowledge Center, 2017. https://www.ibm.com/support/knowledgecenter/en/SSWRJV_10.1.0/lsf_welcome.

24. Devops, infrastructure as code, and powershell dsc: The introduction. 2016. https://www.powershellmagazine.com/2016/01/05/devops-infrastructure-as-code-and-powershell-dsc-the-introduction/.

25. Kubeflow: The machine learning toolkit for kubernetes. 2019. https://www.kubeflow.org/.

26. Kubernetes cluster autoscaler. 2019. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

27. Production-grade container orchestration. 2017. https://kubernetes.io/.

28. Marathon: A container orchestration platform for mesos and dc/os. 2019. https://mesosphere.github.io/marathon/.

29. Azure Container Service, 2018. https://azure.microsoft.com/en-us/services/container-service/.

30. namespaces – overview of Linux namespaces, 2019. http://man7.org/linux/man-pages/man7/namespaces.7.html.

31. navops by univa. Univa Corporation, 2017. https://www.navops.io/.

32. Overlay Filesystem, 2018. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed March 2018.

33. Pbs professional. Altair Engineering, Inc., 2017. http://www.pbsworks.com.

34. Rightscale cloud management. 2019. https://www.flexera.com/.

35. rkt: A security-minded, standards-based container engine. 2019. https://coreos.com/rkt/.

36. Snappy Ubuntu Core. http://developer.ubuntu.com/en/snappy/, 2015.

37. Harbor: An enterprise-class container registry server based on Docker Distribution, 2018. http://vmware.github.io/harbor/.

38. A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind facebook messages: Using hbase at scale. 2012.

39. A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, et al. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

40. L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

41. D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson, and I. Shapira. Sla-aware resource over-commit in an iaas cloud. In *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*, pages 73–81. IEEE, 2012.

42. M. Burns. Prime down: Amazon's sale day turns into fail day, 2018. URL https://techcrunch.com/2018/07/16/prime-down-amazons-sale-day-turns-into-fail-day/. Access data: 2018-08-19.

43. G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services.

44. J. Clark. Google: 'EVERYTHING at Google runs in a container', 2014. https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

45. W. Cunningham. Scaling for growth: A q&a with uber's vp of core infrastructure, matthew mengerink, 2018. URL https://eng.uber.com/core-infra-2018/. Access data: 2018-08-19.

46. E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 2014.

47. C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL http://doi.acm.org/10.1145/2541940.2541941.

48. K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *ACM SIGOPS Operating Systems Review*, 33(5):261–276, 1999.

49. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

50. W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

51. J. Frey. Condor dagman: Handling inter-job dependencies, 2002.

52. M. Gagnaire, F. Diaz, C. Coti, C. Cerin, K. Shiozaki, Y. Xu, P. Delort, J.-P. Smets, J. Le Lous, S. Lubiarz, et al. Downtime statistics of current cloud solutions. *International Working Group on Cloud Computing Resiliency, Tech. Rep*, 2012.

53. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

54. L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia. Shifter: Containers for HPC. *Journal of Physics: Conference Series*, 898(8), 2017.

55. I. Goiri, J. Guitart, and J. Torres. Characterizing cloud federation for enhancing providers' profit. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 123–130. IEEE, 2010.

56. Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

57. T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

58. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

59. D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl_2):W729–W732, 2006.

60. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

61. J. H. Jonsson, R. N. Durkin, G. Cuthbertson, and G. Lin. Batch job execution using compute instances, May 21 2019. US Patent App. 15/275,246.

62. W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin. FID: A Faster Image Distribution System for Docker Platform. In *2nd IEEE International Workshop on Foundations and Applications of Self* Systems (FAS* W)*, 2017.

63. G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459. URL `https://doi.org/10.1371/journal.pone.0177459`.

64. J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, Eurosys '14. ACM, 2014.

65. H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

66. H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10. ACM, 2010.

67. A. G. H. Ltd. Dragonfly: Intelligent p2p file distribution system, 2018. URL `https://github.com/alibaba/Dragonfly`. Access data: 2018-08-19.

68. A. G. H. Ltd. Pouchcontainer: Alibaba's open-source container runtime, 2018. URL `https://github.com/alibaba/pouch`. Access data: 2018-08-19.

69. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

70. F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

71. M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

72. M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, 2012.

73. M. Mazzucco, D. Dyachuk, and R. Deters. Maximizing cloud providers' revenues via energy aware allocation policies. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 131–138. IEEE, 2010.

74. P. Menage. Adding generic process containers to the Linux kernel. In *Linux Symposium*, 2007.

75. D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

76. I. Molnar. Linux completely fair scheduler, 2007. URL `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`. Access data: 2018-08-19.

77. S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan. CoMICon: A Co-operative Management System for Docker Container Images. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2017.

78. B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. BlobSeer: Next-generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, 71(2), 2011.

79. A. Nordal, Å. Kvalnes, and D. Johansen. Paravirtualizing tcp. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 3–10. ACM, 2012.

80. B. Nowicki. Nfs: Network file system protocol specification. Technical report, 1989.

81. K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

82. P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26. ACM, 2009.

83. H. N. Palit, X. Li, S. Lu, L. C. Larsen, and J. A. Setia. Evaluating hardware-assisted virtualization for deploying hpc-as-a-service. In *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*, pages 11–20. ACM, 2013.

84. S.-M. Park and M. Humphrey. Self-tuning virtual machines for predictable escience. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 356–363. IEEE, 2009.

85. T. Patikirikorala, A. Colman, J. Han, and L. Wang. A multi-model framework to implement self-managing control systems for qos management. In *Proceedings of the 6th international symposium on software engineering for adaptive and self-managing systems*, pages 218–227. ACM, 2011.

86. D. A. Patterson et al. A simple way to estimate the cost of downtime. In *LISA*, volume 2, pages 185–188, 2002.

87. R. Priedhorsky and T. Randles. Charliecloud: Unprivileged Containers for User-defined Software Stacks in HPC. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

88. M. A. Rodriguez and R. Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5): 698–719, 2019.

89. J. ROHIT and L. DAVID. Cat at scale: Deploying cache isolation in a mixed workload environment. *LinuxCon+ Container-Con North America*, 2016.

90. N. Savage. Going Serverless. *Communications of the ACM*, 61(2), 2018.

91. M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. 2013.

92. S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

93. G. Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

94. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

95. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

96. B. Tovar. Resource management with makeflow & work queue. In *CCL Workshop on Scalable Scientific Computing 2016*, 2016.

97. B. Tovar, R. F. da Silva, G. Juve, E. Deelman, W. Allcock, D. Thain, and M. Livny. A job sizing strategy for high-throughput scientific workflows. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):240–253, 2017.

98. B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI):239–254, 2002.

99. B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.

100. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

101. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

102. D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Transactions on Internet Technology (TOIT)*, 7(1):7, 2007.

103. K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron. An overview of micron's automata processor. In *2016 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*, pages 1–3. IEEE, 2016.

104. T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. An Ephemeral Burst-buffer File System for Scientific Applications. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

105. Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27 (12):3547–3560, 2016.

106. C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

107. T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

108. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9): 633–652, 2011.

109. D. Williams, R. Koller, M. Lucina, and N. Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 199–211. ACM, 2018.

110. L. Xia and P. A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, pages 11–18. ACM, 2012.

111. A. Zhang and W. Yan. Scaling uber's hadoop distributed file system for growth, 2018. URL `https://eng.uber.com/scaling-hdfs/`. Access data: 2018-08-19.