Deadline-Aware Reservation-Based Scheduling

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Ahmad Rahman

Permission to Use

Abstract

The ever-growing need to improve return-on-investment (ROI) for cluster infrastructure that processes data which is being continuously generated at a higher rate than ever before introduces new challenges for big-data processing frameworks. Highly complex mixed workload arriving at modern clusters along with a growing number of time-sensitive critical production jobs necessitates cluster management systems to evolve. Most big-data systems are not only required to guarantee that production jobs will complete before their deadline, but also minimize the latency for best-effort jobs to increase ROI.

This research presents DARSS, a deadline-aware reservation-based scheduling system. DARSS addresses the above-stated problem by using a reservation-based approach to scheduling that supports temporal requirements of production jobs while keeping the latency for best-effort jobs low. Fined-grained resource allocation enables DARSS to schedule more tasks than a coarser-grained approach would. Furthermore, DARSS schedules production jobs as close to their deadlines as possible. This scheduling policy allows the system to maximize the number of low-priority tasks that can be scheduled opportunistically. DARSS is a scalable system that can be integrated with YARN.

DARSS is evaluated on a simulated cluster of 300 nodes against a workload derived from Google Borg's trace. DARSS is compared with Microsoft's Rayon and YARN's built-in scheduler. DARSS achieves better production job acceptance rate than both YARN and Rayon. The experiments show that all of the production jobs accepted by DARSS complete before their deadlines. Furthermore, DARSS has a higher number of best-effort jobs serviced than Rayon. And finally, DARSS has lower latency for best-effort jobs than Rayon.

Acknowledgements

Throughout my research, I have received a great deal of support and assistance.

First and foremost, I would like to thank my supervisor, Professor Nadeem Jamali, from the bottom of my heart. Nadeem, your leadership and expertise made it possible for me to contribute to the field of distributed computing. Your detailed feedback allowed me to refine my work. Thank you for supporting me well beyond my research. I am grateful to you for all of the opportunities I was given.

I am grateful to the Department of Computer Science at the University of Saskatchewan for the Departmental Scholarship and to the Canada First Research Excellence Fund (CFREF) for the funding through Professor Jamali's P^2IRC Grant.

I want to thank Professor Dwight Makaroff, Professor Zadia Codabux, and Professor Ramakrishna Gokaraju for being my thesis committee members. All of them made many important suggestions to improve the quality of this work.

I want to thank my friends and colleagues, Ahmed Abdel Moamen, Waqas Ur Rehman, and Lujie Duan, for their support throughout the course of my studies. I would also like to thank Ms. Sophie Findlay and Ms. Gwen Lancaster, who coordinated my program and helped whenever needed.

Finally, I would like to express my gratitude to my partner, Saalimah. Without her tremendous understanding and encouragement in the past few years, it would be impossible to complete my study.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
CS	Capacity Scheduler
DAG	Directed Acyclic Graph
DARSS	Deadline-Aware Reservation-based Scheduling System
GCU	Google Compute Unit
HPC	High Performance Computing
PJ	Production Job
QoS	Quality of Service
ROI	Return-on-Investment
RSL	Resource Specification Language
SLA	Service Level Agreements
TTD	Time-to-Deadline

Chapter 1
INTRODUCTION

This chapter presents an overview of deadline-aware scheduling in big-data systems. Section 1.1 discusses the motivation behind this work. Section 1.2 characterizes the limitations of state-of-the-art systems. Section 1.3 details the cluster scheduling approach taken in this work. Section 1.4 presents the contributions of this research. Finally, Section 1.5 presents the structure of the remainder of the thesis.

1.1 Motivation

Data production has seen an exponential increase since the emergence of the World Wide Web (www). More than 90% of the world's data has been produced in the past four years [19]. More data will be produced over the next three years than the data created over the past 30 years [23, 24]. According to an estimate [22], 1.7 MBs of data was produced every second for every person on earth in 2020. Another estimate [39] puts the worldwide daily data production at 463 exabytes by the year 2025. The data being generated has greater variety, arrives in increasing volumes and with ever-higher velocity. These three characteristics make data, big data [24]. Although data has intrinsic value, extracting the value requires processing the data. It turns out that big data is too large and complex to be processed with traditional data-processing techniques.

Systems used to ingest, store, and analyze big data are called big data systems. Big data systems can be implemented using cluster computers or grid computers. A cluster computer is a centralized distributed computing environment where multiple standalone homogeneous computers, i.e., having the same type of hardware and same operating system) work together as a single consolidated system [9]. A grid computer is a decentralized distributed computing environment where multiple standalone homogeneous or heterogeneous computers, i.e., having different types of hardware and operating systems, work together to perform a task [46]. All the distributed resources in a cluster act as one consolidated system due to the tight coupling of individual machines, whereas grid computers consist of individual machines that are autonomous and act as independent entities.

To support big data applications such as large-scale services, data analytics, and data-intensive scientific applications, companies like Google, Amazon, Microsoft, Facebook, Twitter, Yahoo, and others use cluster computing [47, 38]. Following the success at these companies, cluster computing has enjoyed a surge in interest and adoption [37].

In production environments, where systems are being used for business, cluster computing is more attractive than grid computing due to several reasons. Clusters have a lower financial cost because they are made up of commodity hardware. Clusters have a range of widely available programming frameworks. The components of clusters are widely available off-the-rack hardware. Clusters are highly scalable. Clusters are easily configurable and upgradeable due to their centralized nature.

Besides the above-mentioned business benefits, cluster computing also appeals to system designers and application developers. System designers prefer clusters over grid computers due to their lower cost, higher availability of components, flexibility, and ease of configuration. Application developers prefer cluster computing, due to the diverse range of available open-source programming models and a relatively simple software development process.

Developing applications for big data systems is a complex process. To simplify this process, developers use cluster programming models or frameworks. These frameworks (also referred to as cluster managers) manage the resources of the cluster and are responsible for scheduling and executing software applications (also known as jobs). A typical workload submitted to these clusters at production data centers [44, 50] consists of hundreds of thousands of jobs submitted and run daily. These jobs can be broadly categorized into the following two groups:

1. Workflows: These are production jobs[1] that are submitted to the cluster at regular intervals by automated systems [43, 26]. They are designed to process data feeds, refresh models, and publish insights. These jobs have high resource demands, longer runtimes, and are highly data-intensive. They are crucial to businesses and must be completed before the accompanying deadline.[2]

2. Best-effort jobs: These shorter ad-hoc jobs are submitted to the cluster by system administrators, data scientists, and application developers. They are small queries that require a statistically significant low amount of resources and have shorter runtimes. However, the number of best-effort jobs submitted is considerably larger than production jobs. These jobs are latency-sensitive instead of deadlines.

The ratio of production jobs to that of best-effort short jobs in a cluster varies widely. Best-effort jobs can be as many as 95% of all jobs submitted. However, 90% of the resources are consumed by production jobs [50, 12, 14].

Cluster computing architecture is built to analyze and generate insights from massive amounts of data. Although the commodity hardware costs have significantly dropped in the last two decades, the cost of building and operating a production data center still runs in millions of dollars. As these data centers become widely adopted, maximizing the return on investment becomes increasingly important.

[1]From this point forward, "workflows" and "production jobs" are used interchangeably
[2]These deadlines are called Strict Service Level Agreements (SSLAs)

1.2 State of Affairs

Production workflows are critical to businesses. Missing a deadline can have a significant financial impact. Maximizing cluster throughput has been a focus of existing big-data systems [51, 50, 15, 25]. Their sharing policies are based on priority, fairness, and capacity. Although prioritizing production workflows over best-effort jobs helps the chances of completing them before their deadlines, it increases the latency for best-effort short jobs. Prioritizing best-effort short jobs improves their latency, decreasing the odds of production workflows completing before their deadlines. In both cases, we encounter head-of-line blocking,[3] which prevents these systems from fulfilling the requirements of both types of jobs. Especially, no guarantees can be made on job allocation over time. This means that some jobs could stay in the queue indefinitely.

There are two ways existing big data systems cope with the problems mentioned above. They either include extra resources (which costs money and consequently is harmful to the return-on-investment) or by having labor-intensive workarounds: cluster operators can manually schedule job submissions to cope with these limitations. Another way these systems cope with the limitations is to have personnel ensure workflows complete before their deadlines by monitoring and freeing the resources being consumed by the best-effort jobs. This is burdensome for larger businesses and simply too expensive for smaller businesses. Furthermore, these problems worsen in public clouds like Amazon Web Services cloud, Azure, etc. Public clouds operate at a larger scale, accepting a larger number of jobs, servicing a more significant number of users.

Modern big data systems do not have isolated clusters running single application frameworks (like MapReduce) anymore [11]. Furthermore, they typically have a diverse mix of applications running at any given point in time [15, 25, 44, 50, 51]. This introduces further challenges of supporting inter-job dependencies and inter-task dependencies in complex workflows with DAGs of jobs that, in turn, have DAGs of smaller jobs or tasks.

1.3 Deadline-Aware Reservation-based Scheduling

This research presents a Deadline-Aware Reservation-based Scheduling System (DARSS). DARSS improves the rate of timely completion of production jobs, the throughput of best-effort jobs, and the latency for best-effort jobs when compared with existing systems. To build a scalable system, DARSS is divided into four stages. Each job submitted to the system that gets accepted by the system goes through all four stages throughout its life cycle. These stages are

- Configuration: where the client submits the job along with its requirements,
- Analysis: where the system calculates the resource requirements of the job and analyzes the available capacity of the cluster according to the system schedule;

[3]Head-of-line blocking is a phenomenon that occurs when the first job in a queue of jobs takes too long to complete limiting the performance of the system

- Reservation: where the system either accepts or rejects the job based on the analysis stage and reserves the resources required by an accepted job, and

- Scheduling: where the system allocates resources to jobs according to the schedule.

As mentioned in section 1.1, there is a mix of jobs submitted to a production cluster. This mix includes best-effort short jobs, which are interactive, and highly complex workflows containing interdependent jobs. Clients of the production clusters need to communicate the requirements of each job. To ensure effective communication of the requirements of these jobs, Requirement Specification Language (RSL) is required. Using RSL, clients can communicate the requirements of jobs from the most complex workflows to the more straightforward ad-hoc jobs. This job submission is made in the configuration stage.

The information provided in RSL is used to perform job admission control in the analysis stage. The system maintains the schedule in the form of a reservation file that logs the resources reserved for a job. Each time a job request is submitted, the system searches the reservation file for the total number of unreserved containers from the time of submission until the deadline for the workflow. The system's admission control either accepts or rejects a job request based on the available resources and the resource requirements of the job.

The system maintains a best-effort job quota to make sure that the production jobs or workflow do not starve best-effort jobs. This means that there are some resources reserved for best-effort jobs periodically. If a job fits in the schedule, it gets accepted, and the system updates the reservation file. Otherwise, the job is rejected. Instead of reserving resources for a specific time, the system reserves the number of resources required by a production job to finish before its deadline. This allows maximum fluidity and flexibility in scheduling. All of this is done in the reservation stage.

During the Scheduling stage, the system rearranges and schedules individual tasks belonging to the already accepted jobs. Task scheduling is done in the scheduling stage with the help of a ttd (time-to-deadline) list. The system maintains a list of deadlines for all the individual tasks to support fluidity in scheduling individual tasks. This list is based on the inter-dependencies of the tasks and the deadline for the job.

1.4 Objectives

This thesis has the following objectives toward understanding and improving the execution of jobs with temporal requirements in a cluster:

- Develop a taxonomy to explain the current state of affairs in cluster scheduling and resource management;

- Survey the existing mainstream cluster scheduling systems based on the taxonomy;

- Devise a requirement specification language capable of translating requirements of complex workflows;

- Devise a task-based scheduler capable of scheduling tasks from production workflows and best-effort jobs;
- Design a system that provides high throughput and low latency for best-effort jobs, high acceptance rate, and 100% completion rate (for accepted jobs) for production jobs;
- Prototype and implement the designed system and experimentally evaluate it.

1.4.1 Thesis Statement

To address the limitations mentioned in section 1.2, I have designed and developed the Deadline-Aware Reservation based Scheduling System (DARSS). DARSS is a system capable of supporting the following criteria:

- Complex Workloads: handling workloads consisting of multiple interdependent jobs;
- Production Job Deadlines: completing accepted complex production workflows before their deadlines;
- Best-effort Job Latency: providing lower latency for interactive short jobs that do not have deadlines.

1.4.2 Evaluation

To evaluate the presented design and compare it with the popular cluster scheduling systems, DARSS is evaluated using simulation based on the configuration of Google's production cluster [40]. DARSS's performance is evaluated across four dimensions:

- Production job acceptance: percentage of production workflows accepted by the system;
- Production job fulfillment: percentage of accepted workflows completed before deadlines;
- Best-effort jobs throughput: number of total best-effort jobs completed by the system;
- Best-effort job latency: service time for best-effort jobs.

DARSS is evaluated using a simulated cluster of 300 nodes. Experiments were conducted considering four scenarios where production jobs to best-effort jobs ratios were different. The ratios used in the experiments were 80% best-effort jobs and 20% production jobs, 85% best-effort jobs and 15% production jobs, 90% best-effort jobs and 10% production jobs and 95% best-effort jobs and 5% production jobs. These ratios were based on real-life production cluster workloads [40, 16]. Our experiments show that DARSS improves the production job acceptance by up to $\sim 67\%$ over Rayon (when 20% of all jobs submitted are production jobs). This means that more production jobs are accepted and finished before their deadlines than Rayon. DARSS also improves the production job fulfillment by up to $\sim 76\%$ over YARN (when 20% of all jobs submitted are production jobs). The highest increase in best-effort jobs throughput achieved by DARSS is an 18% increase in throughput relative to YARN (when 80% of all jobs are best-effort jobs). This means that more best-effort jobs are completed than YARN. Rayon achieves a slightly higher best-effort job throughput than DARSS for scenarios where the percentage of best-effort jobs submitted is lower than 95%. This happens due to Rayon's higher production job rejection rate relative to DARSS. DARSS reduces latency for up to $\sim$ 60% of best-effort short jobs compared to YARN (when 80% of all jobs are best-effort jobs). This means that 60% best-effort jobs are serviced quicker than YARN. DARSS reduces latency for up to $\sim$ 9% of best-effort short jobs compared to Rayon (when 80% of all jobs submitted are best-effort jobs).

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the background and a survey of existing big-data cluster scheduling systems. Chapter 3 describes the architecture and implementation details of our Deadline-Aware Reservation-based Scheduling System (DARSS). Chapter 4 presents the experimental work and results. Finally, Chapter 5 details conclusions and future work.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter presents the background and related work for this research. Section 2.1 presents a brief overview of big-data computing environments. Section 2.2 summarizes some salient characteristics of modern big-data workloads. Section 2.3 presents a classification of cluster scheduling systems for big-data processing frameworks. Finally, Section 2.4 details some of the main cluster scheduling systems.

2.1 Big-Data Computing Environments

Most modern big-data processing frameworks utilize cluster computing to process large volumes of data efficiently [28]. In a cluster computing environment, computers within a network work together as a single large system. The computers within the network do not operate as separate entities but as a worker of the system as a whole. Each node has a specific task to do. These tasks are scheduled by a system-wide scheduler. In some ways, this is similar to an ant colony. Individual ants do not work for themselves but for the colony. Each ant has a job to do which serves the colony as a whole. This is different from how grid computing works. In grid computing, each node can operate independently of the system as a whole and is "loosely linked" by the Internet or low-speed networks.

The following characteristics of types of resources and their arrangement in production clusters are typical components of Big Data Computing:

- *Jobs:* Users submit applications in the form of jobs to modern clusters. The jobs submitted to the cluster come from multiple users and are heterogeneous [48]. This mixture of these jobs varies from best-effort short jobs to long-running time-sensitive production jobs. These jobs also have varying resource requirements. Each job can be composed of multiple smaller tasks, which can be interdependent and heterogeneous or independent and homogeneous. The requirements specific to a job (resource requirements, temporal requirements, task interdependence, fault-tolerance requirements, priorities, etc.) are communicated to the cluster at the time of job submission as part of the job definition [21].
- *Commodity clusters:* Most modern big-data processing frameworks utilize commodity clusters. These clusters are built using inexpensive commodity machines, each of which can fail. Since most big-data processing frameworks have fault tolerance built into them at the software level, organizations achieve a cost advantage by using commodity hardware in their big-data processing clusters [4]. The use of

commodity clusters by organizations has led to big-data framework designers building a high level of fault tolerance into their software [51].

- *Large number of nodes:* Large web-service providers like Amazon, Microsoft, Google, Facebook, and Twitter power their services with clusters made up of hundreds of thousands of machines [6, 50]. Furthermore, organizations that do not provide web services are also continually building larger clusters since cloud computing and publicly available cluster computing frameworks are rising. The design of cluster scheduling systems is directly impacted by this trend of increasing cluster sizes. Since the complexity of scheduling algorithms is directly associated with the number of machines on which the jobs would be scheduled over, the increase in the number of machines directly impacts the design of the scheduler. Furthermore, the requirements related to multi-tenancy start to become more demanding as the size of the cluster increases. This mostly translates into an increase in the rate of job arrival [30].

- *Worker nodes:* Machines in a cluster that can be used for deploying and executing tasks are called worker nodes. Each of the worker nodes has a worker agent that is responsible for collecting and reporting local resource consumption metrics to the master node, starting and stopping tasks deployed on the node, and managing local resources.

- *Cluster Manager - Master Node:* The core of the framework is a Cluster Manager or Master Node. The cluster manager is made up of several important components. One of the important components of the cluster manager is an admission control module that decides if a submitted job is to be accepted by the cluster based on the user's resource quota and available cluster resources. One way of deciding which jobs to accept and which to reject could be rejecting jobs simply if the available cluster resources are insufficient. Most modern cluster managers, however, use a more sophisticated approach where jobs are preempted to free resources [41] for other more *important* jobs based on the users' priorities and Quality of Service (QoS)[1] classes.

  The scheduler, is responsible for mapping jobs onto cluster resources. This mapping is done by considering the following criteria:

  - resource requirements and availability;
  - maximizing cluster resource utilization;
  - QoS class requirements.

  The *task relocator* is responsible for relocating tasks if and when they are preempted. One way of deciding where to relocate a preempted task could be to either place the task at the back of the queue or discard it. Most modern cluster managers [51, 25, 15, 50], however, use a more sophisticated approach where maximizing cluster utilization is at the center of the analysis done to find an optimal plan to

[1]The description of the performance of a service

relocate the task. Another component, the *task monitor*, also aids in the analysis done for determining the optimal task relocation. The task monitor manages the resource consumption of running tasks. Based on the decision by the scheduler or the task relocator, the *task launcher* launches task containers on the specific worker nodes. The *accounting module* and the *resource monitor* tracks the actual resource usage and consumption metrics. The resource monitor is also responsible for sharing this information with other modules of the cluster manager. Lastly, the *resource provisioner* is responsible for the addition of new cluster nodes. The addition of new nodes can either be a manual or an automatic process. In the manual process, the system administrator is responsible for launching a new node with worker node software already on the node. In the automatic process, the resource provisioner is responsible for automatically adding new virtual nodes and for shutting down virtual nodes when they are no longer needed.

- *Heterogeneous resources:* It is not always feasible to have all the machines in a cluster be of the same hardware configuration. Large clusters often have heterogeneous hardware resources (such as the number of disks, amount of memory, number of cores, etc.) deployed in their clusters. This heterogeneity is not restricted to hardware; individual machines in a cluster can also have different operating systems or different versions of an operating system running on them. The organizations with the largest clusters – like Facebook [15], Google [51], Cloudera [13], Microsoft [29], and others have heterogeneous cluster environments. Since the individual machines that make up the clusters get replaced over time with newer and faster machines, a large cluster tends to employ machines from multiple generations with different configurations. Almost every year, a new set of hardware is released with a new generation of processors, memory, and storage drives with increased capacity, etc. This means that each time a node is added to the cluster, a new generation of hardware is incorporated into the cluster, which continues to have other machines with older generations of hardware. This introduces a cross-generation heterogeneity that directly impacts the design of cluster scheduling systems [30].

- *Commodity network infrastructure:* Designing the network infrastructure for a cluster involves a choosing network devices and hierarchy. This leads to a trade-off between speed, scalability, and cost of the network. Most data centers use an Ethernet network organized as a two-level hierarchy [6]. The bisection bandwidth per node in such networks is lower than that of out-link from each node. This out-link bandwidth is driven by off-rack communication traffic [30]. Recent studies have significantly reduced these over-subscription ratios. Due to this, full bisection bandwidth can be achieved even in the case of high traffic [1]. Even in the case of cloud instances of clusters, service providers provide "cluster placement groups" that allow full bisection bandwidth of 10 gigabits between cloud instances in the cluster placement group [2]. This means that achieving full bisection bandwidth can significantly reduce the impact of data locality in the future of cluster computing [3, 54].

- *Power saving techniques:* Even the most widely used power saving methods of yesteryear such as turning the idle servers off have been rendered ineffective in the modern clusters. This is due to the role of modern clusters. Clusters these days are utilized as data storage systems. This means that turning off idle servers can cause data unavailability [51]. Another option is cluster-level approaches such as covering set [32]. This kind of approach exploits the replication property of distributed file systems for turning off the servers. Unfortunately, they increase execution times, over-provision some of the servers, and require modification of distributed file system [31] which makes this kind of approach undesirable. The best approach for power efficiency of a cluster is server-based solutions such as PowerNap [35]. This kind of a solution works in nodes' local operating systems. PowerNap focuses on minimizing idle power and transition time. PowerNap enables the entire system to transition between two coarse-grained power-performance states instead of introducing fine-grained power-performance states. The two states are a high-performance active state and a near-zero-power idle state.

- *Resource managers:* Resource managers used in most popular big-data systems [15, 25, 44, 50, 51] utilize resource sharing mechanisms that are not time-aware. For example, Google's Borg [51] uses priorities to decide cluster sharing policies. Mechanisms like queues, pools, priorities, etc., are commonly used to share resources in big-data systems [16]. Consequently, trying to express temporal requirements using static concepts such as capacity, fairness or priority leads to a trade-off between cluster utilization and latency and deadlines. On the one hand, prioritizing cluster utilization can lower operational costs at the expense of user satisfaction (predictable execution times). On the other hand, prioritizing user satisfaction by static partitioning of resources increases high operational costs. This situation, while not desirable, may be tolerable for long-running batch jobs that are flexible. However, this is unsustainable for production jobs with strict temporal requirements, an all-too-common scenario in most big-data systems [56].

2.2 Characteristics of Big-Data Workloads

In this section, we summarize some salient characteristics of modern big-data workloads from public information that appeared in recent reports [11, 12, 14, 7, 20, 10]. We use this to derive high-level requirements.

Big-data workloads running in most clusters can be classified as Production jobs and Best-Effort jobs. The ratio of production jobs to best-effort jobs varies from cluster to cluster. However, production jobs consume most of the resources in most clusters [16]. Complex workflows with jobs that require all their tasks to run at the same time are also a common occurrence at most clusters. It is vital to integrating these workloads is to increase the return-on-investment. However, it is challenging given the existing approaches to scheduling workloads [16].

2.2.1 Production Jobs

Production jobs or workflows are long-running jobs that have strict temporal requirements. Production jobs are typically made up of multiple interdependent jobs, which in turn can be made up of smaller interdependent tasks. Production jobs typically have extreme peak-to-average ratios [7, 20]. Due to the high difference between peak resource requirements and average resource requirements, using static resource allocation mechanisms can lead to extremely poor cluster utilization and, consequently, higher operational costs. To honour a Service-Level Agreement,[2] a cluster operator may have to allocate resources based on peak resource requirements. This leads to unused resources that could be temporarily allocated to run other jobs at non-peak times. However, no guarantees can be made on latency or execution times.

2.2.2 Best-Effort Short Jobs

Best-effort jobs are short jobs that do not have strict temporal requirements. However, they are sensitive to completion latency. Best-effort jobs are typically ad-hoc interactive jobs that are high in number but are significantly less resource-hungry [16]. The difference between peak and average resource requirements is also substantially less. Static resource allocation based on peak resource requirement for production jobs takes away resources from best-effort jobs, which leads to best-effort jobs taking longer to complete. If a best-effort job takes too long to complete, it becomes *stale*. Low completion latency is important for cluster operators to maintain user satisfaction. These characteristics point to the need for a framework capable of supporting complex workflows with inter-stage dependencies, predictable completion times for production jobs, low latency for best-effort jobs, and high cluster utilization.

2.3 Taxonomy

2.3.1 Application Model

This subsection details a classification for the application model used by the cluster management system as depicted in Figure 2.1. This classification is related to the user-submitted *Jobs* mentioned in 2.2. The different ways jobs can be described by the users and the different characteristics pertaining to jobs are also presented in this subsection.

The two ways to run applications or jobs are 1) to run applications on separate clusters of homogeneous containers, or 2) running different types of applications, each in its own container on a shared cluster. Most organizations prefer the latter to utilize resources efficiently. A common type of job is a production job that are long-running services that require high availability and must handle latency-sensitive requests. Web services, user-facing websites are examples of long-running jobs. Another type of job is a best-effort job that

[2]A contract between a cluster operator and a client dictating quality, availability, and responsibilities of the service

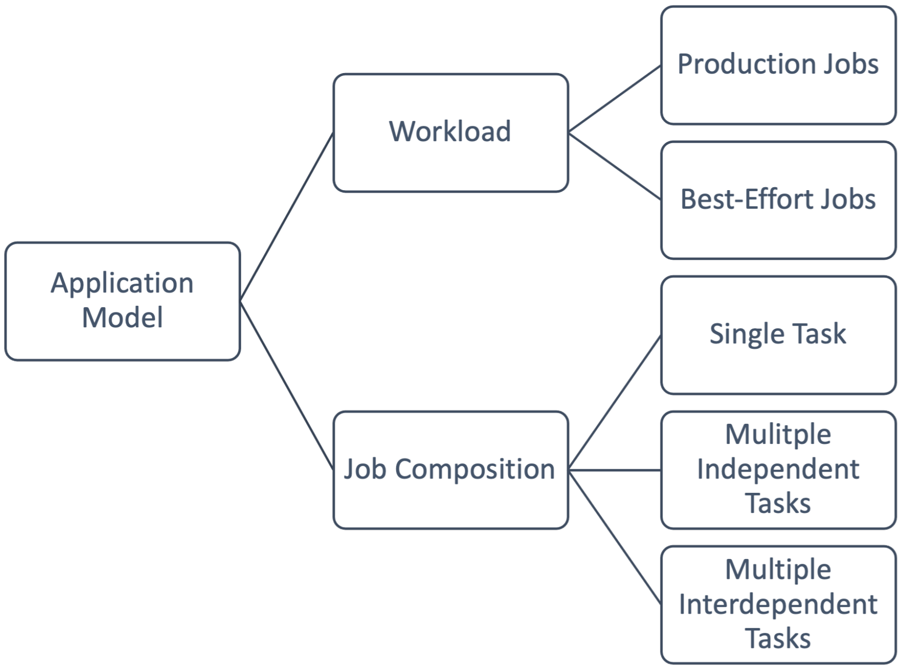


Figure 2.1: Application model taxonomy

have a limited lifetime and are more tolerable towards performance fluctuations. Scientific computations submitted by data scientists, engineers testing or debugging ideas, and MapReduce jobs are some examples of best-effort jobs. Another key aspect of jobs is the makeup of jobs. Typically production jobs are made up of several interdependent or independent tasks, whereas best-effort jobs can be composed of a small number of independent tasks or even a single task.

Although supporting a workload that consists of the different types of jobs with different SLAs and QoS requirements presents challenges for big-data frameworks, having such a workload running on a cluster has significant benefits in terms of resource utilization. According to Google [27], if they had production jobs and best-effort jobs running on separate clusters instead of one cluster running mixed workload, they would need 30% more machines to run the equivalent workload. In the case where a mixed workload is running on a single cluster, best-effort jobs can use the resources that are allocated to but not used by the production jobs. Best-effort jobs can tolerate being placed in a less resource-rich node than those requested for the job. Therefore, better job packing can be achieved.

2.3.2 Scheduling

This subsection details a classification for schedulers used by the cluster management system as depicted in Figure 2.2. The difference in scheduler architecture, how that impacts job placement, and advantages and disadvantages of each architecture are also presented in this subsection.

Several studies show that the architecture of the scheduler is the most important characteristic in understanding the different techniques used in scheduling jobs in cluster management systems. Krauter et al. [8],

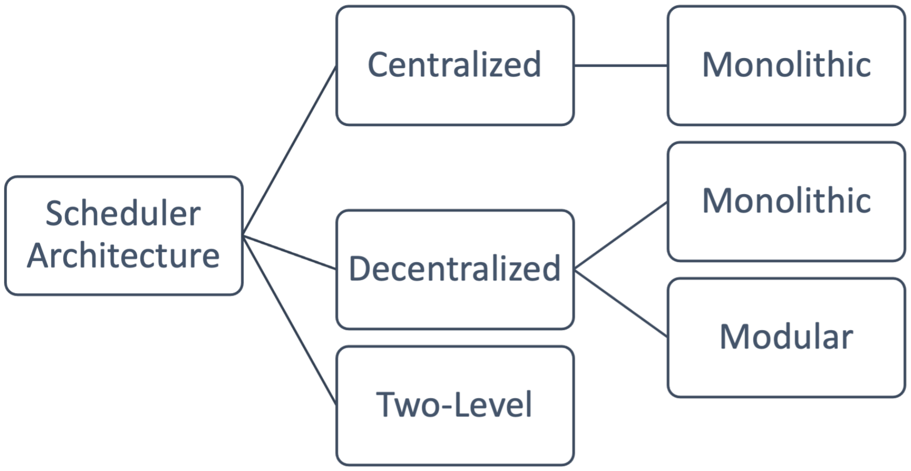


Figure 2.2: Scheduler architecture taxonomy

Sabuncuoglu et al. [21], Kuhl et al. [51], and Rodriguez et al. [41] have all included the scheduler architecture when classifying the cluster management systems in their studies. The three most prevalent types of scheduler architecture found in most modern big-data systems are centralized architecture, decentralized architecture and two-level architecture.

In a centralized architecture, a single scheduler is responsible for scheduling jobs on cluster nodes. The scheduler is also responsible for implementing all the policies pertaining to the job based on SLAs and QoS for that job. This property makes the scheduler a monolithic centralized scheduler. A centralized scheduler has a consolidated view of the whole cluster and its resources. It can place a job on any available node in the cluster, which helps in optimizing the job placement. This also helps in relocating the jobs in case of preemption. A single centralized scheduler responsible for all these tasks also makes it a single point of failure in the system. Another disadvantage of such an architecture is poor scalability in the case where one or both of incoming scheduling requests and the number of cluster nodes increase.

In a decentralized architecture, multiple instances of a distributed scheduler exist on different nodes. These scheduler instances can be monolithic or modular. A monolithic decentralized scheduler instance is responsible for a subset of jobs requests but implement all the policies and handle all the jobs in the subset. Each modular decentralized scheduler instance handles a specific type of job and/or implement a different set of policies than other such instances. For instance, there can be a scheduler for production jobs and a different scheduler for best-effort jobs. Partitioning the workload between the decentralized schedulers depends on whether the schedulers are monolithic or modular. Partitioning the workload between the monolithic decentralized schedulers can be done by traditional load balancing techniques. Partitioning the workload between the decentralized, modular schedulers can be done simply based on job type.

In a two-level architecture, scheduling is done over two separate layers. In such a system, resource man-

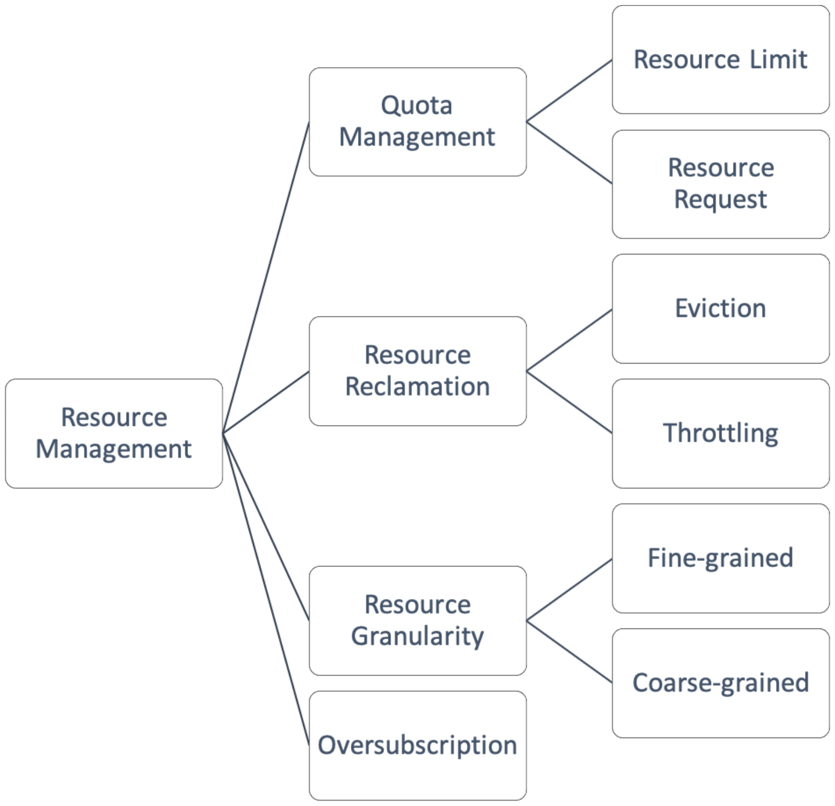


Figure 2.3: Resource management taxonomy

agement is handled separately from application framework and job placement. The top layer is responsible for resource management. The top layer divides the resources among the application frameworks running on the bottom layer. The resources are given to application frameworks based on either a resource request or a resource offer. The application frameworks running on the bottom layer are responsible for scheduling jobs on the resources they are offered.

2.3.2.1 Reservation-based Scheduling

Reservation-based scheduling is an approach that reserves resources ahead of time, based on the temporal requirements of the incoming jobs. This allows the delivery of time-predictable resource allocations required for meeting production job deadlines while minimizing latency for best-effort jobs.

2.3.3 Resource Management

This subsection details different resource management techniques used by the cluster management system as depicted in Figure 2.3.

Most modern big-data systems are multi-tenant environments which means that they cater to multiple users on one or more physical clusters. Managing cluster resources in such systems is a challenging problem. As discussed earlier, jobs are made up of one or more tasks. Cluster management systems typically handle resource management by allowing users to define the computational resources (CPU and memory) that a particular task will require at its peak time in the job definition submitted to the cluster. The task resource requirement is used by the system to determine which jobs will be assigned to which machines based on the available resources on each machine. A task, however, cannot request unlimited resources. There are limits on the amount of resources a task is allowed to consume. These limits are based on the SLAs and QoS classes and are user-specific. Cluster management systems enforce these limits by reclaiming resources by either throttling the resource or simply stoping the task and putting it in the queue to be rescheduled.

The level of granularity at which resources can be allocated to and requested by a task depends on the system. Some systems use fine-grained resource granularity, whereas others use coarse-grained resource granularity. Systems that use coarse-grained resource granularity generally allocate resources in fixed-size coarse resource chunks. YARN, for example, allocates resources in increments of 1 virtual core and 1 megabyte of RAM for CPU and memory, respectively. A user can request multiples of units for a task. Systems that use fine-grained resource granularity are much more flexible and give users more control over the amount of resources they can request.

To increase cluster utilization, many systems oversubscribe their resources. This is done due to the following reasons. The resource request submitted by the user is based on peak resource requirements. A task does not need the requested amount of resources throughout its lifetime. As noted in section 2.1, the difference in peak resource consumption and average resource consumption is common amongst the jobs submitted to big-data systems. The other reason systems oversubscribe their resources is that it is not uncommon for users to overestimate the resources required for a task.

The overestimation of resource requirement by users and varying resource consumption of tasks lead to reserved resources staying in an idle state for long periods of time. This leads to poor cluster resource utilization. To address this problem, cluster management systems use resource consumption estimation to predict the amount of actual resources that will be used by a task over its lifetime. Cluster management systems can over-subscribe their resources by accurately estimating the actual resource consumption of tasks leading to better cluster utilization.

2.4 Survey of Existing Systems

This section discusses some of the state-of-the-art cluster scheduling systems and analyzes them in the context of the presented taxonomy.

2.4.1 Centralized Scheduling Systems

Among the surveyed systems, Google's Borg is the only system with a centralized monolithic scheduler architecture.

2.4.1.1 Borg

One of the most advanced and widely used cluster systems is Google's Borg [51]. Borg has a high average cluster utilization rate of approximately 70% while cluster throughput is approximately 15 thousand tasks executed per minute across its several physical clusters [48]. Furthermore, Borg supports most of the advanced features detailed in 2.3 such as

- Handling mixed workloads containing different types of jobs;
- Allowing fine-grained resource requests;
- Task preemption and relocation; and
- Cluster over-subscription to maximize cluster utilization.

Borg runs hundreds of thousands of jobs from a mixed workload submitted to Borg across its several physical clusters of heterogeneous machines that reside in different geographic locations. Each cluster is composed of tens of thousands of heterogeneous machines. Figure 2.4 shows the high-level architecture of Borg.

The jobs in Borg's workload can be coarsely categorized as production jobs and best-effort jobs. Production jobs are long-running and high-priority jobs that are sensitive to availability and latency. Production jobs are usually user-facing web services that run for long periods of time. Best-effort jobs are low-priority jobs that can take from a few seconds to days to complete. Best-effort jobs do not have the strict availability and time-sensitive requirements that production jobs do. Both types of jobs are composed of one or more tasks. Tasks from jobs run inside containers that are deployed on physical machines. A container is a self-contained unit of software that packages the code and all its dependencies. Required resources for tasks are specified with containers in terms of CPU, memory, storage, and network requirements. Jobs submitted to Borg run on Borg *cells*. A Borg cell is a collection of heterogeneous machines that are managed as a unit. A physical cluster generally contains one large cell. The median size of a Borg cell is ten thousand machines, excluding test machines [51].

Borg's scheduler maintains a queue of tasks that is monitored asynchronously. Tasks in a queue have priorities assigned to them. The highest priority tasks are scheduled first. If two or more tasks have the same priority, round-robin scheduling is used to schedule the tasks. Scheduling is done in two phases: feasibility checking and scoring. First, the scheduler tries to find all the available machines that match a task's requirements. Second, the scheduler chooses one of the machines short-listed in the first phase using a scoring mechanism. The scoring mechanism considers the following criteria:

- User-specified preferences;
- Choosing machines that already have a copy of tasks' packages;

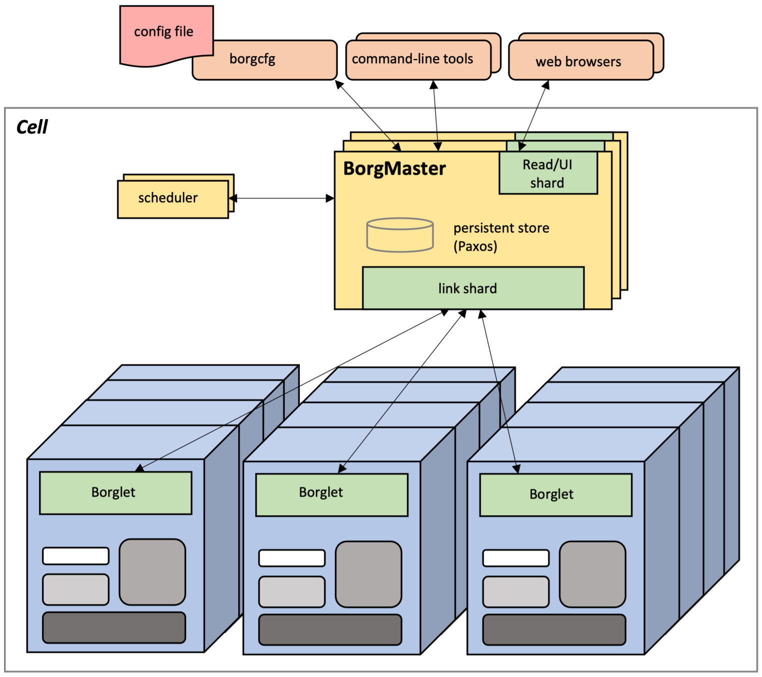


Figure 2.4: The high-level architecture of Borg [51]

- Spreading tasks across power and failure domains;
- Mixing low priority and high priority tasks scheduled on a machine to allow high priority tasks to consume more resources during peaks; and
- Minimizing preemption of tasks.

Although every job has a job definition with task resource requirements, Borg does not reserve resources based solely on resource requests. Instead, Borg uses a dynamic fine-grained resources consumption estimation for each task measured every few seconds. This estimation is used to dynamically reserve resources. Resources that are reserved for a task and are not in use are reclaimed. These reclaimed resources are then allocated to low-priority tasks. Tasks that use more resources than requested are handled in one of two ways. If the tasks are consuming more CPU power than requested, they are throttled to the requested amount without killing it. If the tasks are consuming more memory, they are killed to reclaim the extra memory.

2.4.2 Decentralized Scheduling Systems

There are several cluster management systems that have a decentralized scheduling architecture. These systems are presented in the following subsections.

2.4.2.1 Kubernetes

Kubernetes [21] is an open-source container-based cluster management system that is very popular among developers. Kubernetes enables users to deploy applications in self-contained packages, containers. Kubernetes

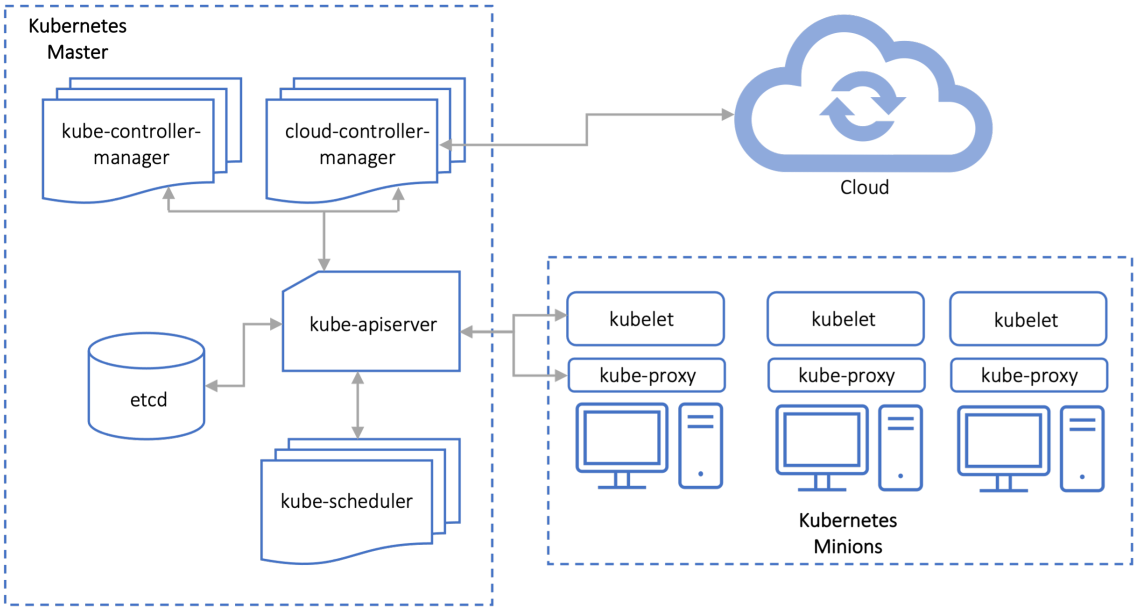


Figure 2.5: The high-level architecture of Kubernetes [21]

has an extensive development community support and development base. Like Borg, Kubernetes was also initially developed by Google. Although Kubernetes does not scale as well as Borg, Kubernetes is capable of supporting clusters of up to 5 thousand nodes as of version 1.20 [21]. Figure 2.5 shows high-level architecture of Kubernetes.

The most basic deployable object in Kubernetes is a Pod. Pods contain one or more containers that share the same set of resources and a specification for how to run the containers. Containers in a pod are always co-located and scheduled together. Containers in a pod also share storage resources and a network IP. A pod typically runs a single instance of a job. For scalability, multiple pods could be run. The resource requirements for a container are specified while creating a pod. This specification is used by the scheduler to make the pod placement decision. Resources required by a container can be specified in two ways; as a maximum amount of resources allowed for the container to consume or as a requested amount of resources.

The scheduler is also responsible for ensuring that a node's resources are not oversubscribed even if the actual resource consumption is low. This means that pods cannot request more resources than the capacity of the node they are placed on. Kubernetes does this to make sure that tasks do not run out of resources during peak consumption. Kubernetes stops a running container in some cases. For example, a container is stopped to be scheduled again if the container has exceeded its memory limit. If a node runs out of memory, containers that have exceeded their memory requests are stopped. However, exceeding CPU usage limits are handled a little differently. Because CPU usage differs throughout the life-cycle of a container, Kubernetes may or may not allow a container to exceed its CPU limit based on the node's CPU consumption. Kubernetes has a strict policy on exceeding storage limits. Containers that exceed the storage limits are evicted. The scheduler is also responsible for managing new resources added to the system.

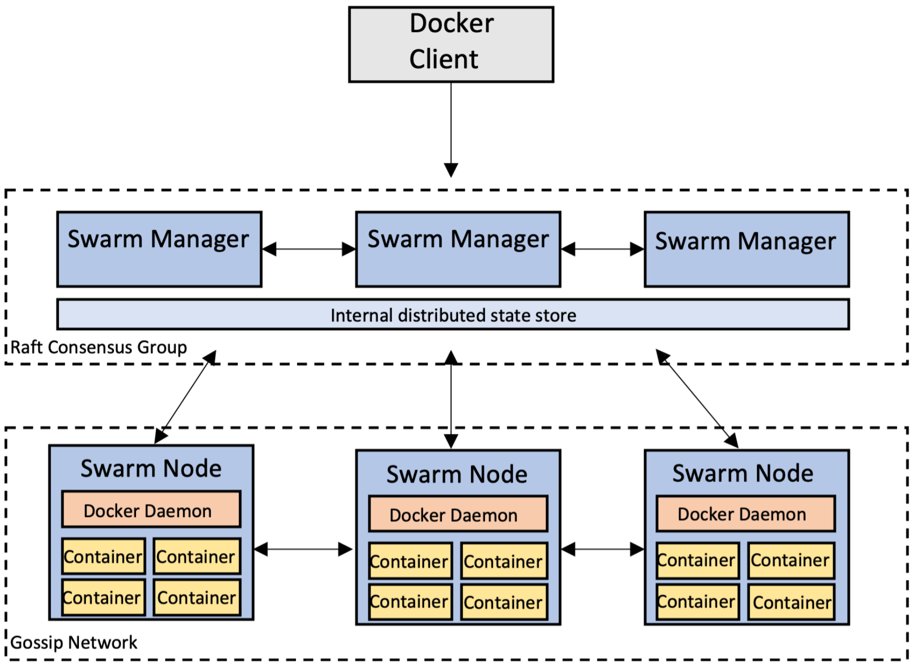


Figure 2.6: The high-level architecture of SWARM [18]

One of the reasons Kubernetes is so popular among developers is the use of a plug-and-play approach to scheduling. Although Kubernetes has a scheduler that can be classified as decentralized and monolithic, users are allowed to use a different scheduler instead. This allows Kubernetes users to manage and schedule their pods in a way that suits their needs. Kubernetes makes integration of custom schedulers easy, which makes it an attractive choice for users that have specific business needs.

2.4.2.2 Swarm

A very popular system among developers, Docker Swarm [18], schedules jobs on a cluster of nodes running Docker. Docker enables users to deploy applications in self-contained packages, containers. Swarm is a lightweight and flexible scheduler for Docker users. Swarm schedules Docker containers with an impressive performance in scalability and time it takes to make scheduling decisions [18]. This is possible due to Swarm's simple yet effective scheduling strategies. Swarm does, however, lack some of the key features that all other systems in this section support. One such feature is support for node failure. Most systems support node failure by maintaining instances of a task running on a different node until the task completes. Swarm, however, does not have a built-in fault-tolerance for node failures.

Figure 2.6 shows high-level architecture of Swarm. Users submit job requests to a Swarm Manager, which is responsible for the deployment and management of jobs on the cluster nodes. These jobs are composed of one or more Docker containers. These jobs are also represented in the system as long-running services. Although the failure of a service is handled by the Swarm, a user can replicate services manually. Swarm offers three scheduling strategies to deploy tasks.

- *Random:-* Container is deployed on a random node from the cluster;
- *Spread:-* Container is deployed on the node with the least amount of tasks running on it; and
- *Binpack:-* Container is deployed on the random node with least amount of available compute resources.

In addition to three different scheduling strategies, Swarm also provides its users the ability to change the placement behaviour of their schedulers specific to their needs. This is done with the help of placement constraints such as value, label, and affinity-based constraints.

2.4.2.3 Omega

Another cluster management system developed by Google is Omega [44]. Omega, like Mesos, offers flexibility to users and allows custom schedulers to be developed on top of its platform. This is due to Omega's parallel scheduler architecture. Omega maintains a centralized transaction-based data store that can be accessed by schedulers. This data store helps maintain the shared state of the cluster, which is important in an environment where multiple schedulers may be scheduling tasks.

Although this approach can lead to conflicts where multiple schedulers try to schedule tasks on the same set resources, Omega resolves these conflicts by using lock-free optimistic concurrency control. The optimistic concurrency control allows different schedulers to consider all available resources while making a scheduling decision and only reschedules the tasks when a conflict arises [44]. This is different from the pessimistic concurrency approach, where while one scheduler is considering a resource at a time, it holds a lock on that resource until the scheduling decision is made. Although the optimistic concurrency approach leads to some overhead, the additional work was found to be acceptable, given the benefits of resolving resource blocking conflicts and better scalability than the pessimistic concurrency approach [44].

2.4.2.4 Apollo

Microsoft's Apollo [8] is a complex distributed scheduler that aims to improve the scalability bottleneck of centralized while also improving the optimization of scheduling decisions by independent, distributed schedulers. Apollo does this by using a distributed and coordinated approach to task scheduling. Figure 2.7 shows high-level architecture of Apollo.

Apollo estimates execution times for tasks based on historical data for similar tasks. Unlike other systems, Apollo aims to minimize the task completion time. The task execution time estimates are used to achieve this goal. Another goal of the system is to achieve high cluster utilization. Apollo does this by using opportunistic scheduling. As discussed in section 2.3.3, most tasks do not consume the resources requested for peak times over their lifetime, which leads to under-utilization for the cluster resources. Opportunistic scheduling addresses this problem by opportunistically scheduling more tasks that execute using the resources that are not used by the already scheduled tasks. The system maintains the execution plan for jobs in the

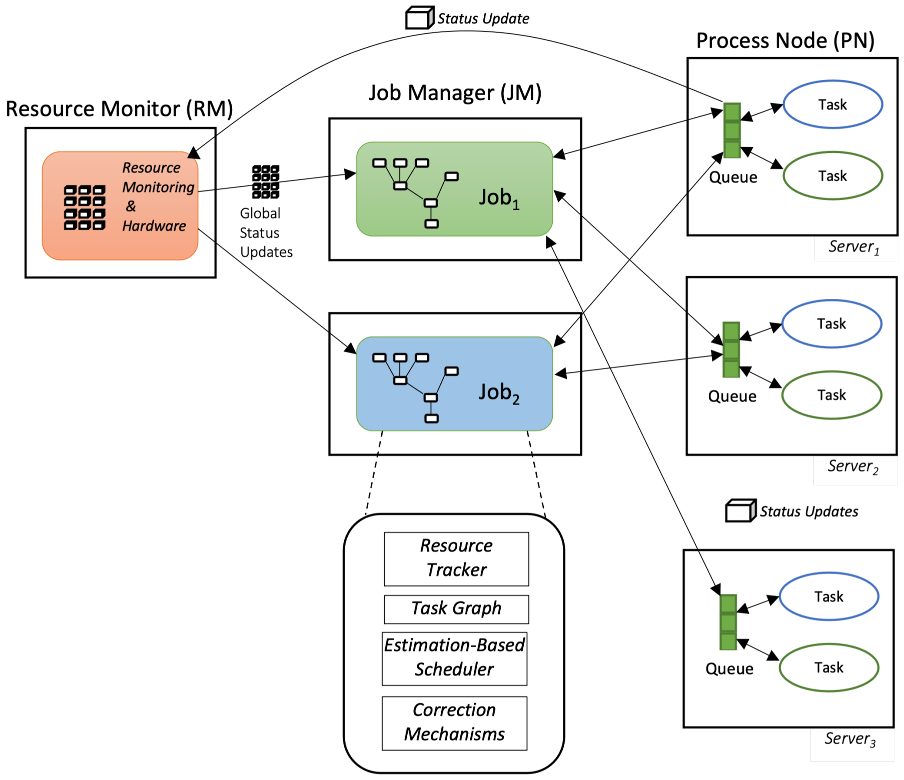


Figure 2.7: The high-level architecture of Apollo [8]

form of a *Task Graph*. A task graph is a DAG of jobs where tasks are represented by the nodes, and the flow of data is represented by edges in the graph.

2.4.3 Two-Level Scheduling Systems

Many of the systems surveyed have a two-level scheduling architecture. These systems are presented in the following subsections.

2.4.3.1 Mesos

Mesos [25] is an open-source cluster management system that provides flexibility to its users and applications. Mesos allows its users to deploy different application frameworks and schedulers on top of Mesos based on their business requirements. Different schedulers can use their own unique approach to task scheduling that they specialize in. Mesos also has high cluster utilization rates enabled by a fine-grained resource sharing approach and two-level scheduling. Furthermore, Mesos has high scalability due to its simplicity. Mesos can scale up to 50 thousand emulated nodes.

Mesos uses a two-level scheduling approach that enables it to work with other application frameworks. Application frameworks, like Hadoop [45], handle task scheduling while Mesos handles the resource distribution amongst the different application frameworks running on the cluster. On the first level, Mesos divides

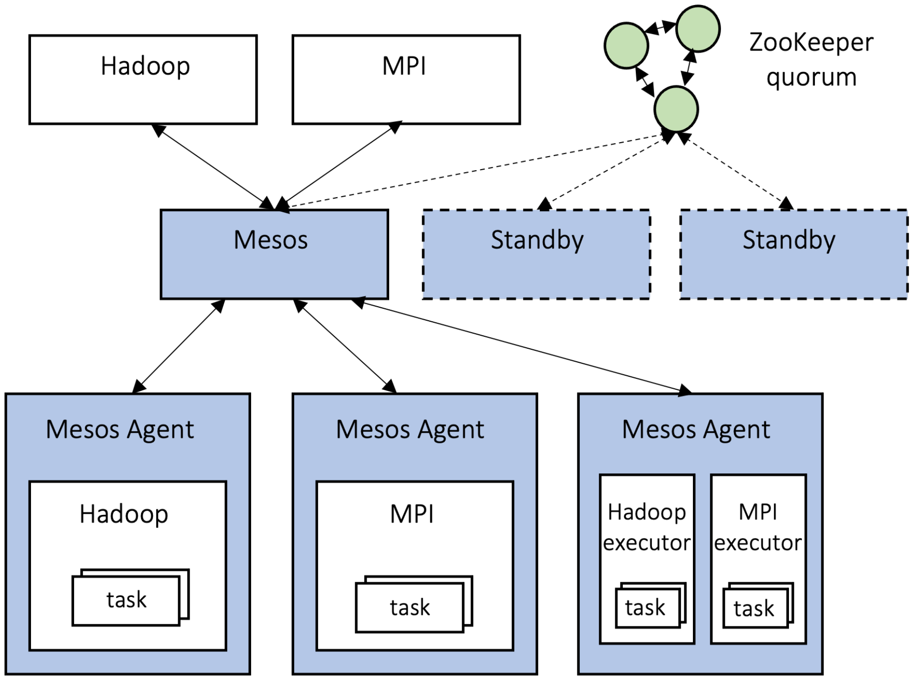


Figure 2.8: The high-level architecture of Mesos [25]

cluster resources based on priority and fairness policies and offers resources to each framework running on top of Mesos. These resource offers are made periodically, and each framework can either accept or reject the resource offers. When a framework accepts the resource request, resources are allocated to the framework. On the second level, the framework can use the allocated resources to schedule their tasks on them. When a framework schedules tasks to be deployed on a node, Mesos actually deploys tasks on the machines. Figure 2.8 shows how Mesos uses two-level scheduling.

One problem that may arise from using a two-level scheduling approach is resource hogging. A framework might have an excessively long scheduling cycle or schedule long-running jobs on the resources that will use the offered resources for a long time. This is not ideal in an environment where multiple frameworks share the cluster. Mesos addresses this issue by making its resource offers time-bound. The resources offered to frameworks have a maximum task execution time limit. If a task takes longer than the specified time, that task is stopped, and the resources are freed. This encourages frameworks to schedule short tasks while reducing the impact of necessary long tasks on the whole system. This time limit also encourages the frameworks to make faster scheduling decisions.

2.4.3.2 Aurora

Due to the flexibility and open-source nature of Mesos, many schedulers have emerged that run on top of Mesos. Many of these schedulers are typically developed by companies to cater to requirements specific to their business. One such scheduler is Aurora [5], developed by Twitter. Figure 2.9 shows high-level architecture of Aurora. Aurora enables production jobs and best-effort jobs to be run on a cluster. Since

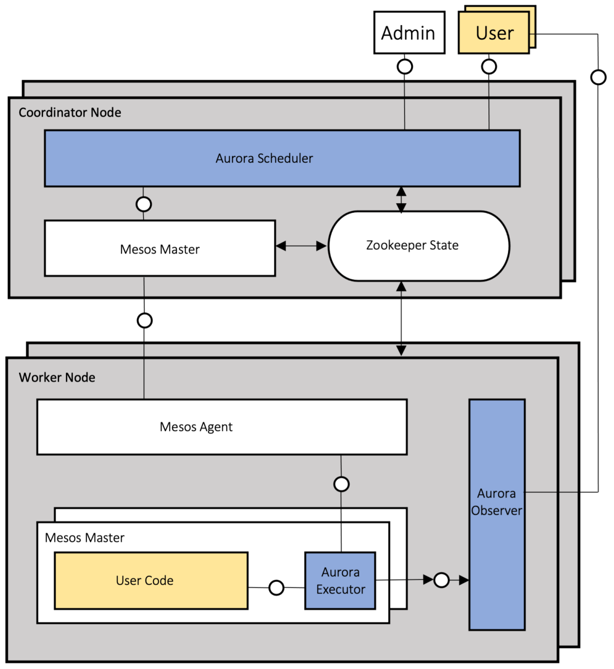


Figure 2.9: The high-level architecture of Aurora [5]

Twitter's business depends on the availability of its user-facing web services, fault-tolerance is one of the main features of Aurora. Aurora ensures that jobs pertaining to user-facing web services are kept running continuously. In case of a machine failure, Aurora swiftly reschedules the tasks that were running on the faulty machine onto an available machine. Unlike Mesos, Aurora supports the deployment of jobs made up of smaller homogeneous tasks. Each task in the job is made up of one or more processes. The Aurora Executor process that runs on each of the worker nodes is responsible for launching and monitoring tasks. Users submit jobs along by first submitting a job configuration. The job configuration contains details such as resource requirements and constraints for each task in the job. Depending on the requirements specified in the job configuration, Mesos makes resource offers which are then used to map each task onto specific resources.

2.4.3.3 Marathon

Another scheduler that is functionally similar to Aurora is Marathon [17]. Marathon runs on top of Mesos and specializes in providing production jobs with fault tolerance and high availability. Marathon, like Aurora, ensures that long-running production jobs are kept running continuously. Machine failures are also handled similarly to Aurora. Marathon also adds advanced features to Mesos. Features like task placement constraints, job progress checks, and monitoring, and auto-scaling jobs. Marathon also supports resource sharing between jobs running on the same physical machine. Marathon does this by allowing users to set placement constraints such that a set of jobs must run on the same machine. In many ways, this set of co-located jobs is similar to Pods in Kubernetes.

While Marathon is similar to Aurora, they have a few differences. First, Marathon only handles long-

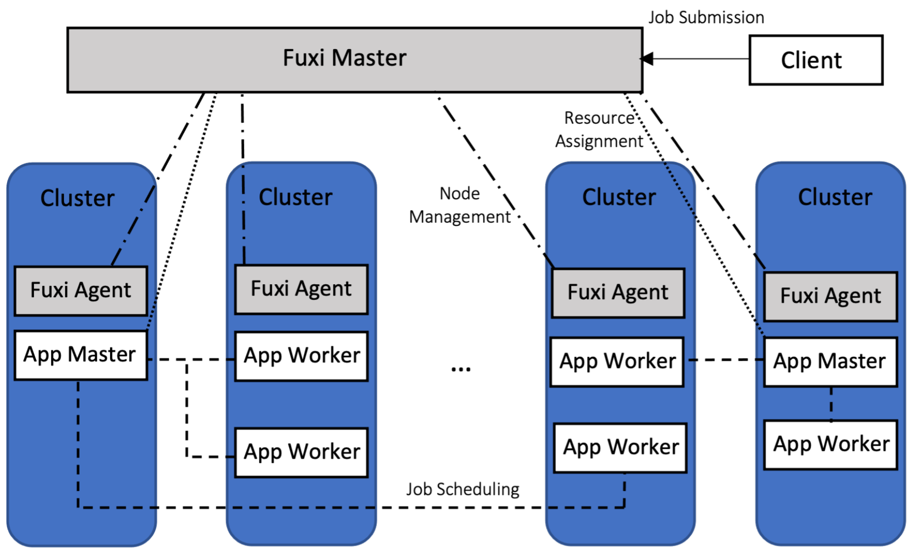


Figure 2.10: The high-level architecture of Fuxi [57]

running production jobs, whereas Aurora also supports best-effort jobs. Second, Marathon is less complicated to set up and use [49]. Furthermore, Marathon has a user-friendly interface that allows users to schedule jobs directly.

2.4.3.4 Fuxi

Fuxi [57] is a resource management and scheduling system developed by Alibaba. Fuxi is a part of a larger system, Aspara. Aspara manages the physical resources of clusters and the parallel execution of jobs that have to run in parallel. Figure 2.10 shows high-level architecture of Fuxi.

Jobs are submitted to the Fuxi Master along with the job definition, which contains all the necessary information. A Fuxi Agent runs on every node of the cluster. Fuxi Agents are responsible for launching a process called an Application Master for each job. These processes are responsible for determining the resource requirements for tasks in the job from the job definition and sending the resource requests to the Fuxi Master. If Fuxi Master accepts the resource request, the application master makes the task execution plans and sends them to Fuxi Agents. The application master is also responsible for tracking the job completion progress and returning resources back to Fuxi Master upon job completion.

Fuxi uses locality-based scheduling. The locality-based scheduling approach reduces the time spent on scheduling decisions compared to global scheduling approaches. In a locality-based scheduling approach, scheduling decisions are made on the node level whenever a resource is freed. This means that when a resource is freed, the resources are allocated to the tasks waiting in the queue on that node. This helps Fuxi make scheduling decisions in microseconds.

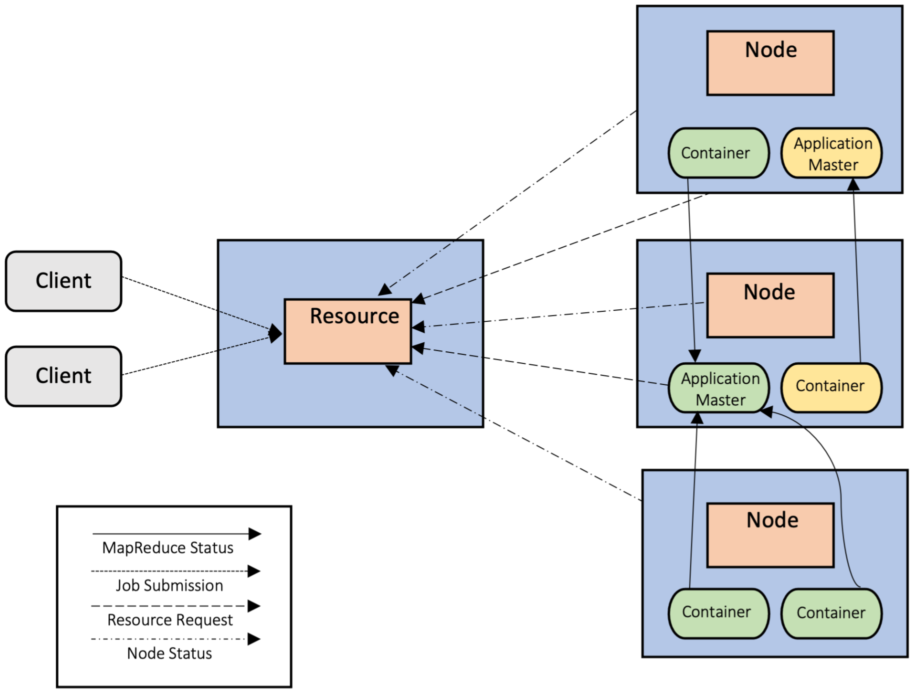


Figure 2.11: The high-level architecture of YARN [50]

2.4.3.5 Apache YARN

Hadoop remains a popular choice among developers and companies [49]. The high rate of adoption by companies that operate on different scales encouraged the next generation of Hadoop computing platforms. Over the years, several new requirements regarding cluster resource management and scheduling emerged that Hadoop's initial design could simply not support. Apache YARN [50] is a cluster manager that can schedule Hadoop tasks with improved scalability, reliability, and multi-tenancy. YARN is a very flexible system that supports other frameworks such as Spark, Giraph, and Storm to run on top. Each of these frameworks running on top of YARN can coordinate its own flow of execution. YARN also supports two types of containers: a custom built-in container based on Linux cgroups and Docker containers.

YARN delegates the programming model to per-job agents and scheduling duties to a per-cluster *Resource Manager*. This allows the programming model and scheduling duties to be handled by two separate entities. The resource manager is a central entity that is responsible for distributing resources to applications. It is also responsible for enforcing fairness, capacity, and locality policies. Similar to how the resource manager is responsible for managing the resources of the entire cluster, a *Node Manager*, which resides on each node of the cluster, is responsible for managing the resources of a node. Figure 2.11 shows high-level architecture of YARN.

Jobs submitted to YARN are a mix of production jobs and best-effort jobs. Both types of these jobs are composed of one or more tasks. Users submit jobs to the resource manager. Once a job is submitted to the resource manager, it creates an *Application Master* corresponding to the job on a node. The application master coordinates the execution of the job. The application master is responsible for managing the execution of the job's tasks throughout its life-cycle. The application master tracks the progress of tasks, handles failures, handles peak resource demands, and applying optimizations. The application master makes resource requests to the resource manager in periodic heartbeats. These heartbeats also inform the resource manager of the liveness of the application master and, consequently, the status of the job it is managing. YARN supports three built-in scheduling policies: fair-share, First-In-First-Out, and capacity scheduling. Depending on resource availability and the selected scheduling algorithm, the resource manager satisfies resource requests in the form of containers. Upon receipt of the containers, the application master schedules tasks to run inside those containers.

2.4.3.6 Spark

Apache Spark[55] is an open-source big-data processing framework. Spark improves on Hadoop's performance for MapReduce jobs by introducing an in-memory system. Spark is one of the most widely used data analytics platforms popular among large organizations such as Yahoo, Tencent, Baidu, etc. Spark is capable of performing complex analysis on large data sets. Spark's in-memory system for MapReduce jobs enables it to achieve upto 100x higher performance than Hadoop. Spark supports Java, Python and Scala through its APIs. Furthermore, Spark supports all file storage systems supported by Hadoop. The key data abstraction introduced by Spark is a Resilient Distributed Dataset (RDD). An RDD is a read-only collection of objects that reside in system memory divided across multiple machines in a Spark cluster. If a partition is lost or faulty, the RDD can be rebuilt. The key advantage of the RDDs is that the objects in a RDD can be accessed without requiring a disk access.

The Apache Spark project consists of several components:

- Spark SQL: Spark SQL allows users to use SQL to manipulate and run queries over data imported from structured relational databases and RDDs;

- Spark Streaming: Spark Streaming is a powerful tool that allows users to perform complex analytics over both historical data stored in RDDs and streaming data;

- MLib: MLib is a scalable machine learning library that provides various optimized algorithms and achieves speeds upto 100x faster than Hadoop; and

- GraphX: GraphX is a library that enables users to perform graph-parallel computations. GraphX extends RDDs to support a Resilient Distributed Property Graph. A Resilient Distributed Property Graph is a DAG with each vertex and edge having associated properties. GraphX contains a growing library of several graph algorithms such as PageRank, Connected components, Label propagation, and triangle counting.

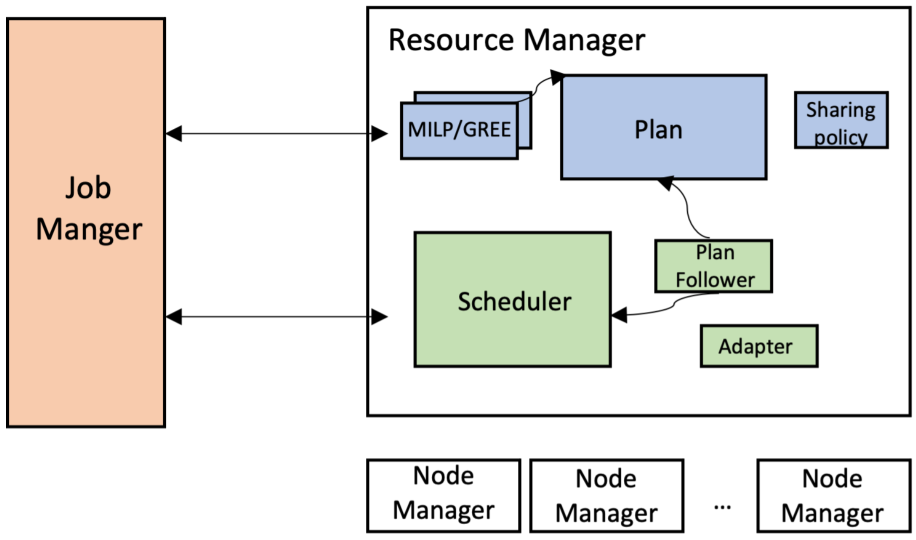


Figure 2.12: The high-level architecture of Rayon [16]

2.4.3.7 Corona

Another cluster manager that uses a two-level scheduling system is Corona [15]. Corona was developed by Facebook to support the high scalability and high fault-tolerance requirements that are important to their business. In some ways, Corona's architecture is similar to YARN's. For example, Corona has a central resource manager that works with numerous Job Trackers to schedule tasks. For each job submitted to the system, a job tracker is made. Job trackers are responsible for monitoring the job's progress and resource consumption. Although Corona and YARN share similarities, there is one key difference. Unlike YARN, Corona uses a push communication approach between nodes and the resource manager to request resources or report the progress of jobs.

2.4.3.8 Rayon

Rayon [16] is a scheduling system that was developed at Microsoft and released as an extension to YARN. Rayon uses a reservation-based approach to scheduling. This simply means that resources are reserved for jobs based on information about the deadline and time-varying resource needs of jobs. One key contribution that Rayon makes is the reservation definition language (RDL). RDL is used to specify constraints, including deadlines, malleable and gang parallelism requirements, and inter-job dependencies. Rayon is not only capable of handling a mixed workload consisting of production jobs and best-effort jobs, but it concentrates on increasing the acceptance rate of production jobs with SLAs and satisfying SLAs of all accepted jobs. Like YARN, Rayon also has a central Resource Manager, per-node node managers, and per-job Job Managers. Also, like YARN, containers used by Rayon supports CPUs and memory as resources. Other customizable resources are not supported. This is because Rayon uses YARN as the underlying framework. Figure 2.12 shows high-level architecture of Rayon.

Rayon's scheduler is responsible for two main parts of the scheduling approach, planning and dynamic allocation of cluster resources. Rayon uses greedy heuristics to plan for jobs with SLAs. This allows also allows Rayon to dynamically allocate resources for both production jobs and best-effort jobs based on the changing cluster conditions. Planning is done in two phases at Rayon: first, to accept or reject jobs on arrival and second to reorganize accepted jobs based on changing cluster conditions.

Users submit jobs to Rayon along with the job specification expressed in Rayon's RDL. Upon the arrival of production jobs, Rayon starts planning and assigns it a start time. Best-effort jobs are scheduled on the unreserved resources by the dynamic and adaptive scheduling once the resources for accepted production jobs are reserved. It is worthwhile noting that Rayon plans one job at a time. Once the placement decision is made for a job, it is never reconsidered. Jobs that arrive at Rayon are composed of one or more tasks that are run in a container. Each container has a specified amount of CPU, memory, and a minimum runtime for the container that depends on the task's execution time estimate. If the cluster is under-reserved and there are resources available on the cluster, Rayon schedules a production job to run as a production job with guaranteed resources up to a point after which the job is run as a best-effort job until completion.

2.5 From Existing Systems to DARSS

In comparison to existing cluster schedulers, DARSS's scheduler is the first task-based scheduler, which assures completion before the deadline for production jobs while reducing latencies for best-effort short jobs.

State-of-the-Art Schedulers YARN [50], Corona [15], Omega [44], and Mesos [25] can be directly compared to DARSS. DARSS can be implemented over any of the systems mentioned above. None of the above-mentioned systems have a temporal deadline aspect to their resource allocation, which differentiates DARSS's scheduler from existing cluster schedulers.

Workflows and Interdependent Jobs Scheduling WOHA [33] introduced deadline-aware workflow scheduling over Hadoop clusters, where tasks are scheduled according to their progress-based priorities (the priorities assigned to the tasks are based on the number of tasks in the workflow that have been scheduled). These priorities are assigned using the resource allocation hints provided by the scheduling plan locally generated at the client node instead of the master node upon job submission. WOHA is implemented as an extension of Hadoop 1.2.1, which makes it specific to MapReduce workflows and is therefore difficult to apply to modern YARN-based frameworks. Furthermore, while WOHA focuses on Workflows or multiple interdependent MapReduce jobs, it does not consider ad-hoc jobs. Microsoft's Rayon [16] does a better job of dealing with workflows that have multiple types of interdependent jobs while also trying to reduce latencies for the ad-hoc jobs. Rayon is implemented as a part of Hadoop 2.6, which makes it compatible with other YARN-based frameworks. However, the level of granularity at which Rayon schedules production jobs still makes some ad-hoc jobs suffer high latencies.

Deadline-Aware Scheduling DARSS builds on the contributions made in the prediction of execution times and deadlines of batch processing frameworks [36, 34] by adding support for multiple types of interdependent jobs. Although deadline-constraint-based schedulers exist, they are Hadoop specific and can only handle deadlines for MapReduce jobs. With the emergence of new frameworks [50, 25, 51, 42, 44] that support multiple types of jobs or a combination of them, schedulers that only support MapReduce jobs have become outdated [16].

Reservation-based Scheduling Existing big-data systems [50, 25, 21, 51, 15] focus on maximizing cluster throughput. Their resource-sharing policies are based on priority, fairness, and capacity. On the one hand, prioritizing production workflows helps the chances of completing them before their deadlines, while increasing the latency of best-effort short jobs. On the other hand, prioritizing best-effort short jobs improves their latency, while decreasing the chances of production workflows completion before their deadlines. In both cases, we encounter head-of-line blocking, which prevents these systems from fulfilling the requirements of both types of jobs. Especially, no guarantees can be made on job allocation over time.

Chapter 3
SYSTEM DESIGN AND IMPLEMENTATION

This chapter presents the design and implementation of the Deadline-Aware Reservation-based Scheduling System (DARSS). Section 3.1 presents architectural details which consider recent advances in cluster computing and limitations of the existing big-data processing frameworks. Section 3.2 discusses the implementation of DARSS and how the architecture translates to the implementation.

3.1 DARSS Architecture

DARSS's design addresses the shortcomings (Section 2.2) of the existing big-data systems. These shortcomings point to the need for a framework that can handle complex workloads, support production jobs' SLAs, lower best-effort job latency, and increase cluster throughput. When considered individually, there were several choices for improvement for each of these aspects. For example, to support the completion of a high number of production jobs before their deadline, production jobs could be prioritized. However, these issues are connected. Therefore, improving one imposes limitations on the possible choices for the other aspects.

The design concentrates on deadline-aware and reservation-based scheduling to solve these issues. Figure 3.1 shows the four stages through which DARSS handles job scheduling:

- Configuration, in which the client submits the job request (production job or best-effort job);
- Analysis, where the system calculates resources and analyzes the workflow requirements;
- Reservation, based on the analysis, where the system either confirms reservation or declines request; and
- Scheduling, where the system schedules individual tasks using the time-to-deadline list.

The following two subsections discuss deadline-aware and reservation-based scheduling, respectively. The subsequent four sections explain the four stages of the system design.

3.1.1 Deadline-Aware Scheduling

DARSS, like several existing deadline-aware schedulers, takes advantage of the existing approaches of prediction of execution times of jobs submitted and executed by batch processing frameworks. However, DARSS also supports multi-framework interdependent jobs.

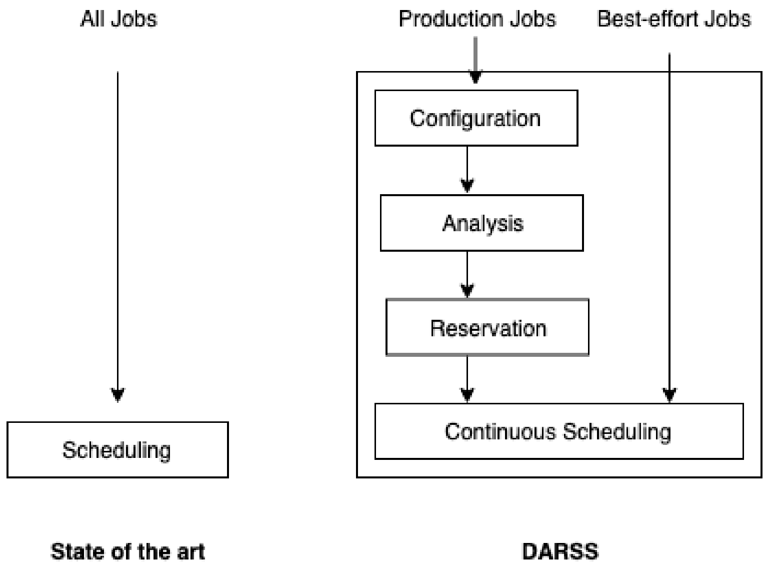


Figure 3.1: Overview of DARSS design

3.1.2 Reservation-Based Scheduling

DARSS's design is influenced by Microsoft's Rayon, one of the first scalable systems to take the reservation-based scheduling approach.

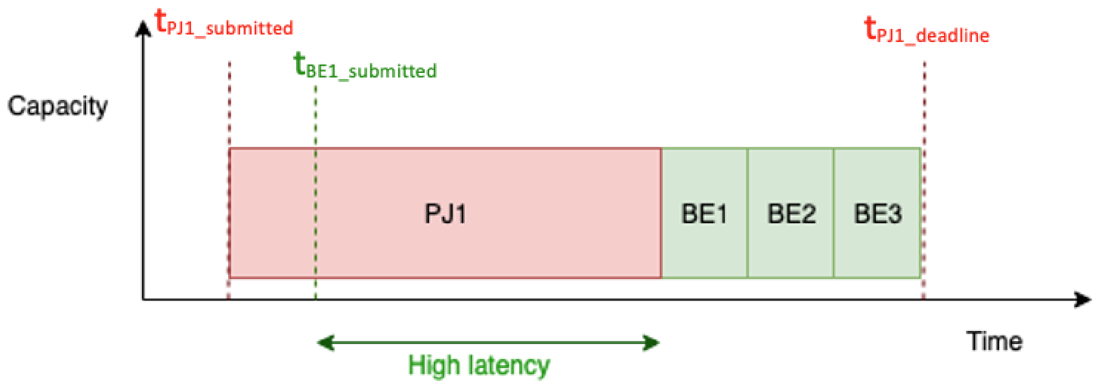


Figure 3.2: The high latency encountered by best-effort jobs

Figure 3.2 illustrates the high latency encountered by best-effort jobs. Prioritizing production jobs can have a big head-of-line blocking impact on the jobs queue. In this figure, PJ1 is a production job submitted at time $t_{PJ1\_submitted}$. Since there are no other jobs in the queue, the scheduler assigns resources to PJ1. BE1, a best-effort short job, is submitted at time $t_{BE1\_submitted}$. Due to head-of-line blocking caused by scheduling PJ1 first, BE1 cannot start until PJ1 finishes. This causes high latency for all the best-effort jobs that are submitted before $t_{PJ1\_deadline}$ and after PJ1 is submitted.

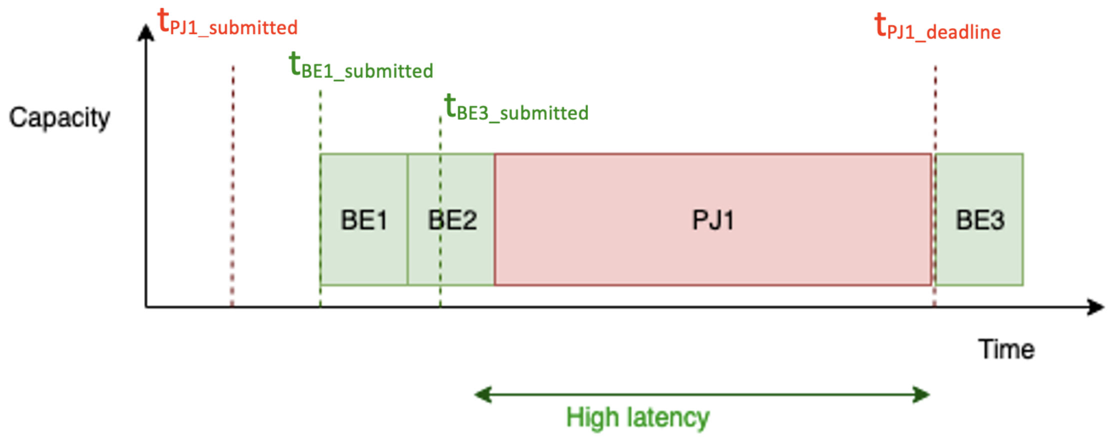


Figure 3.3: Reduced latency by scheduling production jobs close to their deadline

Figure 3.3 shows Rayon's approach to solving this problem. Rayon's approach is to meet PJ1's deadline and improve BE1 and BE2's latencies by delaying PJ1. By the introduction of reservation-based scheduling, the system is armed with significantly more information to handle jobs with deadlines. The additional information, when combined with effective continuous scheduling delivers an advantage over other existing approaches. Although this approach reduces the latencies for best-effort jobs significantly, the delaying of scheduling PJ1 can lead to unutilized resources.

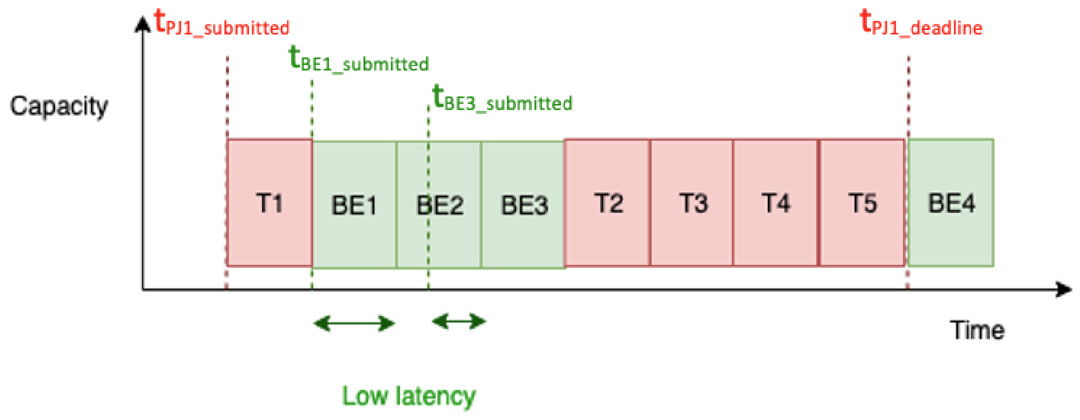


Figure 3.4: Reduced latency for best-effort jobs by introducing fluidity

To address this problem, DARSS introduces a more fluid scheduling approach. In the example shown in Figure 3.2, PJ1 is a production job (or a Workflow of Jobs). As mentioned in section 2.2.2, production jobs are made up of smaller jobs. These smaller jobs are made up of even smaller tasks. DARSS breaks up the production jobs into tasks and schedules tasks independently. These tasks are scheduled as close to the deadline as possible. This introduces more fluidity than Rayon by delaying the scheduling of the tasks instead of workflows. Due to this fluidity, more best-effort jobs can get scheduled, and a relatively smaller number of

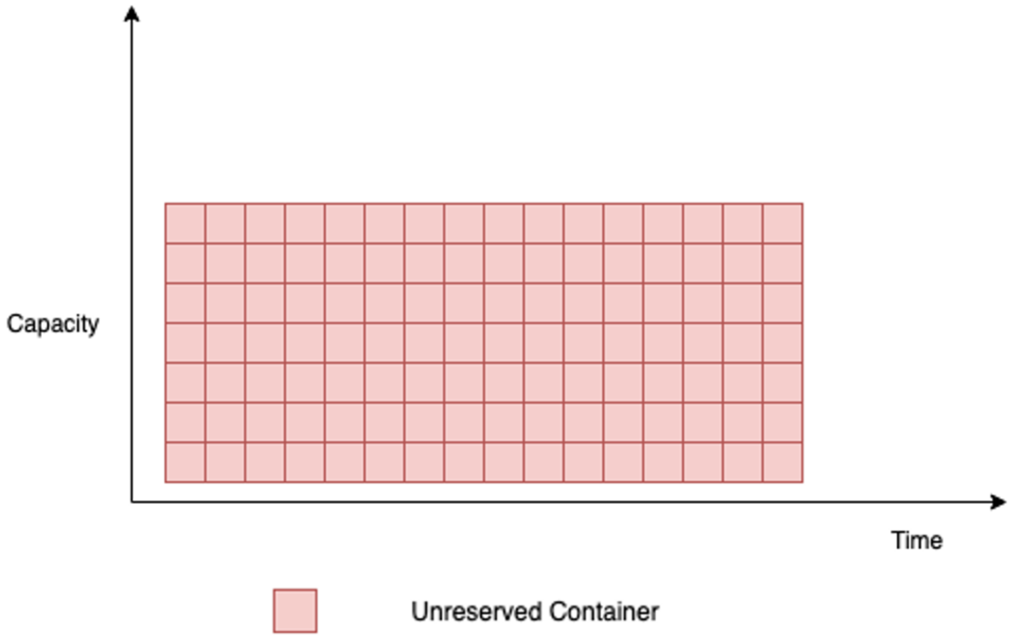


Figure 3.5: System resources divided into Atomic Resource Units

best-effort jobs have to face relatively low latency. In Figure 3.4, Production Job PJ1 is broken into Tasks T1, T2, T3, T4 and T5. DARSS analyzes the interdependence among the tasks and derives individual deadlines for each task. Equipped with individual task deadlines along with the estimated time-to-completion, DARSS puts the tasks on a Task List. If no best-effort job is available, a task from the list is scheduled. In the figure, there are no best-effort jobs submitted at time $t_{PJ1\_submitted}$. So, Task1 gets scheduled. At the completion of T1, BE1 gets submitted to DARSS. Since T2 can still finish without compromising the completion of PJ1 as a whole, BE1 gets scheduled to reduce the latency which best-effort jobs often face after a production job has started. Figure 3.4 shows how the delayed individual scheduling of tasks reduces latency for best-effort jobs BE1, BE2, and BE3.

To support maximum fluidity, the resources of the cluster are divided into atomic resource units. These units are YARN containers that have 1 core and 1 GB of RAM with a minimum lease duration of 1 second. With this design, DARSS achieves maximum fluidity. If a task requires 2 cores and 2 GB of RAM for 2 seconds, the container assigned to the task is equivalent to 4 atomic resource units. Figure 3.5 shows a cluster that can run 7 containers in parallel.

3.1.3 DARSS's Stages

Jobs submitted to DARSS go through four stages: Configuration, Analysis, Reservation, and Scheduling. Each stage is described in the next four subsections.

3.1.3.1 Configuration

In the configuration stage, a job (production job or best-effort job) is submitted to DARSS. A job is submitted with a configuration file that contains all the necessary information for the analysis stage. The information is formulated through DARSS's requirement specification language.

Figure 3.6 illustrates a workflow with interdependent jobs that have interdependent tasks. The figure shows a workflow with jobs 1, 2, and 3. Job 2 cannot start before the completion of Job 1. Job 3 is independent and can start irrespective of when either of the other jobs started. Job 1 shows partial inter-dependency where tasks 2 and 4 both are dependent on task 1's completion, whereas task 3 is independent of any other task. Job 2 shows no inter-dependency. All the tasks can start independently of one another. Job 3 shows complete inter-dependency where task 2 cannot start before completion of task 1, and task 3 cannot start before the completion of task 2. All of this information is encoded in a configuration file using the Requirement Specification Language.

Requirement Specification Language The configuration file contains the information that can be expressed as follows:

$$\begin{aligned}
W^i &= \{D^i, G^i\} \\
G^i &= \{J^i, P^i\} \\
J^i &= \{J^i_1, J^i_2, \ldots, J^i_n\} \\
P^i &= \{P^i_1, P^i_2, \ldots, P^i_n\} \\
J^i_k &= \{T^i_k, Q^i_k, L^i_k, M^i_k\} \\
T^i_k &= \{T^i_{k,1}, T^i_{k,2}, \ldots, T^i_{k,m}\} \\
L^i_k &= \{L^i_{k,1}, L^i_{k,2}, \ldots, L^i_{k,m}\} \\
Q^i_k &= \{Q^i_{k,1}, Q^i_{k,2}, \ldots, Q^i_{k,m}\} \\
M^i_k &= \{M^i_{k,1}, M^i_{k,2}, \ldots, M^i_{k,m}\}
\end{aligned}$$

where,

W = User-defined workflow

D = Set of deadlines for each job in workflow i

G = DAG of jobs

i = Workflow identifier

J^i = Set of Jobs in workflow i

P^i = Family (power set) of sets of prerequisite jobs of each job in J^i

$$
\begin{aligned}
P^i_j &= \text{Set of prerequisite jobs of } J^i_j \text{ ,if } J^i_k \in P^i_j, J^i_j \text{ cannot start before } J^i_k \text{ finishes}\\
T^i_k &= \text{set of tasks in job } J^i_k\\
Q^i_k &= \text{family (power set) of sets of prerequisite tasks of each task in } Tik\\
Q^i_{k,l} &= \text{set of prerequisite tasks of } T^i_{k,l}\\
L^i_k &= \text{set of number of time quanta each task needs to lease}\\
M^i_k &= \text{set of units of resource allocation}
\end{aligned}
$$

The configuration file for a submitted contains a user-defined workflow W^i. This workflow contains a Directed Acyclic Graph of job (smaller jobs that constitute a workflow or a production job) $G^i$and a set of deadlines D^i that has a deadline for each job in the workflow. A set of deadlines is used in DARSS instead of one deadline for the whole workflow. This is because in production workflows, there are often different deadlines for different jobs. In the case where the individual jobs do not have deadlines but the workflow as a whole has a deadline, deadline for all the jobs is the deadline for the workflow.

The DAG of jobs contains a set of all the jobs in the DAG J^i and a power set P^i containing sets of prerequisite jobs for each job in J^i. The inclusion of prerequisites jobs is to express inter-dependencies between jobs. As shown in Figure 3.6, the production jobs or workflows consist of smaller jobs that can be (and in most cases, they are) interdependent.

Each job J^i_k in a set of jobs J^i is represented by a quadruple $\{T^i_k, Q^i_k, L^i_k, M^i_k\}$, where T^i_k is a set of Tasks, Q^i_k is a power set containing sets of prerequisite tasks for each task in T^i_k, L^i_k is a set of the number of time quanta each task needs to lease for completion, and M^i_k is a set of resource allocation units. The inclusion of prerequisites tasks is to express inter-dependencies between tasks. As shown in Figure 3.6, the jobs in workflows consist of smaller tasks that that can be interdependent. This means one task cannot start before another completes.

The allocation unit M^i_k, j represents processors and RAM. This is similar to a YARN container. For example, M^i_k, j = {1 Core, 1GB Ram}

3.1.3.2 Analysis

RSL represents the needs of all the jobs. DARSS receives the reservation requests along with the job submission. The information provided with the RSL is used to perform online job selection. This means that only the jobs which are accepted are the ones with deadlines that can be satisfied by the system and, all the rest are rejected.

The system maintains a reservation file that logs the number of unit containers reserved for a workflow until some time T. Each time a workflow request is submitted, the system uses the reservation file to check the total number of unallocated and unreserved containers from the time of submission to the deadline for

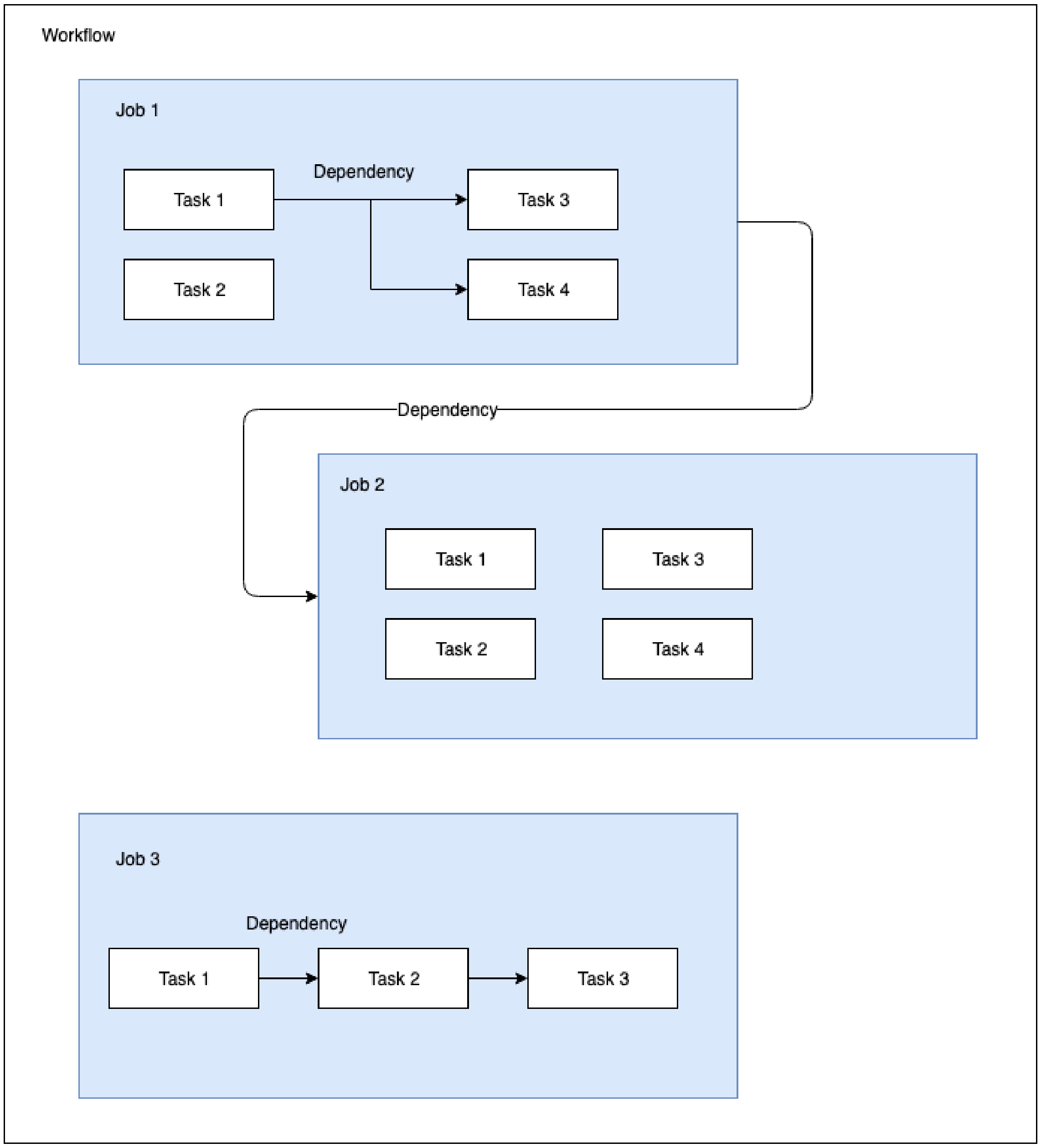


Figure 3.6: A workflow with interdependent jobs and tasks

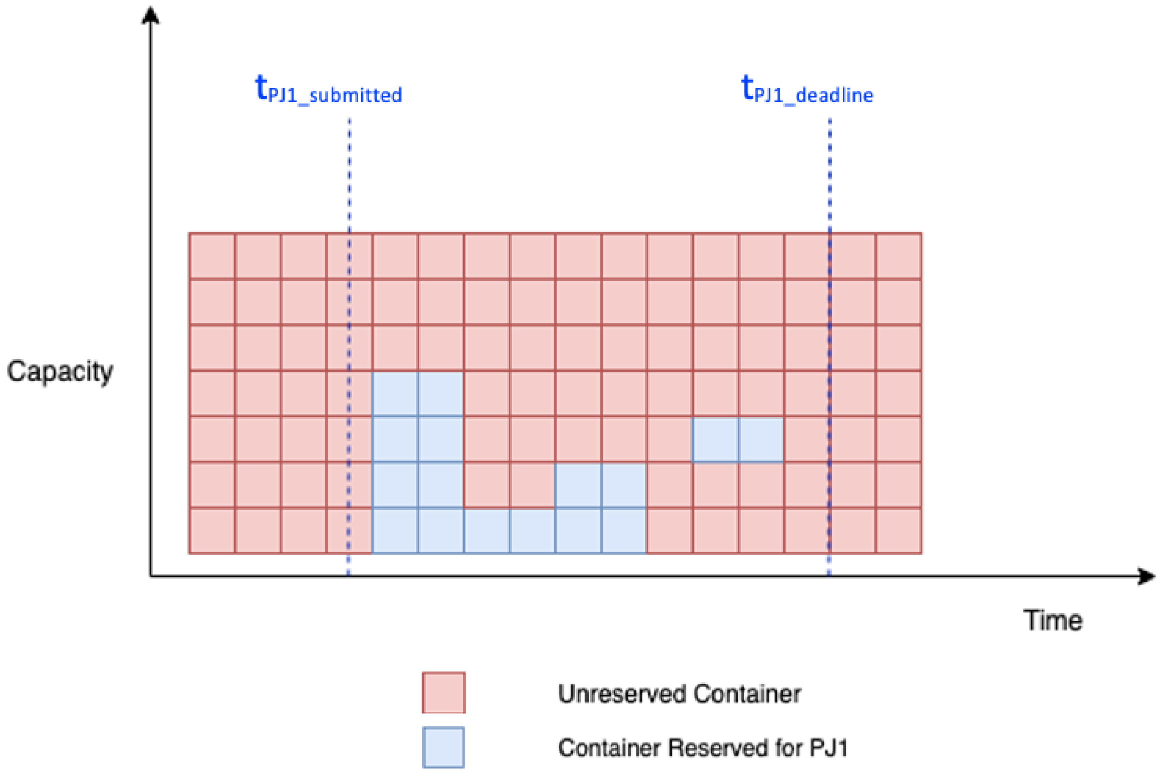


Figure 3.7: DARSS calculating the total free containers until deadline

the workflow. The system also calculates the number of unit containers required to complete the workflow. This can be known from the number of cores required by the workflow defined in the submitted configuration file.

Figure 3.7 illustrates the process of calculating free containers ahead of time. From the time t1 when a production job PJ1 is submitted till time t2, the deadline of PJ1, there are 70 total containers in this example. Out of those, 16 containers have been reserved for another job that had been accepted previously. This means there 54 containers unreserved. DARSS calculates this number in the analysis stage.

3.1.3.3 Reservation

In this stage, DARSS either accepts a job request or rejects it based on the output of the analysis stage. One of the primary objectives of DARSS is to reduce latency for best-effort jobs. To make sure that the production jobs or workflow do not starve best-effort jobs, DARSS introduces a periodic best-effort job quota. This allows some containers to be reserved for best-effort jobs over a period of time. More specifically, m containers per n total containers are reserved for best-effort jobs over a period of l time units. The values of m,n, and l are dynamic and are set by cluster administrators depending on production job to best-effort job submission ratio. Cluster traces from Facebook [13], Google [48] and Yahoo [14] show that more than 80%

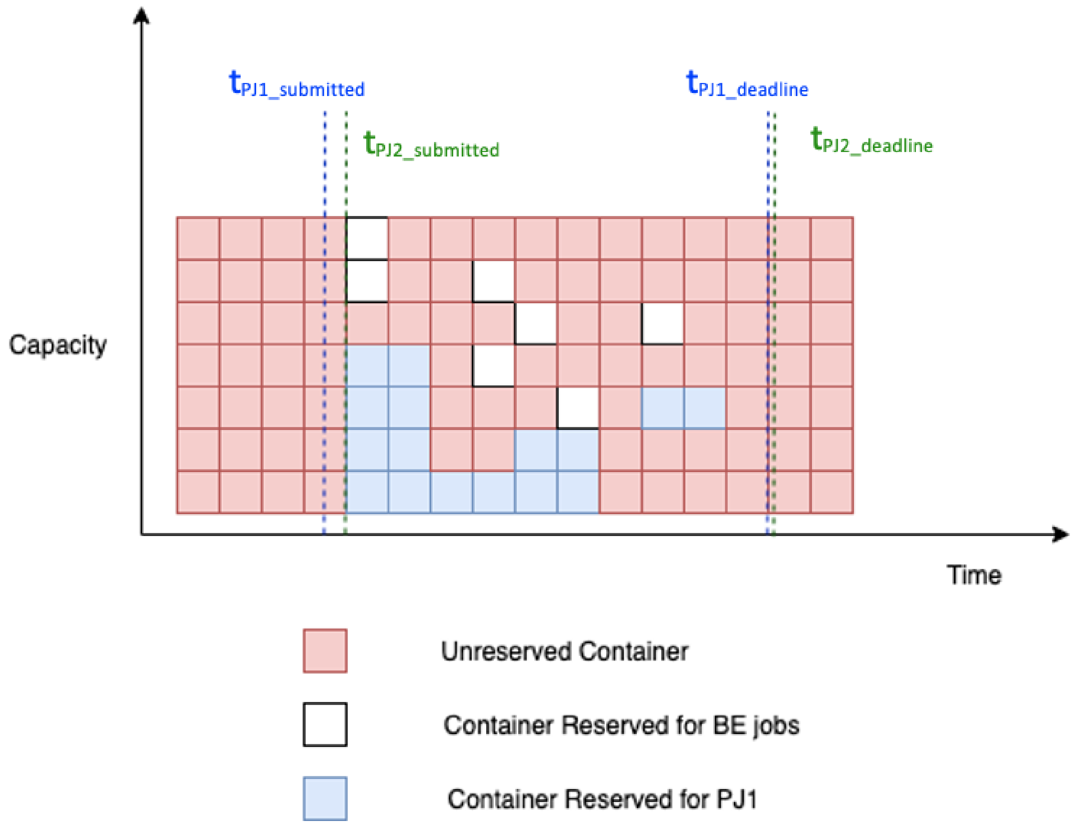


Figure 3.8: Acceptance of a production job based on unallocated containers

of the workloads consist of best-effort short jobs. Best effort-jobs can be as much as 95% of all jobs. This, however, does not mean that prioritizing best-effort jobs would increase throughput, since production jobs consume far more resources.

The total number of reserved containers until time t is the number of containers reserved for production jobs and the number of containers reserved for best-effort jobs. If the total number of containers required by a job request with the deadline at time t is greater than the total number of unreserved containers, the job request is rejected. Otherwise, the job request gets accepted.

Figure 3.8 shows a scenario where there are two workflows (PJ1 and PJ2) requests made with both workflows having the same deadline at 10 time units from the time of submission. PJ1 was submitted and accepted before the submission of the PJ2 request. Assume, m=7 containers, n=70 containers, and l = 10 time units and a task requires 1 container for 1 time unit to complete. This means that there are 47 containers that are unreserved until $t_{PJ2\_deadline}$. If the number of containers required by PJ2 is less than or equal to 47, PJ2 will be accepted. Otherwise, it will be rejected.

If a job request gets accepted, the system updates the reservation file showing the number of containers reserved for that job. It is important to note that DARSS does not reserve specific containers but the number of containers required by the job. This allows maximum fluidity and flexibility in scheduling.

3.1.3.4 Scheduling

During the Scheduling stage, DARSS rearranges and schedules individual tasks that belong to jobs accepted as a result of the Analysis stage. The system schedules individual tasks with the help of a ttd (time-to-deadline) list.

Using the configuration provided by the client, the system derives a list of ttd (time-to-deadline for each task in the workflow). This list contains tasks from best-effort jobs as well as from production jobs and their respective individual deadlines. The deadlines are calculated using the time that the task needs to lease the allocation unit, the deadline of the job or the workflow, and the dependencies each task has on other tasks. Since best-effort jobs do not have deadlines, the ttd of their tasks is based on time of submission and values of m,n, and l (from section 1.1.5). Best-effort job tasks are scheduled as soon as possible (given that there are containers available that were reserved for best-effort jobs). This guarantees that m number of best-effort job tasks will get executed over a period of l time units (assuming 1 task takes 1 time unit to complete using 1 container). The list has two parameters: task identifier and ttd (time-to-deadline). Each time a new job request is accepted, the ttd list is revised. Whenever a container finishes the previously allocated task, the task at the top of the list is scheduled for that container.

The overall scheduling heuristic delays scheduling a task as close to its deadline as possible. This approach improves the chances of jobs getting submitted late and still getting accepted (and consequently completing before their deadline). The Admission Control and the Reservation stage provides a guaranteed allocation of resources to jobs to complete before their deadlines, whereas the Scheduling stage provides jobs an opportunity to utilize more resources (if there are unutilized resources) than guaranteed and thus completing even earlier than the expected completion.

Consider the workflow in Figure 3.6. For simplicity, assume the workflow is submitted at time t, the capacity of the cluster is 2 (number of containers that can run in parallel), each task takes 1 container for 1 time unit to execute, $m = 6$, $n = 20$, $l = 10$ and the workflow was accepted. Table 3.1 shows the TTD list derived from the workflow. The task identifier w1j1t1 means the first task of the first job of the first workflow. Each task in the workflow has its own deadline corresponding to the identifier.

Figure 3.6 shows that in the workflow, job3 is independent of the other two jobs. Within job3, tasks are dependent on the previous one. Therefore, the ttd for task3 of job 3 is the same as the deadline for the whole workflow, i.e., 10. Since each task takes 1 time unit to complete, task 2 of job 3 has the ttd of 9. Similarly, task1 of job3 has ttd of 8.

The workflow also has two other jobs, one of which is dependent on the other. This means that job2 cannot start before the completion of the last task of job1. Within job2, none of the tasks are dependent on the other. Therefore, the ttd for all the tasks in job 2 is 10. Tasks within job1, however, have dependencies amongst them. Since tasks 3 and 4 cannot start before completing task 1, the ttd for tasks 3 and 4 of job 1

Table 3.1: Workflow Tasks scheduled close to deadline

TTD List	
Task Identifier	**Time-To-Deadline**
w1j1t1	8
w1j3t1	8
w1j3t2	9
w1j1t2	9
w1j1t3	9
w1j1t4	9
w1j3t3	10
w1j2t1	10
w1j2t2	10
w1j2t3	10
w1j2t4	10

is 9. Consequently, the ttd for task 1 is 8. Task 2 of job1 can run independently, so it has a ttd of 9.

3.2 DARSS Implementation

The section discusses the overview of the system architecture, steps involved in running the jobs, and the details of DARSS implementation.

3.2.1 System Overview

This subsection lists the main components of the system architecture and discusses the steps involved in running workflows and best-effort jobs. Figure 3.9 shows an overview of system architecture. The DARSS architecture contains three main components:

- A Central Resource Manager controls the allocation of physical resources to tasks;
- Per-Node Node Managers that run on each worker node and are responsible for controlling access to resources at the node level; and
- Per-job Application Managers that are responsible for negotiating resources with the resource manager.

Next discussed are the steps involved in running a job in DARSS.

Step 1. The system receives a job request along with the configuration file expressed in RSL. The application manager uses the requirements encoded in RSL to communicate the needs and specifications of the job to the central resource manager. (Configuration stage)

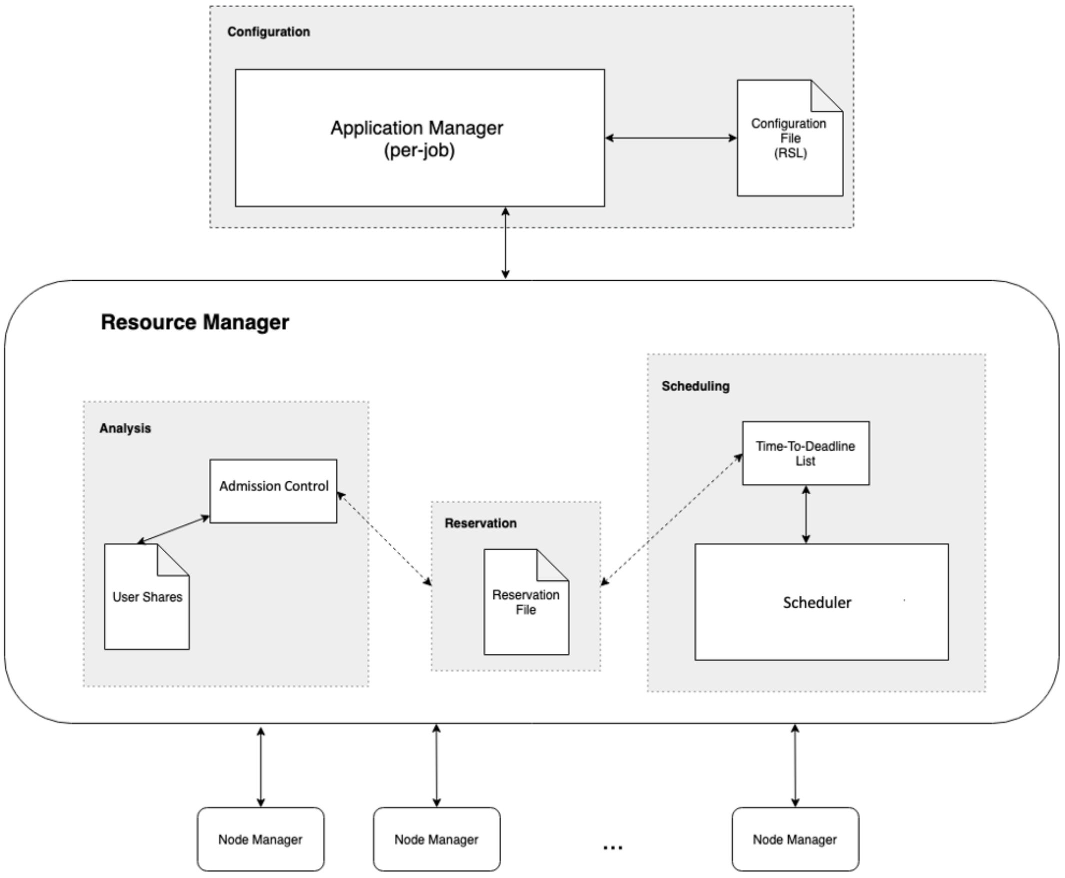


Figure 3.9: DARSS Architecture Overview

Step 2. The reservation file in the resource manager maintains a view of the cluster resource allocations by logging all the reservations made by the system. This step also is responsible for enforcing the user shares or user quotas. (Analysis stage)

Step 3. The allocation that results from the previous step is validated against the physical resources. (Analysis stage)

Step 4. The user is informed of acceptance or rejection of a job request. In case of acceptance, the reservation file is updated, and the user receives a reservation identifier. The reservation can be tracked in the reservation file through this identifier. If the job request is rejected, the user can relax some of the job's temporal constraints and resubmit the job request. (Reservation stage)

Step 5. The system maintains a ttd list that distributes the deadline of a workflow to its jobs and tasks. The ttd list is used to transfer the reservation information from the reservation file to the schedulers. (Reservation stage)

Step 6. The scheduler uses the ttd to assign resources to the tasks. The central resource manager assigns tasks to node managers in order to service the accepted jobs. (Scheduling stage)

Step 7. The system dynamically updates the ttd list and the reservation file each time a new job is accepted.

3.2.2 YARN

Although DARSS's architecture is compatible with most of the recent big-data systems, DARSS's implementation is based on Apache Spark's YARN-based scheduler, because of its popularity and it being an open-source project. Apache Spark is a big-data framework most widely used with YARN. Like Apache Spark, DARSS is implemented using Scala. Another reason for selecting Apache Spark is the variety of available Spark libraries (GraphX, MLib, Spark Streaming, etc.) that support different types of jobs. This makes it possible to evaluate DARSS with a wide variety of jobs in the workload.

As discussed in Section 2.4.10, YARN has a central Resource Manager for the whole cluster, Node Managers for each node in the cluster, and Application Masters that can be viewed as a job manager as it coordinates and controls the execution of tasks in a job on the cluster. The Resource Manager is a rack-aware master node that manages the global assignments of resources among all the applications. Most importantly, it provides two schedulers. Fair Scheduler and Capacity Scheduler. DARSS leverages both the schedulers to implement its own continuous scheduler.

Fair Scheduler The Fair Scheduler schedules resources such that, over time, all the applications get an equal share of the total resources on average. If there is one job in the queue, all the resources get assigned

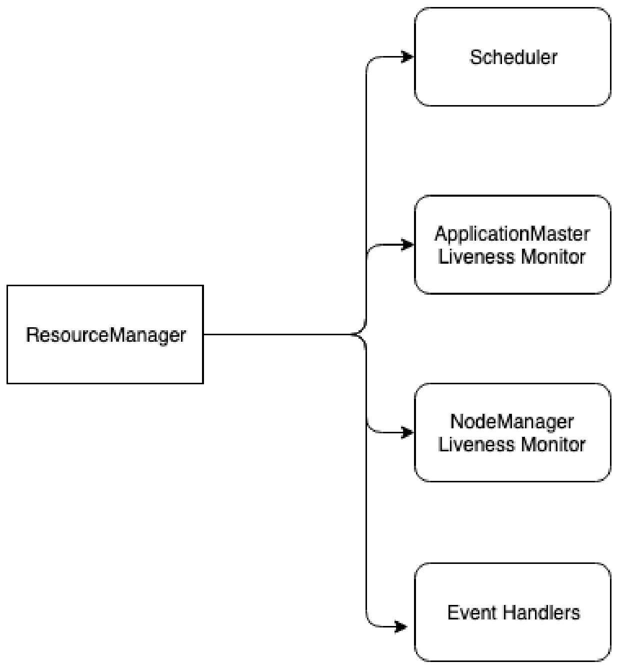

ResourceManager
Scheduler
ApplicationMaster Liveness Monitor
NodeManager Liveness Monitor
Event Handlers

Figure 3.10: YARN's Resource Manager

to that job. As more jobs come, the resources are divided amongst them "fairly." In addition to this, it also allows to provide guaranteed minimum shares of resources to queues. This provides a way to partition resources such that certain groups or users get sufficient resources. In general, Fair scheduler is more flexible of the two schedulers as it allows applications to utilize unused resources.

Capacity Scheduler The Capacity Scheduler is based on a cluster sharing idea. It is designed for a shared, multi-tenant cluster to maximize the cluster throughput of the cluster.

Organizations have their own private clusters that have sufficient resource capacity to fulfil their computational needs under peak conditions. This often leads to poor average resource utilization. Alternatively, sharing clusters between organizations can help organizations to enjoy the benefits of cluster computing without having to invest in costly private clusters. However, in a shared cluster, organizations have concerns about the resource capacity availability for their production jobs with SLAs. The Capacity Scheduler addresses this concern by providing each organization, sharing the cluster, a minimum resource capacity guarantee. The available resources in a shared cluster are divided among multiple organizations who pay their share of the cost depending on their computational needs. This approach also allows the unutilized resources to be used by an organization.

The Capacity Scheduler provides a strict set of rules to ensure that one organization or job does not use a unbalanced amount of resources in the cluster. The Capacity Scheduler uses *queues* to enforce these resource capacities. These queues are setup by cluster administrators and represent the division of resources in the shared cluster

Table 3.2: API Extension to Admission Control Protocol

DARSS's API Extension		
API	**Input Variables**	**Return Values**
submitJobRequest	configurationFile cf1	boolean, reservationID
deleteJobRequest	reservationID r1	boolean
listAllRequests	OrgID o1	List <reservationID>

3.2.3 Request Submission

YARN's application submission protocol has been modified to support the configuration, analysis, and reservation stage. DARSS has the following APIs added to the SPARK code-base.

These APIs allow users to submit job requests before execution and receive acceptance or rejection information.

- The "submitJobRequest" API lets users submit a job request with a configuration file that includes workflow specification expressed in RSL. The API call returns a Boolean value indicating acceptance or rejection and a reservation identifier. In the case of success, the reservationID contains the identifier for the reservation. In the case of rejection, the value of reservationID is NULL.

- The "deleteJobRequest" API lets users delete their reservation before any task of their job has started. The API call returns a Boolean indicating success or failure. If the system had not started the execution of the job, the API call returns success. Otherwise, the API call returns failure.

- The "listAllRequests" API lets a user list all the active accepted job requests that a user has submitted. The API call returns a list of reservationIDs.

DARSS significantly extends YARN's ResourceManager and SPARK's metrics.properties file to support the APIs introduced above. DARSS introduces the analysis layer, reservation layer, and continuous scheduling layer to YARN's architecture. This includes the reservation file showing all the resource commitments made by the system and accepted jobs, the modified scheduler and the time-to-deadline list that is used by the scheduler.

3.2.4 User Share

DARSS allows cluster administrators to control the cluster capacity assigned to different user groups. This is achieved through user shares or user quotas. In the case where there are no defined user shares, DARSS allows groups to request arbitrary resource allocations. These requests are only bound by the cluster's physical resource constraints. DARSS extends YARN's capacity implementation to enforce constraints over resources.

Algorithm 1: Scheduling a task in DARSS

```
Input: Containers cr, ttdList tdl, TimeInterval ti
Output: A task id that has been scheduled
foreach Container c in cr do
    foreach Time t in ti do
        if c.isBusy() = true then
            skip
        else
            tsk = tdl.peek()
            tsk.assign(c)
            tdl.dequeue()
            return tsk.id
```

The cluster administrators can restrict the amount of resources requested by a group through these user share policies. These restrictions can be for resources at a particular time (jobs can have varying peak resource utilization times) or over a period of time. User share policies can be changed to enforce any policy from completely unrestricted resource allocation to restricted resource allocation where user share of the resources in the cluster are capped by a certain percentage. For example, a user share policy could dictate that no user can utilize more than 10% of cluster capacity or that no user will be allocated more than 5% of cluster capacity on average over a period of 72 hours.

3.2.5 Time-to-deadline

The Analysis stage and Scheduling stage track the resources. Scheduling stage has a myopic view of containers and time units. It is responsible for allocating resources (containers) to tasks for the immediate time unit. The Analysis stage, however, views resources for the whole cluster over the time continuum. The commitments made to the users by accepting their job requests have to be translated to the scheduling queues. In other words, the view of the analysis stage has to be translated into the view of the scheduling stage. DARSS achieves this through a time-to-deadline (ttd) list. The list has two parameters: task identifier and ttd (time-to-deadline). This list is a priority queue, with the lowest ttd value being the most prioritized task. The ttd list is updated periodically and triggers the scheduler queues to be updated according to the ttd list. DARSS modifies the Capacity Scheduler and Fair Scheduler, allowing the incorporation of ttd list into scheduler queues.

Chapter 4

EVALUATION

This chapter presents the evaluation of DARSS. Among the systems presented in Chapter 2, YARN, Mesos, Spark, and Rayon have open-source implementations. Among them, only Rayon is deadline-aware, which makes it the most suitable system to evaluate DARSS against. Another similarity between DARSS and Rayon is that both systems are built on top of YARN.

DARSS is also evaluated against the stock YARN Capacity Scheduler. This scheduler was picked because of its popularity in production environments and because DARSS is built on top of YARN. The difference in performance between DARSS and YARN can be expected to be similar to the performance difference between DARSS and Mesos. There are two reasons for this: the similarity between YARN and Mesos' schedulers and the deadline agnostic designs of YARN and Mesos. Due to the similarity between YARN and Mesos, DARSS is evaluated against YARN. The results for Mesos can be expected to be similar.

4.1 Simulation Modeling

Simulation modeling is used to test the efficiency of the system in different real-world scenarios. This method permits different sets of configurations for different scenarios and decide what changes can make the system more efficient in solving the problem it intends to solve. AnyLogic PLE was the simulation tool used in this research.

The model is a hybrid model (a discrete event model with custom agents flowing across the pipeline). To understand the rationale for choosing this combination of modeling approaches, one should be familiar with the basic domain knowledge. The Workflows, Jobs, Tasks, and Containers have properties and behaviors associated with them. Therefore it only makes sense that they are represented as agents. Since the Workflows (which are actually jobs, which in turn are interdependent tasks) are submitted to a big data processing server and go through a variety of processes before they can be processed into presentable information, it would only make sense that this series of processes performed on workflows be represented as a pipeline, this can be done by discrete event modeling.

4.2 Experimental Evaluation

This section details our approach in evaluating DARSS, our experimental setup, and the workload used in our experimental evaluation.

4.2.1 Workload

The experimental workload used in the experimental evaluation is based on a cluster trace from a Borg cluster. The following section presents the details of the trace.

4.2.1.1 Google Trace

Google released a Borg cluster trace in October 2011 that consists of 29 days of activity in May 2011 on a cluster of 12,500 machines [40]. The cluster was made up of several individual machines that were connected using high-bandwidth connections. Clusters are also called compute cells. A cell is a set of machines, contained in a single cluster, that share a common cluster management system that is responsible for allocating work to machines. Work arrives at a cell in the form of jobs.[1]

A job consists of one or more tasks. Each task is accompanied by a set of resource requirements used for scheduling the tasks onto machines. These jobs are also accompanied by temporal requirements, which are represented as *priorities*. Priorities in this trace vary from 0 to 11. The *importance* is interpreted in the ascending order of the priority number.[2] This means that 0 and 1 are *free* priority or best-effort jobs whereas 9,10 and 11 are *production* priority or production jobs. The trace consists of a mix of best-effort jobs and production workflows. In total, there are more than 668,000 jobs in the trace that are submitted by 679 users over the course of 29 days.

As mentioned in Chapter 1, cluster computing environments have much higher diversity than Grid computing environments. The cluster from which the Google cluster trace was published has a high diversity in different aspects. The nodes that constitute this cluster are from a variety of machine classes. These machines vary from each other in the amount of processing power, memory, storage, architecture, and more. Table 4.1 shows the resource configuration of the cluster. As mentioned above, the mix of jobs submitted has high diversity. Job duration range from a few seconds to more than a month. Not only do the jobs vary in duration but also in resource requirements. Some jobs require minimal resources, whereas others can require more than 70% of all resources.[3] Table 4.3 shows the task distribution from the trace.

The Google cluster trace contains some obfuscated information due to confidentiality reasons. This includes random hashing of free-text data and linear scaling of resource sizes. The data, however, is still useful for research purposes due to the technique and consistency in obfuscation [40]. Although this trace

[1]These jobs are the same as discussed in chapters 1, 2 and 3
[2]bigger number means more importance
[3]production workflows

Table 4.1: Resource configuration of Google cluster

Google's Cluster			
Number of machines	CPU	Memory	Platform
795	1.00	1.00	C
3	1.00	0.50	C
5	0.50	0.97	B
1001	0.50	0.75	B
6732	0.50	0.50	B
3863	0.50	0.25	B
52	0.50	0.12	B
1	0.50	0.06	B
5	0.50	0.03	B
126	0.25	0.25	A

Table 4.2: Estimations of resource specification from Google cluster trace

Obfuscated Data	Estimation
CPU x 1.00	12 processing cores
Memory x 1.00	16 Gigabytes
Platform A	Intel Itanium processor series
Platform B	Intel Xenon processor series
Platform C	AMD Opteron processor series

Table 4.3: Task distribution in Google cluster trace

	Max	Min	Average
Number of Tasks per job	5000	1	36.56
Task duration (seconds)	1 x 10^6	N/A	3000
Number of cores per task	0.00	0.50	0.0338
Amount of memory per task (GB)	0.955	0.95 x 10^{-6}	0.0284

does not include accurate information regarding the purposes of jobs, the distribution of jobs, machines, and user characteristics can still be estimated.

Some of the cluster features were estimated from this trace by researching the hardware technology in wide use from 2010 to 2011. This estimation is based on the assumption that the hardware that constitutes the cluster was not older than two years. Table 4.2 shows the estimation of resource specification from the trace. The highest number of processing cores on a single machine in 2010 to 2011 was 12.[4] This is estimated to be the "platform C," as mentioned in the trace. "Platform B," as mentioned in the trace, has half the processing cores compared to "Platform C." This would mean that Platform B would have around 6 cores. The most popular 6 core server machines in 2010 were Intel Xenon series of processors. The Xenon chips could support upto 16 Gigabytes of memory in 2010. This means that the max memory of any machine in the cluster had 16 GB of RAM.

Deadlines The Google cluster trace, like most traces from state-of-the-art cluster frameworks, does not provide information about deadlines associated with jobs. This research tackles this problem as in [20]. Based on previous research [16], the ratio of jobs with deadlines to best-effort jobs is 5% of total jobs with deadlines. 10% over-estimation is also assigned over the actual resource requirements of the job. This is possible due to the knowledge of actual resource requirements from the trace. Deadlines are derived from the available information from the trace (resource requirements, time of completion, etc.) This is not ideal, but for the purposes of this research, it is sufficient.

The Google trace was chosen as the experimental workload because of its completeness and comprehensive details regarding the content of the trace. The Google trace contains comprehensive information regarding the configuration of nodes within the cluster, resource requirements, and duration for individual jobs and tasks. Other existing traces do not have either the information about resource configuration within the cluster or resource requirements of individual tasks. Although this trace is about eight years old, it is still one of the few available complete cluster traces. Google's cloud engineering team still supports and maintains it [52].

2019 Trace In April 2020, Google released a new version of the trace [48]. The new trace contains data collected from eight different Borg cells during the entire month of May, 2019. The complete trace is about 2.4 TBs compressed. Google has only made the trace data available through Google's BigQuery multi-cloud data warehouse [53]. The new trace shows that the number of machines per cell remains roughly the same as in 2011 trace [48]. There are several differences between the 2011 and 2019 traces. The main differences are listed below:

- The 2019 trace contains data from eight compute clusters instead of one (in case of 2011 trace);
- The 2019 trace includes more information regarding task allocations, parent-child job dependencies, and additional placement constraints;

[4]AMD Opteron 6100 series of processors

- The 2019 trace's workload has changed from most of the jobs in the workload being in the free-tier (lowest priority) to being in the best-effort batch tier;
- The overall usage for production tier (high priority) jobs in the 2019 trace has remained approximately the same as in 2011 trace;
- The job submission rate in the 2019 trace is 3.7 times higher than in 2011 trace. Consequently, the number of tasks that need to be scheduled is also 7 times higher than in 2011 trace;
- The top 1% of jobs consume over 99% of all compute and memory resources, whereas the bottom 99% of jobs consume 1% of all compute and memory resources;

Additionally, the 2019 trace also shows that vertical autoscaling resources for each task provides significant resource savings. Google's Autopilot [42] automatically scales resources both horizontally and vertically at task level to avoid resource wastage. Horizontal scaling refers to managing the number of concurrent tasks in a job, whereas vertical scaling refers to adjusting resource limits for each individual task.

The 2019 trace shows that Borg now uses abstract Google Compute Units (GCUs) instead of physical CPU cores. 1 GCU provides the same amount of computational power on any machine on any of the compute cells. Borg is responsible for mapping the GCU onto the physical CPU cores. This makes it difficult to estimate the physical resources of individual machines in the compute cells. The 2019 trace also shows that workloads differ significantly in size and mix among different cells.

DARSS is evaluated using both the 2011 trace and the 2019 trace.

4.2.2 Experimental Setup

Cluster The configuration of the cluster for the experiments is based on the information provided in the Google cluster trace. To evaluate the performance of DARSS, simulation-based testing was done. Since accessing a production cluster with thousands of machines is expensive, DARSS was evaluated by simulating a cluster with 300 nodes for 2011 trace and a cluster with 200 nodes with increased capacity for 2019 trace. The simulated clusters had the same characteristics and machine configuration as in the Google clusters. As shown in Table 4.1, the Google cluster has 53.5% of total nodes with 0.5 CPU and 0.5 RAM using the Intel Xenon processors. To reflect this percentage in the simulated cluster, 161 out of 300 machines were set to have 0.5 CPU and 0.5 memory. In total, the simulated cluster has 1905 cores and 2262 Gigabytes of memory.[5] The configuration of the simulated cluster is shown in Table 4.4.

The simulated cluster utilizes 20 machines. These machines are an Apple iMac "Core i7" (Late 2014/Retina 5K) featuring a Quad Core 4.0 GHz Intel "Core i7" (4790K) processor with four independent processor cores on a single chip, an 8 MB shared level 3 cache, 16 GB of 1600 MHz DDR3 SDRAM (PC3-12800) installed, a 1

[5]These are simulated cores and memory

Table 4.4: Resource configuration of simulated cluster

Simulated Cluster		
Number of machines	CPU	Memory
161	6 cores	8 GB
92	6 cores	4 GB
24	6 cores	12 GB
19	12 cores	16 GB
3	3 cores	4 GB
1	6 cores	2 GB

TB "Fusion Drive" (1 TB hard drive and 128 GB SSD), and an AMD Radeon R9 M290X graphics processor with 2 GB of dedicated GDDR5 memory.

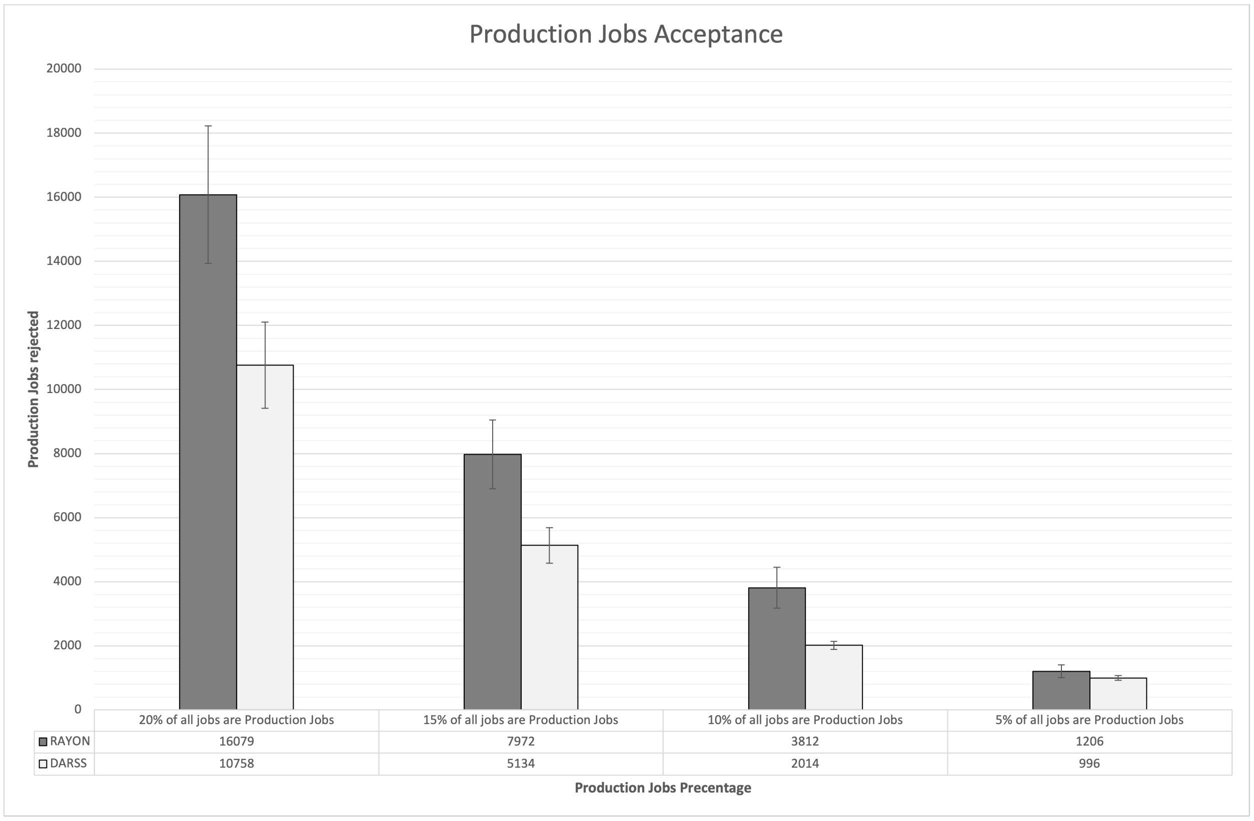
The execution of the tasks was simulated instead of actual execution due to the cluster being simulated and not the actual hardware it represents. The available resources were decreased for the duration of task execution to simulate the actual execution of tasks. To avoid interference, the experiments were conducted during summer and between the hours 12:00 AM and 5:00 AM when the machines were not in use for any other purposes. Furthermore, all the machines were restarted before running experiments to make sure that no other programs were running. All workloads have also been scaled (by limiting the max size and submission rates) to match the cluster capabilities.

When a node in the cluster finishes a task and the next task scheduled belongs to a different job, the memory and the storage on that node often need to access[6] the data associated with that job from other nodes. This is called Context Switching. In a real cluster environment, context switching introduces some overhead. Because the execution of the tasks is simulated in the experiments presented here, the overhead caused by context switching when scheduling tasks belonging to different jobs is not considered.

Context switching overhead for a task depends on several factors, such as the previous task, location of task dependencies, the type of job, etc. Context switching overhead generally falls within the range of 1 milliseconds to 12+ seconds [53]. However, overhead of tens of seconds is uncommon and occurs in specific cases. Table 4.3 shows that the average task duration is 3000 seconds in a typical cluster. In most cases the task duration is significantly higher than context switching overhead.

Since DARSS uses a task-level scheduling approach, DARSS encounters more context switches than a job-level scheduler. In comparison, Rayon (a job-level scheduler) experiences context switching overhead only when starting best-effort jobs and production jobs. When the workload is such that most tasks take an amount of time to run that is closer to the context switching overhead, task-level scheduling approach may not be feasible. Although workload distributions vary widely between different clusters, this case is

[6]by copying the data over



	20% of all jobs are Production Jobs	15% of all jobs are Production Jobs	10% of all jobs are Production Jobs	5% of all jobs are Production Jobs
■ RAYON	16079	7972	3812	1206
□ DARSS	10758	5134	2014	996

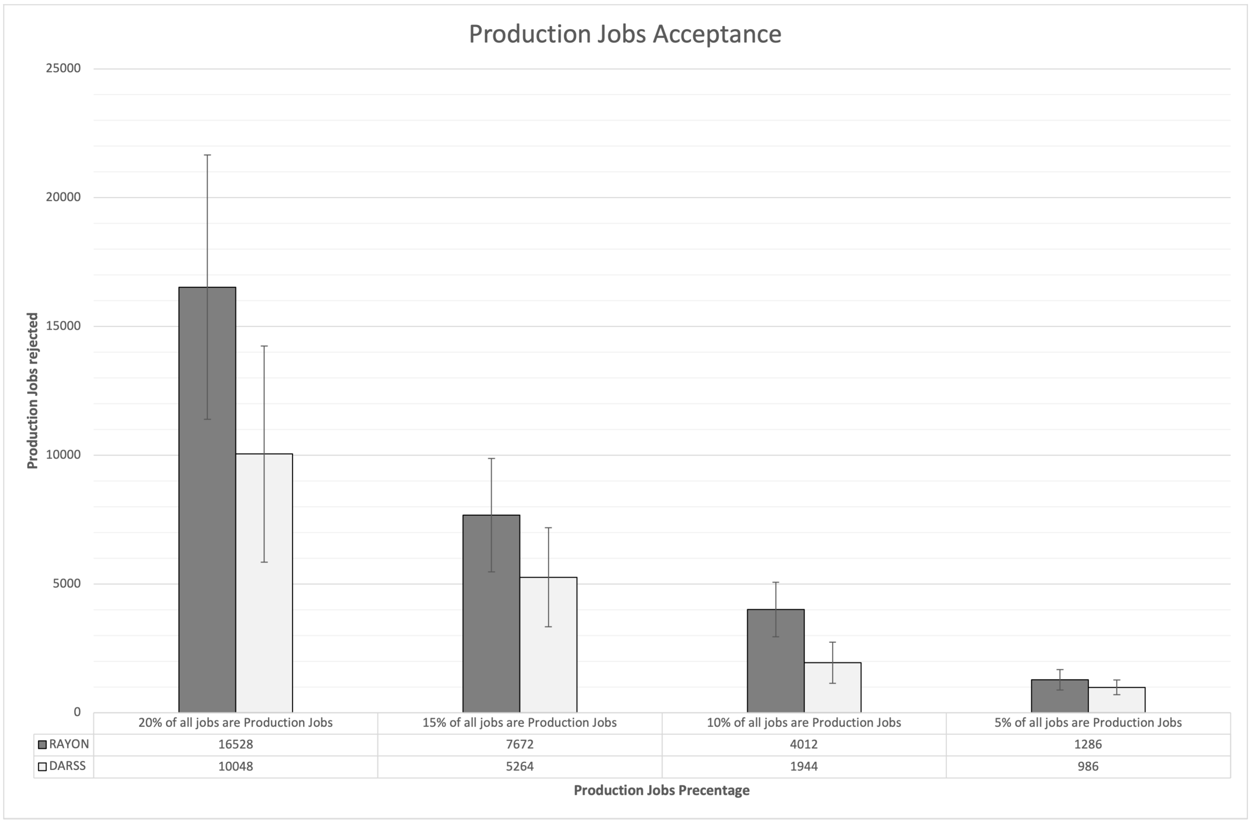
Figure 4.1: Production jobs acceptance using the 2011 Google Borg trace

uncommon [53].

Jobs In the experiments, the jobs are submitted to the cluster at a rate of 5,400 per hour from the 668,000 total jobs in the Google cluster trace for 2011 trace. For the 2019 trace, jobs were sumbitted at 3.7 times the rate in 2011 trace. For job submission, a modified version of GridMix 3.0 is used.

As mentioned before, the Capacity Scheduler (CS) was used to evaluate DARSS against YARN. The baseline CS was tuned to configure queues. The tuning was done under the assumption of perfect knowledge of the workload. To equip YARN with the best static tuning possible, information from the Google cluster trace on the queue capacity and jobs submitted to each queue is used. This is done to compare static tuning against DARSS's dynamic scheduling.

The 2011 trace consists of jobs submitted from 679 users or tenants. The workload randomly chosen from 2019 trace (cell h) consists of jobs submitted from 548 users. To provide a fair capacity to jobs submitted by each user, 679 or 548 queues could be created. However, recall that the workload consists of a mix of workflows and best-effort short jobs. To take advantage of the mechanisms in CS, it is best to create two queues (one for production jobs and one for best-effort jobs) for each user. In total, there are 1358 queues created for 2011 trace. For 2019 trace, a total of 1096 queues are created. The capacity assigned to each of these queues is based on the known resource requirements of jobs submitted to each of these queues. The



	20% of all jobs are Production Jobs	15% of all jobs are Production Jobs	10% of all jobs are Production Jobs	5% of all jobs are Production Jobs
▩ RAYON	16528	7672	4012	1286
□ DARSS	10048	5264	1944	986

Figure 4.2: Production jobs acceptance using the 2019 Google Borg trace

Capacity Scheduler makes it possible to assign a guaranteed minimum capacity to each queue. Furthermore, it enables the maximum resources assigned to a queue to be over its capacity. The maximum capacity of a queue equivalent to two times the guaranteed capacity is set to achieve a sensible balance of resource utilization and guarantees to jobs [16].

The second comparison is against Rayon. Rayon's configuration, in comparison to YARN, does not require tuning. For both Rayon and DARSS, a maximum of 75% of resources are reserved for production workflows. This percentage is based on the average number of best-effort jobs submitted per hour, the tasks' duration within jobs, and the average amount of resources required to execute a task for best-effort jobs. All of this information is know ahead of time and derived from the Google cluster trace.

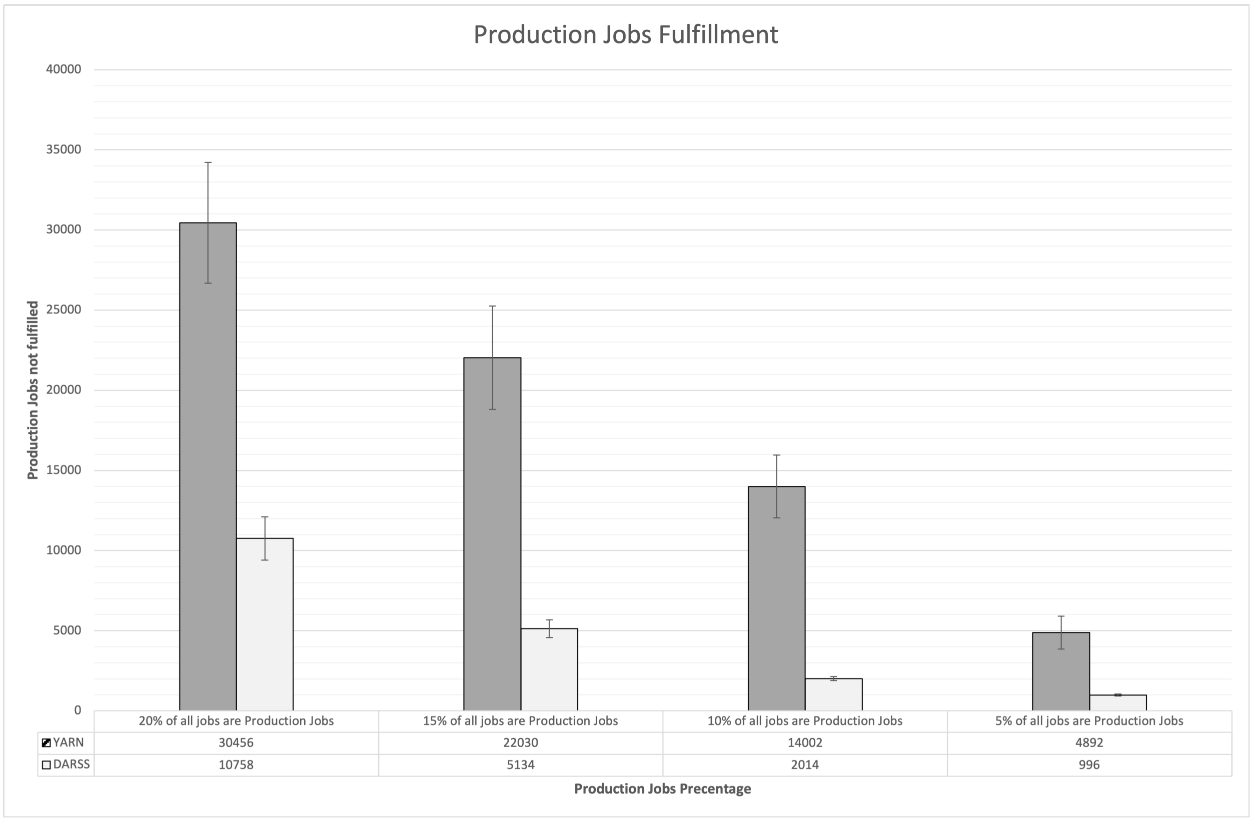
4.2.3 Evaluation And Results

DARSS is evaluated against YARN (CS) and Rayon according to the following metrics:[7]

1. Production job acceptance: percentage of production workflows accepted by the system;

2. Production job fulfillment: percentage of accepted workflows completed before deadlines;

[7]defined over a period of time



	20% of all jobs are Production Jobs	15% of all jobs are Production Jobs	10% of all jobs are Production Jobs	5% of all jobs are Production Jobs
▨ YARN	30456	22030	14002	4892
☐ DARSS	10758	5134	2014	996

Figure 4.3: Production jobs fulfillment using the 2011 Google Borg trace

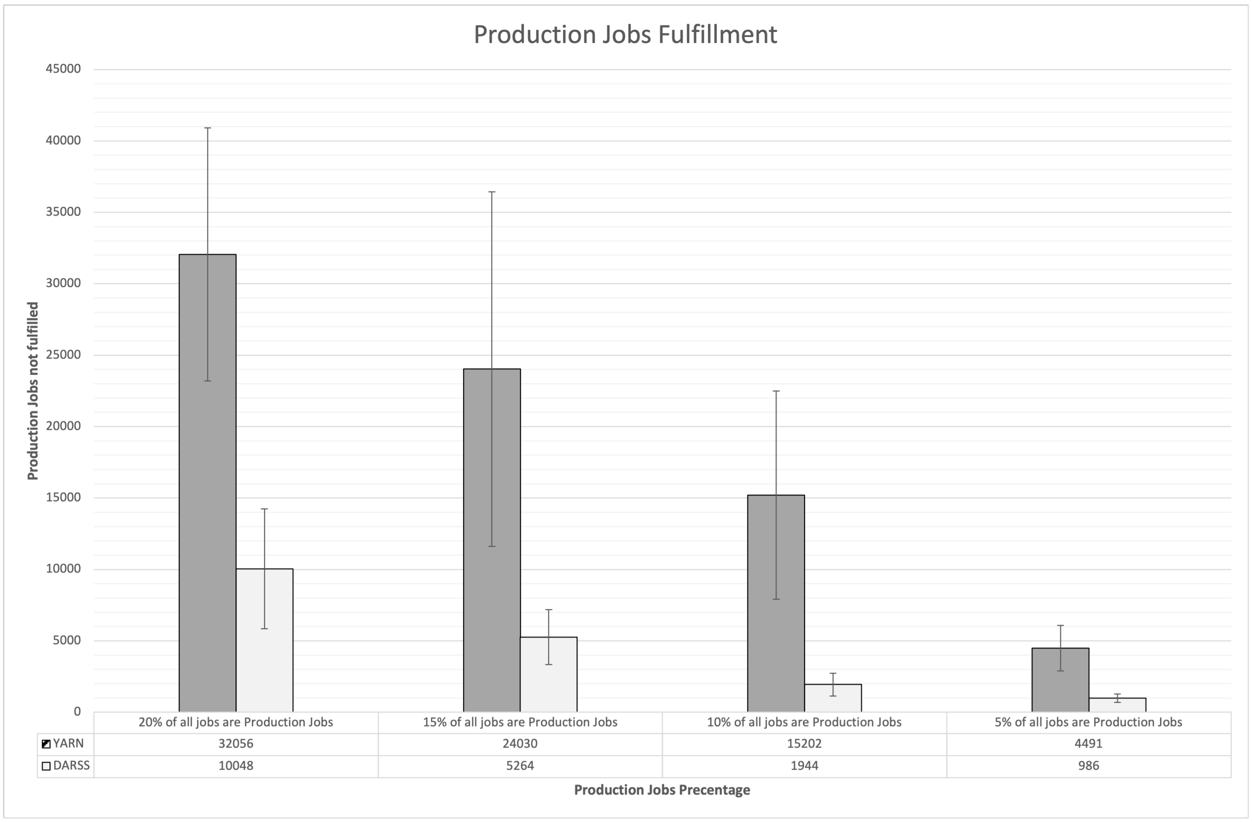
3. Best-effort jobs serviced: number of total best-effort jobs completed by the system; and
4. Best-effort jobs latency: service time for best-effort jobs.

The experiment is conducted with four configurations of workload distribution. 10 simulation runs per trace per system per configuration of workload distribution are conducted. In total, 240 simulation runs are conducted. The four configurations of workload distribution are:

- 20% of all jobs are production jobs;
- 15% of all jobs are production jobs;
- 10% of all jobs are production jobs; and
- 5% of all jobs are production jobs.

4.2.3.1 Production job acceptance

Figures 4.1 and 4.2 show the the difference in Rayon and DARSS's production job acceptance using the 2011 trace and the 2019 trace, respectively. Rayon rejects 12% of production jobs whereas DARSS rejects 7% of production jobs with the same workload distribution when the percentage of production jobs is 20%. This



	20% of all jobs are Production Jobs	15% of all jobs are Production Jobs	10% of all jobs are Production Jobs	5% of all jobs are Production Jobs
▨ YARN	32056	24030	15202	4491
□ DARSS	10048	5264	1944	986

Figure 4.4: Production jobs fulfillment using the 2019 Google Borg trace

means that Rayon accepts 88% of production jobs, whereas DARSS accepts 92% of production jobs when 20% of all jobs are production jobs. This is due to the key differences in the architecture of the two systems mentioned in chapter 3. DARSS has greater fluidity in scheduling tasks than Rayon. Rayon schedules entire production jobs as a single job, whereas DARSS schedules individual tasks within those jobs. This increases the chances of production jobs that are submitted late getting accepted. Figure 4.1 also shows a similar difference in production job rejection rate by the two systems when the percentages of production jobs are 15% and 10%.

The difference in rejection rates of DARSS and Rayon is highly dependent on the percentage of production jobs submitted. As we see in Figure 4.1, the difference between the number of jobs rejected by the two systems is significantly reduced when the percentage of production jobs is 5%. This is due to the total number of production jobs submitted. When the percentage of production jobs submitted is fewer than 15%, DARSS rejects only 3% of all jobs. The results show that DARSS accepts more production jobs than Rayon in all scenarios considered in this experiment.

4.2.3.2 Production jobs fulfillment

Figures 4.3 and 4.4 compare YARN's Capacity Scheduler with DARSS according to the workflow fulfillment when using the 2011 trace and 2019 trace, respectively. Since YARN is deadline agnostic, it accepts all the

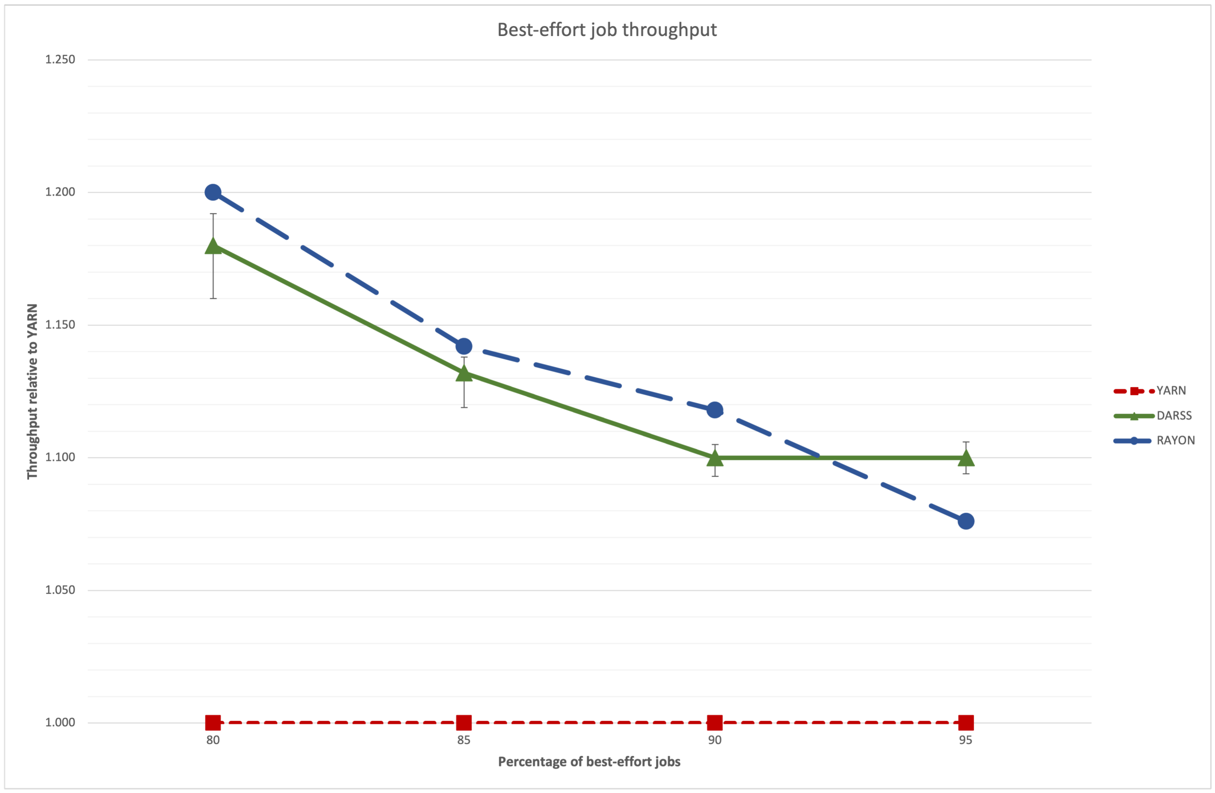


Figure 4.5: Best-effort jobs serviced with respect to YARN using the 2011 Google Borg trace

jobs submitted to the system. YARN fails to complete about 26% of production jobs before their respective deadlines in the case where 20% of all jobs are production jobs. As opposed to YARN, DARSS and Rayon fulfill 100% of jobs that are accepted. Unlike YARN, DARSS and Rayon do not accept all the production jobs that are submitted. DARSS analyzes the production job requirements and detects that it can not accept all the jobs submitted to the system. It rejects 7% of the production jobs that were too large to fit in the schedule before their deadline when the percentage of production jobs is 20%. Although these rejections are pessimistic, they happen at reservation time. This is better than a violation of a job's deadline without any indication. Rejected jobs mean freed resources that are used by other best-effort jobs or tasks belonging to other production jobs which increases other metrics.

4.2.3.3 Best-effort jobs serviced

Figures 4.5 and 4.6 show the throughput of Best-effort jobs with respect to YARN when using the 2011 trace and the 2019 trace, respectively. The highest relative performance in throughput achieved by DARSS is at 80% best-effort jobs, i.e., 18% increase in throughput relative to YARN. In contrast, for higher percentages of best-effort jobs submitted, the number of completed best-effort jobs decreases. This may seem counter-intuitive, but this is because the throughput of best-effort jobs is dependent on the production job acceptance rate. A higher production job acceptance rate means more resources are reserved for production job tasks.

This means fewer available resources for best-effort jobs, which translates to lower throughput. Recall that DARSS has a specified number of reserved containers for best-effort jobs. This provision helps in best-effort job throughput.

Since YARN does not reject any jobs, some production jobs (that would have been rejected in the case of DARSS) hog the resources. These production jobs miss their deadlines while occupying resources causing lower throughput for best-effort jobs.

Rayon has a slightly higher throughput than DARSS for best-effort job percentages lower than 95%. This is caused by the higher rejection rate relative to DARSS. Since Rayon rejects more production jobs, it has more resources available to serve best-effort jobs.[8]

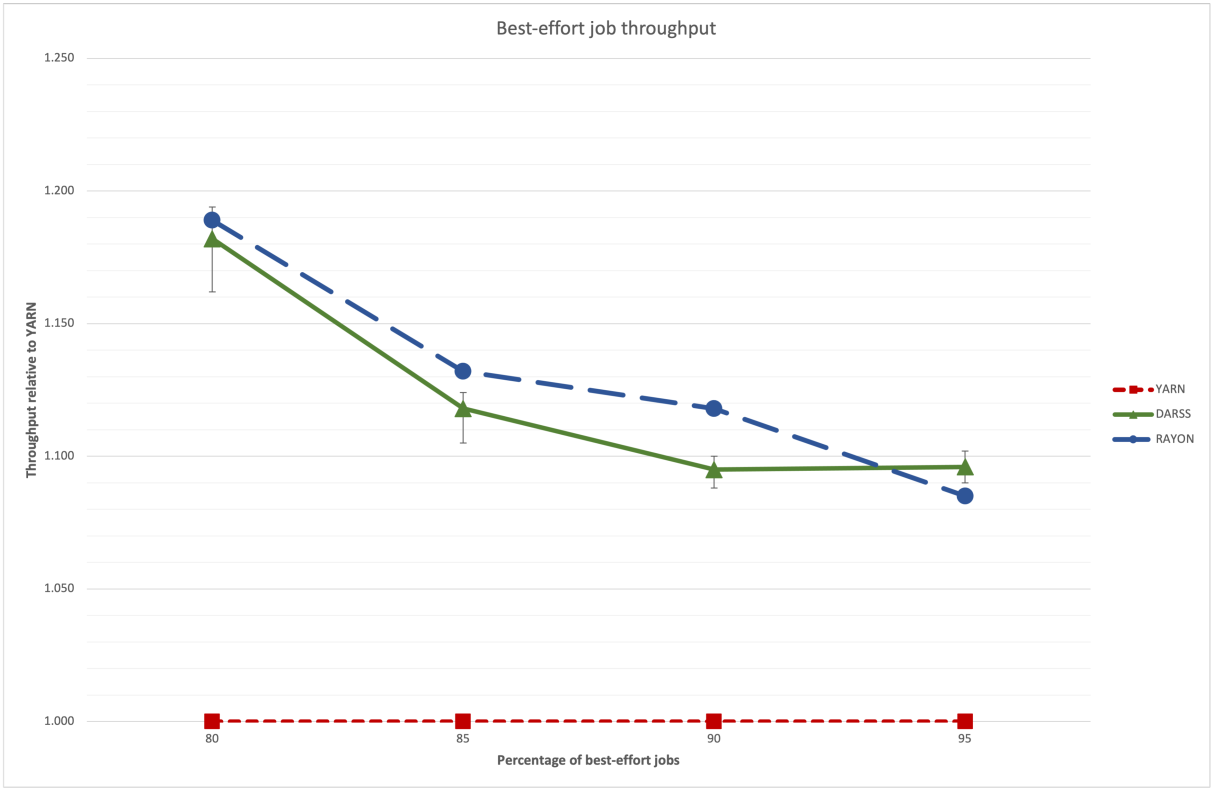


Figure 4.6: Best-effort jobs serviced with respect to YARN using the 2019 Google Borg trace

4.2.3.4 Best-effort jobs latency

Figures 4.7 and 4.8 show the improved latency with respect to YARN when using the 2011 trace and the 2019 trace, respectively. The highest percentage of jobs with lower latency achieved by DARSS is at 80% best-effort jobs, i.e., 66% of the jobs have lower latency relative to YARN when using the 2019 trace. For higher percentages of best-effort jobs submitted, the percentage of best-effort jobs with reduced latency decreases. Similar to throughput, latency reduction for best-effort jobs is dependent on the production job acceptance

[8]provided that there are enough best-effort jobs waiting for service

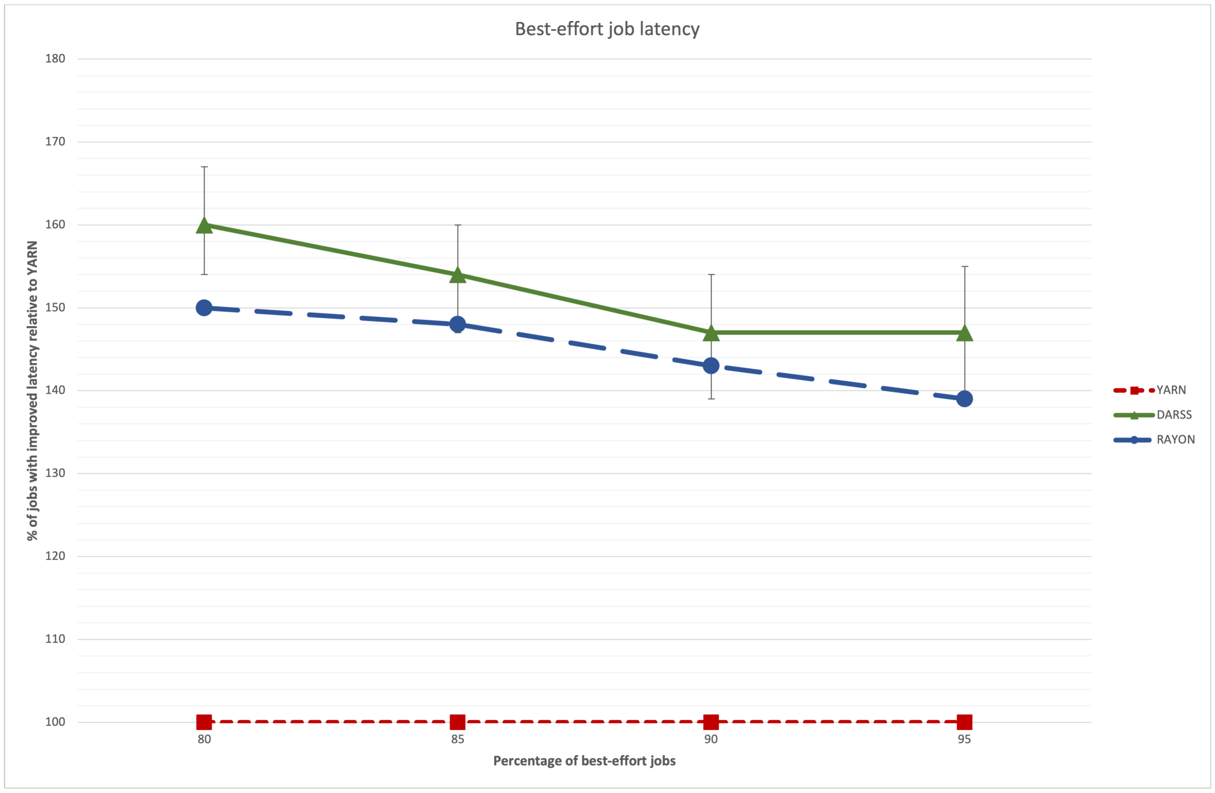


Figure 4.7: Best-effort jobs with improved latency with respect to YARN using the 2011 Google Borg trace

rate. A higher production job acceptance rate means more resources are reserved for production job tasks. Due to this, there are fewer available resources for best-effort jobs, which means some best-effort jobs have to wait for a production job to be completed before its deadline. DARSS's reservation mechanism for best-effort jobs helps in not letting best-effort jobs stale.

Since YARN does not reject any jobs, some production jobs (that would have been rejected in the case of DARSS) hog the resources. These production jobs miss their deadlines while occupying resources causing best-effort jobs to wait for their completion.

DARSS has a higher percentage of jobs with reduced latency than Rayon for all the distributions of the workload. Although Rayon has a higher rejection rate relative to DARSS, the scheduling fluidity achieved by scheduling individual tasks instead of entire production jobs increases the number of best-effort jobs being serviced earlier.

Furthermore, DARSS takes advantage of the fact that production jobs are not concerned with latencies (provided their deadline is not approaching). This reversal of priority causes a higher average latency for production jobs. Due to this, best-effort jobs are scheduled earlier, thus improving their latency.

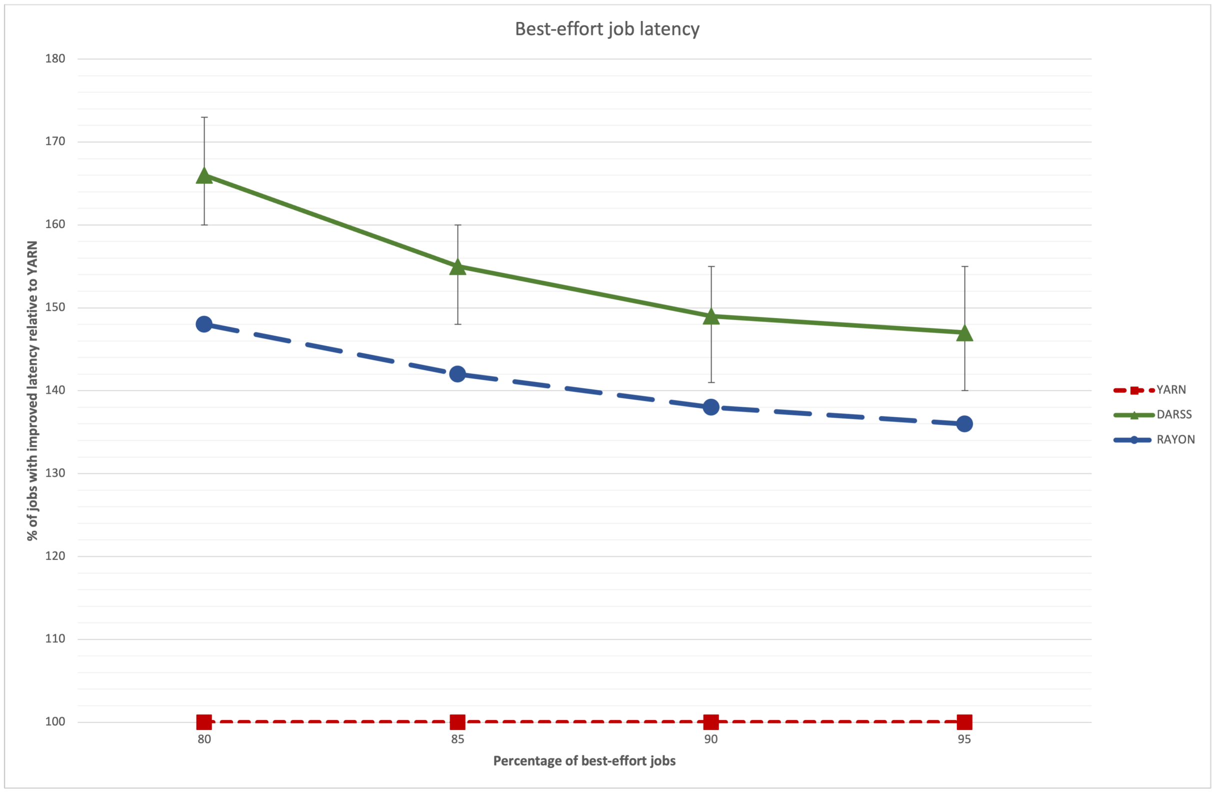


Figure 4.8: Best-effort jobs with improved latency with respect to YARN using the 2019 Google Borg trace

4.2.4 Discussion

Thorough experiments conducted in this research show that DARSS achieves the objectives set in Chapter 1. The results of the experimental evaluation illustrate a few key points of DARSS's comparative performance:

- 100% Production Job Completion Guarantee: DARSS completes all of the accepted production jobs before their deadlines;
- Lower Production Job Rejection Rate: DARSS rejects fewer production jobs compared to Rayon for all four cases considered in the experiments;
- Higher Production Job Fulfilment Rate: DARSS fulfills a higher number of production jobs than either Rayon or YARN for all four cases considered in the experiments;
- Higher Throughput of Best-Effort Jobs: DARSS completes more best-effort jobs than YARN for all four cases considered in the experiments and Rayon for the case where 5% of all jobs are best-effort jobs; and
- Lower Latency for Best-Effort Jobs: a higher percentage best-effort jobs completed by DARSS encounter low latency than either Rayon or YARN for all four cases considered in the experiments.

The above points illustrate that DARSS performs better than YARN and Rayon across all four metrics. High job throughput means high ROI. 100% production job completion guarantee means high confidence in deadline-sensitive and business-critical job completion. Lower latency for best-effort jobs means a higher user satisfaction rate for the best-effort tier customers. All of these points make DARSS an attractive option for multi-tenant cluster environments.

Chapter 5
CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Cluster computing is an attractive option for a diverse range of computational needs due to its lower costs and ease of application development. Modern big-data clusters run a mix of production jobs and best-effort jobs. These two types of jobs have varying scheduling needs, which puts diverse resource demands on the cluster scheduling system. To cope with this diversity, a cluster scheduler should be deadline-aware, be accurate in predicting completion of jobs, guarantee completion before the deadline to the accepted jobs and have low latency for best-effort jobs. A Survey of existing cluster schedulers (Chapter 2) shows that most of these systems lack a crucial aspect of scheduling in a production cluster, i.e., temporal requirements of production workflows.

This thesis makes a case for deadline-aware reservation-based scheduling. This is an approach that takes advantage of explicit information about a job's temporal requirements and resource requirements to plan the schedule of the cluster continually. This approach is used to design a scheduling system, DARSS. A system capable of handling workloads consisting of multiple interdependent jobs, completing accepted complex production workflows before their deadlines, providing lower latency for interactive short jobs that do not have deadlines.

To reduce the best-effort jobs latency, the placement heuristic used in the system places a production workflow as close to its deadline as it can. The production workflows consist of smaller jobs. These smaller jobs consist of even smaller jobs called tasks.[1] DARSS introduces high flexibility and fluidity in scheduling by scheduling the workflows at the low-level tasks. This allows the system to reduce latency for best-effort jobs even further and improves the chances of jobs that are submitted late being accepted. The experimental evaluation (Chapter 4) confirms the planning of the cluster schedule enables DARSS to complete production jobs before their deadlines while providing low latency to best-effort ad-hoc jobs.

5.2 Contributions

This research makes three key contributions to the field of cluster computing and cluster scheduling.

[1]These smaller jobs and tasks can be interdependent

The first contribution of this study is creating a taxonomy of resource managers (Chapter 2) to explain the current state-of-the-art in cluster resource management. This taxonomy, along with the survey of mainstream cluster scheduling systems, provides a structure for future works in the field of cluster scheduling and resource management.

The second and the most important contribution of this research is designing DARSS. It is a scalable, fault-tolerant, and high throughput resource management system for the cluster computing environment. DARSS is scalable in different dimensions as the number of computing resources, the number of users, and the number of executing jobs. DARSS scalability not only does not degrade its fault-tolerance but also improves the robustness of the system because of the modularity in design. Also, thorough experiments based on Google cluster trace show that DARSS has a very high throughput decision-making process which is a necessary feature in a cluster computing environment.

The last but not the least contribution of this work is using YARN as a base for implementing a cluster resource management system. DARSS is prototyped based on reactive systems manifesto, which promotes systems that are responsive in a timely manner, resilient in the face of failure, elastic to the workload, and message-driven based on asynchronous message passing. Being a reactive system, DARSS is very flexible to future changes and improvements. Furthermore, DARSS components are loosely coupled, which improves its fault-tolerance significantly.

5.3 Future Work

One of the interesting challenges in resource management is serving high-priority jobs while all the resources are allocated. This is even more challenging in a cluster computing environment since there are fragmented free resources across a cluster's nodes which are enough for serving a high-priority job. In operating systems, this challenge is usually managed by preempting tasks of a low priority job and executing the high priority tasks in place of them. The same as the application of allocation policies, applying preemption requires a central authority to make decisions. Although DARSS has centralized resource allocation, there is no preemption mechanism in it. Exploring preemption mechanisms for DARSS and implementing them would be an exciting topic for future works.

Another unexplored area in this study is serving jobs with gang-scheduling requirements. Tasks of a gang-scheduling job must be started all at the same time, and usually, a failure in one of them leads to complete job failure. Although this kind of job is rare, e.g. there were no jobs with gang-scheduling requirement in Google's trace, they introduce an interesting set of challenges which are out of the scope of this study. For example, resource hoarding for the tasks of a gang-scheduling job can lead to high resource under-utilization and task starvation. Also, scheduling their tasks on reliable machines is another challenge that requires lots of trade-offs in the scheduler. Currently, job managers in DARSS are capable of asking the resource manager for resources in the case of gang-scheduling demand; however, managing this requirement within distributed schedulers or allocating resources on reliable machines introduces new challenges.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, Seattle, Washington, August 2008.

[2] Amazon. Placement groups, 2020. URL: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html`.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, page 12, Napa, California, May 2011.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[5] Apache Aurora. Apache aurora, 2020. URL: `http://aurora.apache.org/`.

[6] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines, third edition. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, October 2018.

[7] Kedar Bellare, Carlo Curino, Ashwin Machanavajihala, Peter Mika, Mandar Rahurkar, and Aamod Sane. Woo: A scalable and multi-tenant platform for continuous knowledge base synthesis. *Proceedings of the VLDB Endowment*, 6(11):1114–1125, August 2013.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, OSDI'14, page 285–300, Broomfield, Colorado, October 2014.

[9] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems.* Prentice Hall PTR, USA, 1999.

[10] Wei Chen, Xiaobo Zhou, and Jia Rao. Preemptive and low latency datacenter scheduling via lightweight containers. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2749–2762, December 2020.

[11] Wenyan Chen, Kejiang Ye, Yang Wang, Guoyao Xu, and Cheng-Zhong Xu. How does the workload look like in production cloud? analysis and clustering of workloads on alibaba cluster trace. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 102–109, December 2018.

[12] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, August 2012.

[13] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Design insights for mapreduce from diverse production workloads. Technical Report UCB/EECS-2012-17, EECS Department, University of California, Berkeley, January 2012. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-17.html`.

[14] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, page 390–399, Singapore, July 2011.

[15] Avery Ching, Ravi Murthy, Dmytro Molkov, Ramkumar Vadali, and Paul Yang. Under the hood: Scheduling mapreduce jobs more efficiently with corona. `URL:https://engineering.fb.com/2012/11/08/core-data/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona`, November 2012.

[16] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, Seattle, Washington, November 2014.

[17] D2iQ Inc. Marathon, A container orchestration platform for Mesos and DC/OS. URL: `https://mesosphere.github.io/marathon/`.

[18] Docker. Docker swarm mode overview, 2020. URL: `https://docs.docker.com/engine/swarm/`.

[19] DOMO. Data never sleeps 8.0, 2020. URL: `https://www.domo.com/learn/data-never-sleeps-8`.

[20] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, Bern, Switzerland, April 2012.

[21] Cloud Native Computing Foundation. Production-grade container orchestration. URL: `https://kubernetes.io/`.

[22] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest grow the in the far east. *IDC IView*, 2012. URL: `https://www.cs.princeton.edu/courses/archive/spring13/cos598C/idc-the-digital-universe-in-2020.pdf`.

[23] John Gantz, David Reinsel, and John Rydning. Worldwide global datasphere forecast, 2020 – 2024: The covid-19 data bump and the future of data growth. *Market Forecast*, 2020. URL: `https://www.idc.com/getdoc.jsp?containerId=US44797920`.

[24] Laurence Goasduff. Gartner top 10 trends in data and analytics for 2020. URL : https://www.gartner.com/smarterwithgartner/gartner-top-10-trends-in-data-and-analytics-for-2020, October 2020.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 295–308, Boston, Massachusetts, March 2011.

[26] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: Towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, Scottsdale, Arizona, May 2012.

[27] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619, July 2015.

[28] Changqing Ji, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. Big data processing in cloud computing environments. In *12th International Symposium on Pervasive Systems, Algorithms and Networks*, pages 17–23, San Marcos, Texas, December 2012.

[29] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 94–103, Melbourne, Australia, May 2010.

[30] Andrew D. Konwinski. *Multi-Agent Cluster Scheduling for Scalability and Flexibility*. PhD thesis, USA, 2012. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-273.pdf`.

[31] Willis Lang and Jignesh M. Patel. Energy management for mapreduce clusters. *Proceedings of the VLDB Endowment*, 3(1–2):129–139, September 2010.

[32] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, March 2010.

[33] Shen Li, Shaohan Hu, Shiguang Wang, Lu Su, Tarek Abdelzaher, Indranil Gupta, and Richard Pace. Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, page 93–103, Madrid, Spain, June 2014.

[34] Stathis Maroulis, Nikos Zacheilas, and Vana Kalogeraki. A holistic energy-efficient real-time scheduler for mixed stream and batch processing workloads. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2624–2635, December 2019.

[35] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 205–216, Washington, District of Columbia, March 2009.

[36] Sara Mustafa, Iman Elghandour, and Mohamed Ismail. A machine learning approach for predicting execution time of spark jobs. *Alexandria Engineering Journal*, 57, November 2018.

[37] Ashley I. Naimi and Daniel J. Westreich. Big Data: A Revolution That Will Transform How We Live, Work, and Think. *American Journal of Epidemiology*, 179(9):1143–1144, April 2014.

[38] Gregory F. Pfister. *In Search of Clusters*. Parallel programming computer architecture. Prentice Hall PTR, 1998. URL: `https://books.google.com/books?id=XlUZAQAAIAAJ`.

[39] Raconteur. A day in data, 2020. URL: `https://www.raconteur.net/infographics/a-day-in-data/`.

[40] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, San Jose, California, October 2012.

[41] Maria A. Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719, November 2018.

[42] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, April 2020.

[43] Madhulina Sarkar, Triparna Mondal, Sarbani Roy, and Nandini Mukherjee. Resource requirement prediction using clone detection technique. *Future Generation Computer Systems*, 29(4):936–952, June 2013.

[44] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, Prague, Czech Republic, April 2013.

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, page 1–10, Incline Village, Nevada, May 2010.

[46] Manjeet Singh. An overview of grid computing. In *2019 International Conference on Computing, Communication, and Intelligent Systems*, pages 194–198, Greater Noida, India, October 2019.

[47] Thomas Sterling, Ewing Lusk, and William Gropp. *Beowulf Cluster Computing with Linux.* Scientific and Computational Engineering Series. MIT Press, 2002. URL: `https://books.google.com/books?id=M73h9u9Q7sAC`.

[48] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the 15th European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, April 2020.

[49] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen. A comprehensive feature comparison study of open-source container orchestration frameworks. *Applied Sciences*, 9(5), March 2019.

[50] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, October 2013.

[51] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, Bordeaux, France, April 2015.

[52] John Wilkes and Charles Reiss. Borg cluster data trace, 2012. URL: `https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md`.

[53] John Wilkes and Charles Reiss. Borg cluster data trace, 2020. URL: `https://github.com/google/cluster-data/blob/master/ClusterData2019.md`.

[54] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 265–278, Paris, France, April 2010.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, Boston, Massachusetts, June 2010.

[56] Xuezhi Zeng, Saurabh Garg, Mutaz Barika, Albert Y. Zomaya, Lizhe Wang, Massimo Villari, Dan Chen, and Rajiv Ranjan. Sla management for big data analytical applications in clouds: A taxonomy study. *ACM Computing Surveys*, 53(3), June 2020.

[57] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 7(13):1393–1404, August 2014.