MACRM: A Multi-agent Cluster Resource Management system

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

By
Mohammad Hossein Sedighi Gilani

Permission to Use

Abstract

The falling cost of cluster computing has significantly increased its use in the last decade. As a result, the number of users, the size of clusters, and the diversity of jobs that are submitted to clusters have grown. These changes lead to a quest for redesigning of clusters' resource management systems. The growth in the number of users and increase in the size of clusters require a more scalable approach to resource management. Moreover, ever-increasing use of clusters for carrying out a diverse range of computations demands fault-tolerant and highly available cluster management systems. Last, but not the least, serving highly parallel and interactive jobs in a cluster with hundreds of nodes, requires high throughput scheduling with a very short service time.

This research presents MACRM, a multi-agent cluster resource management system. MACRM is an adaptive distributed/centralized resource management system which addresses the requirements of scalability, fault-tolerance, high availability, and high throughput scheduling. It breaks up resource management responsibilities and delegates it to different agents to be scalable in various aspects. Also, modularity in MACRM's design increases fault-tolerance because components are replicable and recoverable. Furthermore, MACRM has a very short service time in different loads. It can maintain an average service time of less than 15ms by adaptively switching between centralized and distributed decision making based on a cluster's load.

Comparing MACRM with representative centralized and distributed systems (YARN [67] and Sparrow [52]) shows several advantages. We show that MACRM scales better when the number of resources, users, or jobs increase in a cluster. As well, MACRM has faster and less expensive failure recovery mechanisms compared with the two other systems. And finally, our experiments show that MACRM's average service time beats the other systems, particularly in high loads.

Acknowledgements

I would like to take this opportunity to say thanks from the bottom of my heart to my supervisor, Dr. Nadeem Jamali. His guidance, support, encouragement and suggestions, as well as his flexibility in allowing me to adapt the scope and focus of this work were instrumental in completion of this thesis. I am grateful to him for accepting me under his supervision as well as for the financial support he provided me.

I would like to thank Professor Chanchal Roy, Professor Kevin Stanely, 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 would like to thank my friends and colleagues Ahmed Abdel Moamen, Hongxing Geng, and Sander Wang for their support through the course of my studies. They spared much time listening to my practice presentations and offering comments.

And finally, I am grateful to the Department of Computer Science at University of Saskatchewan for the Departmental Teaching Scholarship, and to the Natural Science and Engineering Research Council (NSERC) of Canada for the funding through Professor Jamali's Discovery Grant. Also, special thanks go to departmental administration staffs, especially Ms. Gwen Lancaster, who coordinated and helped whenever needed.

Contents

Permission to Use **i**

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

List of Tables **vi**

List of Figures **vii**

List of Abbreviations **viii**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Limitations in Current State of The Art . . . 2
- 1.3 An Adaptive Distributed/Centralized System . . . 4
- 1.4 Objectives . . . 6
- 1.5 Thesis Structure . . . 7

2 Background and Related Works **8**
- 2.1 Cluster Computing Environment . . . 8
- 2.2 Characteristics of Production Workload . . . 9
- 2.3 Taxonomy of Cluster Scheduling Systems . . . 11
- 2.4 Survey of Existing Systems . . . 13
  - 2.4.1 Sequential Centralized Schedulers . . . 13
  - 2.4.2 Parallel Centralized Schedulers . . . 15
  - 2.4.3 Sequential Distributed Schedulers . . . 17
  - 2.4.4 Concurrent Distributed Schedulers . . . 20
- 2.5 From Map-Reduce to MACRM . . . 22
  - 2.5.1 Disk-locality Is Not a Scheduling Concern Anymore . . . 23
  - 2.5.2 Does Not Scale . . . 25

3 System Design and Implementation **28**
- 3.1 MACRM Architecture . . . 28
  - 3.1.1 Let's Divide It Again . . . 28
  - 3.1.2 Low Scheduling Latency in MACRM . . . 31
  - 3.1.3 Power of Two Choices . . . 33
  - 3.1.4 Adaptive Sampling Rate . . . 35
  - 3.1.5 Switch to Centralized Resource Management . . . 39
- 3.2 MACRM Implementation . . . 41
  - 3.2.1 Actor-based System Design . . . 41
  - 3.2.2 Resource Arbitrator Structure . . . 43
  - 3.2.3 Resource Tracker Structure . . . 45

4 Experimental Evaluation **47**
- 4.1 Experiment setup . . . 48
  - 4.1.1 Google Cluster Trace . . . 48
  - 4.1.2 Simulation Setup . . . 50

4.2 Evaluation And Results 51
4.2.1 System Service Time 51
4.2.2 Scheduling Throughput 54
4.2.3 System Scalability 55
4.2.4 Fault-tolerance 59

5 Conclusion and Future Works **62**
5.1 Conclusion 62
5.2 Contributions 62
5.3 Future Works 63

References **65**

A detailed values of the throughput experiment **70**

List of Tables

2.1 The summary of the presented systems 22

4.1 Configuration of machines in Google's cluster. Resources are linearly scaled so that the maximum value for CPU and memory is one. 49
4.2 Interpretation of Table 4.1 49
4.3 The workload's statistics of Google cluster trace 51
4.4 Scalability comparison of MACRM, YARN, and Sparrow 57
4.5 Comparison of MACRM, YARN, and Sparrow in the case of failures. The first part indicates how fast a system can recognize and recover from a failure and the second part shows recovery overhead compared with the other two systems. 61

A.1 Sparrow's results for different initial loads. 70
A.2 MACRM's results for different initial loads. 71
A.3 YARN's results for different initial loads. 72

List of Figures

2.1 Taxonomy of cluster scheduling 11
2.2 Quasar Scheduler [23] 14
2.3 YARN Architecture. Blue parts are system components. Yellow and pink are two running applications [67] 17
2.4 Resource offer example [38] 19
2.5 Comparison of scheduling techniques in a simulated cluster of 10,000 4-core machines running jobs with 100-tasks [52] 21
2.6 Comparison of disk and memory read bandwidths locally and across a network [3] 24
2.7 A fat-tree network topology [2]. 25

3.1 MACRM's system components and their communications 30
3.2 Average execution time for some of the data analytics frameworks. 32
3.3 Node Manager logic for managing node samplings. 34
3.4 Average service time with the power of two choices 36
3.5 Comparison of the power of two choices (in blue) vs. the power of two choices with doubling sampling rate after each failure (in red). 37
3.6 Effects of different parameters on average service time in sampling with the power of two choices at 85% load 38
3.7 Average service time comparison of the adaptive initial sampling rate method (green) with the power of two choices (blue) and the power of two choices with doubling sampling rate after failure (red). 39
3.8 Average service time comparison of the hybrid event-driven/periodic heartbeat + parallel request serving (red) with the power of two choices + adaptive initial sampling rate (blue). . 41
3.9 Resource Arbitrator architecture (Parentheses represent destination/source of the messages). 43
3.10 A scheduler actor pseudo code 45

4.1 Cluster load changes for a sample from each group of the loads. Blue lines present CPU load and red lines present memory load. 52
4.2 Average service time for YARN, MACRM, and Sparrow in a cluster with 500 nodes. The heartbeat inter-arrival time is 2000ms and the timeout for sampling retry in both MACRM and Sparrow is 3ms. 54
4.3 Throughput comparison of Sparrow, MACRM, and YARN. 56

List of Abbreviations

API	Application Program Interface
EIP	Enterprise Integration Patterns
HDFS	Hadoop Distributed File System
JM	Job Manager
MACRM	Multi-Agent Cluster Resource Management
NM	Node Manager
RDD	Resilient Distributed Dataset
RM	Resource Manager
ROI	Return On Investment
RT	Resource Tracker
RTT	Round Trip Time
SLO	Service Level Objectives

Chapter 1
Introduction

This chapter presents a high-level overview of this work on cluster resource management and scheduling. The first section briefly presents motivations. After that, Section 1.2 identifies limitations of cluster management systems. Section 1.3 describes the approach to cluster resource management taken in this research. Section 1.4 lists the contributions of this research, and finally Section 1.5 describes the organization of the rest of the thesis.

1.1 Motivation

In the last two decades, the continuous increase of computational power has produced an overwhelming amount of data which has called for a paradigm shift in the computational architecture and large-scale data processing methods [59]. Large IT companies (such as Google, Microsoft, Facebook, Amazon, twitter, Yahoo! and more) have used cluster computing [56, 62] to support their large-scale Internet services, data-intensive scientific applications, and enterprise analytics [48]. A cluster is a distributed computing system which consists of a collection of standalone computers working together as a single integrated system [11]. In order to develop applications for clusters, application developers employ cluster programming models which simplify the development process. Cluster programming models are frameworks that manage and execute collections of software executables called tasks.

Cluster computing is attractive because of its low development cost, availability of off-the-shelf and vendor independent hardware components which form these clusters, flexibility of configuration and upgrade, high performance, and a diverse range of programming models. A number of reasons make cluster computing an attractive computing platform for both system designers and application developers. Lower cost, availability of components, and flexibility of configuration compared to supercomputers are attractive features in both clusters and Grids for system designers. Utilization of commodity hardware and open-sourced software stack are the main reasons for ease of configuration and maintenance.

The ease of software development and programming, high performance, and availability of a diverse range of programming models are features that attract application developer to clusters rather than Grids or supercomputers. Simplicity of software development and programming is a result of tight coupling of nodes in clusters which presents all the distributed resources as a single system to developers. While in Grid computers, every node is autonomous (i.e. it has its own resource manager) and behaves like an independent entity.

1.2 Limitations in Current State of The Art

Cluster computing frameworks[1] rely on services of cluster middleware for application execution. A cluster middleware is a software system that creates the illusion of a single system for applications and hides the complexity of the underlaying architecture [12]. Two main components of a cluster middleware are a distributed file system and a resource manager. The distributed file system pools the storage capacity of a cluster's servers and provides a reliable and scalable data storage system. Prominent examples include Amazon S3 [53], Windows Azure Storage Blobs [14], IBM general parallel file system [21], and Hadoop Distributed File System (HDFS) [61]. Similarly, the resource manager puts the distribution of computing resources out of sight and provides an uniform interface to access CPU, memory and network bandwidth. To this end, the resource manager is in charge of resource monitoring, job queuing, job scheduling, and controlling job execution [70].

Designing a stand alone resource management system that can provide all these services, like an operating system, is not a viable solution in cluster computer environment. This is because of the following facts:

- Fault tolerance: Deployment of commodity servers and network devices causes higher overall rate of failure in clusters compared to stand-alone systems [9, 37]. As a result, managing hardware failures and software errors is more challenging for resource managers in clusters.
- Scalability and throughput: Production clusters usually contain hundreds to thousands of servers which are responsible for serving thousands of tasks simultaneously. Moreover, hardware and software resource on cluster nodes tend to be heterogeneous as underlying machine types evolve over time [57, 36, 16, 16]. Therefore, managing a large number of heterogeneous resources, tracking their liveness, and serving a very large number of tasks on them are the other major challenges in design of a resource manager.
- Sharing and optimization: The need for sharing data and resources to increase return on investment (ROI) is also an important challenge in design of a cluster resource manager. Serving a large number of users (multi-tenancy), hosting a diverse range of frameworks, and addressing contention between jobs with different service level objective (SLO) are different aspects of sharing. A high level of multi-tenancy overburdens the resource manager for enforcing allocation policies to arbitrate contention. In

[1]Throughout the rest of this thesis, by framework we mean a programming model which has specific resource requirements. (e.g. MapReduce, Hive, Dryad, Spark, MPI, and etc.)

addition, diversity of frameworks makes per-job optimization difficult for the scheduler. Also, isolation of a user's jobs to limit the effects of their malfunction on other users' jobs is a challenge for engineers. And finally, serving jobs with different SLO requires specialized scheduling techniques.

Researchers have tried to satisfy these requirements by using an incremental design process. Primary designs like Hadoop [22] mainly focus on creating an embodiment of cluster computing. They have defined the processing model by considering challenges such as software/hardware failures. Moreover, they have facilitated the software development process by embracing simplicity through static partitioning of nodes' resources [73]. Built on the structure of the primary systems, the new generation of cluster management systems focus on increasing scalability and throughput [25]. These systems can be generally divided into two groups. Systems such as YARN [67], Mesos [32], and Corona [26] primarily consider scalability of the cluster management system. They have achieved scalable designs by separation of scheduling concerns. The second group of designs like Omega [60], Apollo [10], and Sparrow [52] principally focus on increasing throughput of the cluster resource managers. They have optimized the resource management process by applying concurrency with optimistic conflict resolution. Optimistic conflict resolution detects and undoes conflicts in resource allocation rather than providing mutual-exclusion of resource for schedulers.

The first group of the new generation takes out the scheduling details from resource management responsibilities and delegate it to different entities. They delegate scheduling responsibilities to fine-grained per-job or per-framework schedulers; while a central resource manager serves resource requests of the schedulers. To serve resource requests, the central resource manager utilizes a polling protocol to monitor and track free resources inside the cluster. These design preferences trade better scalability against higher latency for jobs [25]. As a result, these systems cannot properly serve applications which require low latency and fast turnaround. Another concern in these systems is reliability and availability of services. Utilizing a centralized resource manager raises reliability concerns about a single-point-of-failure. Failure of the resource manager threatens the availability of service in the whole system since all the system's functionalities and the jobs' progress depend on the resource manager.

In the second group, systems like Sparrow and Apollo build a concurrent decision-making approach by distributing multiple schedulers across a cluster's nodes. Utilizing distributed schedulers eliminates single points of failure concern and increases system fault-tolerance. However, using multiple schedulers makes application of sharing properties, such as fairness among users or capacity among frameworks, very difficult. Considering resource and data sharing as a secondary design goal in these system can be sensible for closed-world[2] and single framework[3] environments. However, they are not amenable to an open platform where

[2]Which has coordinated development of the various frameworks that will be respectful of each other
[3]Like Microsoft Apollo [10] which is just designed for Dryad framework [34]

arbitrary frameworks from diverse independent sources share a cluster [67]. While Sparrow and Apollo utilize distributed schedulers, Omega utilizes parallel schedulers inside a centralized cluster manager to increase scheduling throughput. Again, using a centralized entity threatens the reliability and availability of the system and this threat is more significant in Omega's design in comparison with the first group of the secondary systems. This is due to the fact that the cluster manager in Omega is in charge of all resource management responsibilities, including scheduling, which has been delegated to per-job schedulers in the first group. Apart from these limitations, using concurrent schedulers imposes high operation overhead in order to share cluster information among schedulers.

1.3 An Adaptive Distributed/Centralized System

Considering the aforementioned limitations in the existing systems, we have designed and developed an adaptive distributed/centralized cluster resource management system. We have improved average service time, throughput, scalability, and fault-tolerance in our system compared to the existing systems. To build a scalable system, we have separated scheduling and resource management concerns and delegated them to different entities; like the first group of the new generation systems. Moreover, by further breaking up the resource manager to smaller entities we have largely enhanced scalability and fault-tolerance of the system.

To cope with transformation of production workloads from long running batch jobs to short interactive jobs with a high degree of parallelism [52], we have devised a high throughput concurrent resource allocation method. Our system achieves low service time and high throughput scheduling by the means of concurrent decision making. As a result, it utilizes distributed resource allocation in low loads, while switch to the centralized parallel resource allocation in high loads. Similar to the second group of the systems, we have chosen optimistic conflict resolution to simplify decision making. Finally, we have developed components of our system using the actor model of concurrency to achieve a robust system to failures.

In order to build a system with the aforementioned features, we have taken a multi-agent design approach. Our system has four types of agents: resource tracker, resource arbitrator, job manager, and node manager. The resource tracker is responsible for monitoring a cluster and maintaining the cluster view.[4] It tracks each node liveness, performs accounting of resource usage, and serves the resource arbitrator's queries about the cluster's statistics.

The resource arbitrator is a central authority which has two essential responsibilities: a) Serving users' requests such as new job submissions; and b) Allocating resources for tasks based on the resource availability

[4]A logical database of running jobs and their tasks, current share of each users, cluster nodes and their resources, and the relation and mapping among tasks/users/nodes

information it receives from the resource tracker. The resource tracker and the resource arbitrator together provide the same functionality as the resource manager in the previous systems. However, they have more adjustable structures based on the cluster load and the cluster size because of their modular design. Similarly to the other resource management systems, we have per-node agents or node mangers in our system. Each node manager has three responsibilities in our design: a) Controlling and monitoring resources of its corresponding node; b) Serving resource requests from schedulers; and c) Reporting its internal state to the resource tracker.

And finally, the last type of the agents is job manager. In our system, the resource arbitrator allocates an extra container[5] for each submitted job in which it executes a user-developed scheduler. This scheduler is in charge of managing all life-cycle aspects of the job. However, there is a job manager agent for each job to serve the scheduler's resource requirements. The job manger resides in the same container with the scheduler[6] and acts adaptively based on the cluster load.

In low loads, a job manager inquires two randomly selected servers, power of two choices technique [54], to find free resource for a task. Then it submits a container allocation request to the least loaded server for running the task. To adapt this sampling technique with cluster computing requirements, we have devised an adaptive initial sampling rate based on cluster load rather than fixed sampling rate of two. In this technique, the initial sampling rate increases as the the cluster load increases in order to decrease the likelihood of sampling failure. Using adaptive sampling rate in medium to high loads increases the likelihood of finding free resources considerably. However, overhead of this technique increases exponentially in very high loads ($> 90\%$) where jobs compete to access free resources. In this situations, application of allocation policies becomes apparent for controlling the jobs' competition. But, applying allocation policies with distributed resource allocators is a complex problem.

In order to address both the sampling overhead in high loads and the difficulty of applying allocation policies with distributed resource allocator, our system switches to centralized resource allocation in a highly loaded cluster. To keep the system throughput high in the centralized mode, we have designed a hybrid event-driven/periodic heartbeat and a parallel request serving in our system. Using hybrid event-driven/periodic heartbeats, the resource arbitrator is informed about free resources as soon as they become available. Therefore, the resource arbitrator keeps its cluster view of free resources up to date. At the same time, multiple parallel actors serve resource requests by allocating resources based on the view of the free resources. To decrease the rate of conflicts in this parallel process, we have added a level of randomness to the actors' decision making. The network overhead of hybrid event-driven/periodic heartbeats is less than sampling technique since the number of messages with this method is less than the number of required messages to sample nodes.

[5]Containers are system-level virtualization method that can limit a process tree access to computational resources and devices.
[6]Throughout the rest of this thesis, we use job manager and scheduler interchangeably.

To evaluate the presented design, and compare it with the new generation of resource management systems, we designed a simulation based on the configuration of Google's production cluster [57]. Experiments in a simulated cluster of 500 nodes showed that our system provides lower service time compared to a centralized resource manager; from 32% in 70% cluster load to 56% in 90% load. In comparison to a distributed resource manager, our system falls behind the distributed resource manager by average of 12.19ms compared to 10.96ms in 70% load. However, by increasing cluster load our system outperforms the distributed resource manger; up to 49% in 90% cluster load. We also compared the throughput of our system when it receives a burst of job arrival. Similarly, our system outperforms a centralized resource manager in all loads and a distributed system in high loads.

Two other aspects of the systems that we compared are scalability and fault-tolerance. The proposed system has a better level of scalability compared to the centralized resource manager. This is because centralized resource management causes resource under-utilization when the available resources are more than the requested resources while our system does not have this drawbacks. Also, our system is more scalable than the distributed resource manager because of its capability to manage competition among job when free resources are rare. Moreover, the distributed resource manager is not as scalable as our system when number of the jobs increases since it uses constant number of schedulers. Regarding fault-tolerance, our system is responsive to failures in different parts of the system; while the centralized resource manager imposes higher delay and the distributed resource manager does not have any specified mechanism for some of failures.

1.4 Objectives

This thesis has the following objectives toward understanding of and improving resource management in cluster computing:

- Develop a taxonomy to explain the current state of the art in resource management of clusters and survey the mainstream cluster scheduling systems based on the taxonomy.
- Design a high throughput and fault-tolerant distributed resource management system based on adaptive sampling which can serve tasks with low communication overhead.
- Design a scalable cluster monitoring system based on actor clustering and load balancing.
- Devise an adaptive centralized/distributed resource allocation to address sampling shortcomings in a highly loaded clusters.
- Prototype and implement the designed system based on reactive systems manifest and experimentally evaluate it.

1.5 Thesis Structure

The rest of this document is organized as follow: Chapter 2 presents a taxonomy and a short survey of existing cluster scheduling systems. Chapter 3 introduces the architecture of MACRM - the multi-agent cluster resource manager - and describes its implementation details. Chapter 4 presents experiment setup and the experimental results. Finally, conclusion and future work are discussed in Chapter 5.

Chapter 2

Background and Related Works

This chapter begins with an introduction to our assumptions about cluster computing environments. Then, Section 2.2 describes characteristics of a workload in production clusters. In Section 2.3, we provide a taxonomy of cluster scheduling systems. Section 2.4 presents some of the main cluster management systems found on presented taxonomy. Finally, Section 2.5 discusses our motivations for this research.

2.1 Cluster Computing Environment

In cluster computing, a set of connected computers works together so that in many respects they can be viewed as a single system. Unlike grid computers, the computers that make up a cluster cannot be operated independently as separate computers; therefore, each node set to perform the tasks which are controlled and scheduled by the system. Following is a list of key assumptions about a production cluster that mainly focuses on the type and arrangement of resources in clusters.

- *Commodity servers:* Our study targets clusters which employ inexpensive but unreliable servers. These commodity servers are built by cheap and usually regular quality components that are likely to fail. This is the result of organizations tend to achieve cost benefit by moving fault tolerance from hardware to software [29]. As a result, the software should be designed in a way that can tolerate a relatively high rate of component failure [9].

- *A large number of servers:* Clusters with hundreds to thousands of machines currently power the service offered by large Internet companies such as Google, Amazon, Yahoo, Microsoft and Facebook [9, 67]. Furthermore, as cluster computing becomes more accessible with the emergence of cloud computing and public availability of cluster computing frameworks, non-tech organizations also continue to build larger and larger clusters. Cluster size directly impacts the design of cluster scheduling systems for two main reasons. First, the time complexity of scheduling algorithm is a function of the number of machines being scheduled over. Second, as the size of the cluster increases, usually requirements around multi-tenancy starts to take shape which mostly increases the rate of job arrival [38].

- *Heterogeneous resources:* Observations inside Google [57], Yahoo! [36], Facebook [16], Cloudera [16] and more showed that large clusters tend to employ heterogeneous resources. This includes both hardware (e.g. machines with a different number of disks, amount of memory, and number of cores) and software (e.g. machines with a various versions of operating systems). Since the underlying machine types evolve over time with respect to economically attractive prices-performance, several generations of machines with different configurations are likely to be encountered in clusters. Each time that a set of new servers is purchased, the newest generation of hardware is added. This leads to a cross-generation heterogeneity which directly affects the design of clusters' resource managers [38] since jobs may prefer to run on particular types of machines.

- *Commodity network infrastructure:* Choosing network devices and topology involve a trade-off between speed, scale, and cost. Typical data centers use Ethernet network which is organized as a two-level hierarchy [9]. In these networks, bisection bandwidth per node is mush lower than that of out-link from each node which is driven by off-rack communication traffics [38]. However, recent researches significantly reduce these over-subscription ratios and have demonstrated full bisection bandwidth even when all the servers communicate with each other [2]. In the case of a cluster of cloud instances, cloud providers such as Amazon also provide cluster placement groups which provide full bisection 10 gigabit Ethernet bandwidth connectivity between cloud instances in the group [31]. This implies that the previous attempt to achieve data locality such as delay scheduling [71] is going to be irrelevant in cluster computing [3].

- *Per-server power saving:* Traditional power saving methods that turn idle servers off, regardless of their overhead, do not work in current production clusters; this is because servers are utilized as storage systems and high rate of machine failure can cause unavailability of data [5]. Furthermore, cluster-level approach such as covering set [42] (which exploits the replication property of distributed file systems for turning off the servers) are not applicable since they increase jobs' execution times, over-provision some of the servers, and require modification of distributed file system [40]. Consequently, the best set of approaches for controlling power consumption of a cluster are per-server approaches, such as PowerNap [45], which are applied by servers' local operating systems [46].

2.2 Characteristics of Production Workload

This section discusses the different aspects of the production workloads observed in clusters. Recent studies of cluster traces [57, 16, 36] showed that the workloads can vary along the following dimensions:

Service jobs vs. batch jobs. Workloads in studied cluster traces mostly fall into one of the following broad categories:

- Elastic service jobs: These jobs, such as web services, usually run indefinitely and have external client loads. They balance the load among their instances, but the external dependency causes elasticity in their resource consumption. These jobs are concerned with metrics like instantaneous availability, latency, and tail request response time [57], then they require a certain rate of CPU processing to achieve acceptable performance.

- DAG-of-task jobs: This category contains a variety of jobs that run many independent tasks which are CPU-bound or I/O-bound. Traditionally they were used for batch processing of a large amount of data by a coarse-grained splitting of the input among tasks [22, 34]. However, recent approaches [51, 52] tend to interactive large data processing with a larger number of fine-grained tasks. Some of the recent data-processing techniques are Memory-bound since they cache the input data from disk into the memory [73].
- high-performance computing jobs: These jobs often require many machines simultaneously for an extended period to execute their CPU-bound tasks. They are batch queuing systems such gene sequencing or weather modeling that can tolerate substantial wait times [57].

Each of these categories adds different challenges to cluster scheduling systems. For example, Elastic jobs and dag-of-tasks jobs involve frequent cluster scheduler interaction as a result of the changes in their demands while a high-performance computing job often has a fix resource demand during its lifetime. Also, task granularity has a major effect on cluster scheduler. Fine-grained tasks in interactive data parallel jobs and a large number of tasks in HPC-like jobs increase the average number of tasks per job, thus, decrease the number of jobs per time unit that the cluster scheduler should serve.

Task pickiness. Execution specification of tasks may include constraints on resources they require for their execution. Tasks's constraints can be classified into [66]: a) hard constraints: mandatory requirements for execution of a task such as machine architecture or software packages; and b) soft constraints: or preferences that a task may or may not acquire (for example execution on a 8 core machine rather than 4 core one). Founded on this, pickiness is the measurement of how many cluster resources could potentially satisfy a task's constraints [38]. Analyses [57, 16, 36] of cluster traces show that very tiny portion of tasks have hard constraints (for example in Google production cluster only 6% of tasks have hard constraints [57]).

Job and task duration. While some of the service jobs are intended to run forever, the duration of the other jobs vary widely. Job and task duration range from milliseconds to more than a day. Both Facebook and Cloudera workload inspections [16] showed that task duration follows a long-tail distribution with 50% of tasks have a duration less than a minute. Yahoo! workload analysis [36] observed long-tail distribution in job duration with 95% of jobs completed within 20 minutes. The diversity in both job duration and task duration is an important factor in the design of a cluster schedule; since, short jobs are likely to be more negatively impacted by the time they spend waiting to receive the required resources [38].

Task scheduling-time sensitivity. As mentioned earlier, we are facing interactive data parallel applications that require fast turnarounds, such as Hive [64], Dremel [47], and Impala [39] applications. These

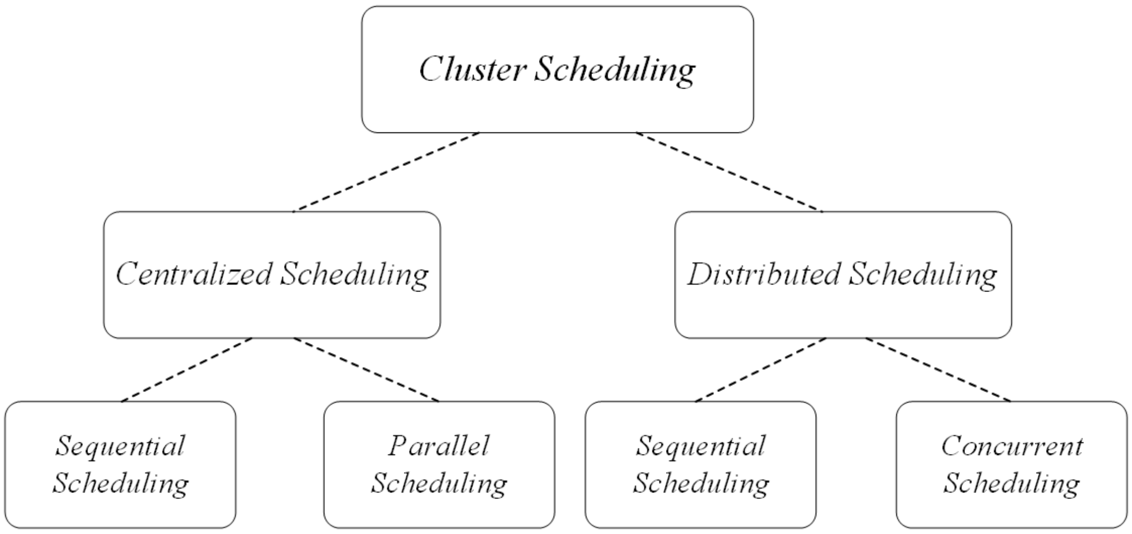


Figure 2.1: Taxonomy of cluster scheduling

jobs have duration as short as a second or even less, then the scheduling delay have a significant impact on user experiences. In the case of service jobs, such as the search engine service of Google or the advertising service of Facebook, sub-second response to resource demands changes is crucial and directly affects company revenue [65].

In this study, we focus on designing a cluster resource manager that serves a mixture of the aforementioned job types. It is worth knowing that many of the mentioned observations in clusters differ from Grid or high-performance computing (HPC) environments. Grid computing is mostly focused on the problem of making virtual organizations on the top of geographically distributed and separately administrated resources in a secure and interoperable way. Also, an HPC environment has the following characteristics: a) Specialized hardware resources, such as expensive high-speed InfiniBand networking [4] and storage devices. b) Jobs are computationally intensive and write-heavy, and tasks are coupled via message passing interfaces. c) Resources across environment are usually homogeneous.

2.3 Taxonomy of Cluster Scheduling Systems

This section presents a taxonomy of cluster scheduling and describing each part of it in details. Figure 2.1 shows the taxonomy of the cluster resource scheduling systems. Although the majority of technical aspects in cluster scheduling systems are widely different; however, a cluster scheduler has following primary responsibilities:

- Tracking resource usage and liveness of a cluster's nodes.
- Enforcing allocation policies and invariants.

- Arbitrating contention among users and their jobs.
- Handling fluctuation in resources that jobs consume.
- Managing the execution flows of jobs.
- Reacting to hardware and software faults.
- Tracking jobs' progress and handling computation skew.
- Applying local and per-job optimizations.

Based on the ways that systems manage these duties, we define each leaf node of the taxonomy tree as follow:

Centralized Scheduler. if all the functionalities are handled by a central decision-making component and the component has exclusive access to all the resources, we call that component a centralized scheduler.

Distributed Scheduler. Compared with a centralized scheduler, decision making in a distributed scheduler is divided among multiple components, and all have access to the cluster's resources. This division can take place in two different ways, either by the delegation of duties or by the replication of the decision maker.

Sequential Centralized Scheduler. In this type of systems, a centralized resource manager makes scheduling decisions for all the jobs one at a time.

Parallel Centralized Scheduler. These systems serve multiple resource requests at a same in a centralized component. Parallel decision makers of the component have access to the cluster's state; but, they do not have access to cluster resource directly.

Sequential Distributed Scheduler. In these systems, there is a meta-scheduler which sequentially servers resource requests of distributed schedulers. Also, the meta-scheduler is in charge of resource tracking, enforcing allocation policies, and arbitration of competition.

Concurrent Distributed Scheduler. These are multi-agent schedulers in which agents compete to access resources. We can further divide these systems to state-based and random schedulers. State-based schedulers require cluster state[1] for scheduling while random schedulers do not require cluster's state.

Since we have established a basic understanding of each kind of cluster scheduling, we can go through existing resource managers for cluster computing environments.

[1]A logical data table in which rows are cluster nodes and columns represent the availability of resources such as cores, memory, network bandwidth, and so on

2.4 Survey of Existing Systems

In this section, we discuss design goals and architectural aspects in some of the cluster scheduling systems based on presented taxonomy. We further explore the details of these systems by talking about their downsides and upsides.

2.4.1 Sequential Centralized Schedulers

Hadoop JobTracker

Apache Hadoop is a cluster management system for large-scale processing of data-sets. Hadoop composed of four main modules: a) Hadoop libraries b) Hadoop Distributed File System (HDFS), c) Hadoop MapReduce programming model [22], and d) Hadoop MapReduce engine. HDFS is designed to be a distributed, scalable, and portable file system which achieves reliability by replicating each data block across multiple nodes (usually three replicas per block) [61]. In HDFS architecture, NameNode is a server that hosts the file system index, and DataNodes are per-node daemons which are responsible for managing data blocks in cluster nodes.

The MapReduce engine comes above HDFS, which is responsible for scheduling user submitted jobs. JobTracker and TaskTracker are two main components of the engine. TaskTrackers are spawned on the nodes and manage execution of submitted tasks. Nodes' resources are divided into fixed size slots, and each task occupies one of these slots for execution. JobTracker is responsible for accepting user submitted jobs and schedule tasks by mapping them to an available slot on the nearest server to the data. As a result, JobTracker usually executes on the master machine where the NameNode is running. In small cluster master node also contains JobTracker and DataNode.

Hadoop default scheduling policy is FIFO (First In First Out), but now it also supports fair scheduling and capacity scheduling. It is obvious that the JobTracker in Hadoop is a sequential centralized scheduler that designed to serve batch MapReduce jobs. As a result, it cannot serve other programming frameworks requirements such as framework specific optimizations or fault management. Furthermore, since it is designed for batch data-parallel processing, it cannot provide low latency scheduling service for latency sensitive jobs.

Spark

Spark [73] is an open source cluster computing architecture and programming model which is complementary to Hadoop programming model. Spark focuses on a class of applications that reuse a working set of data across multiple parallel operations such as machine learning applications on large datasets. By utilizing its special data model and caching policy, Spark achieves from 10x up to 100x faster results than Hadoop for iterative applications. Spark defines resilient distributed dataset (RDD) [72] which is a read-only collection of objects distributed across a set of machines. An RDD can be reconstructed in the case of failure and lost. Spark introduces the concept of in-memory cluster computing, which caches the dataset from disk into memory in order to reduce the latency of I/O operations.

Spark jobs run as an independent set of tasks which are coordinated by a SparkContext object in the driver program (main program). SparkContext can connect to different types of cluster manager or use its standalone cluster manager. Each SparkContext runs a set of processes which are called spark actions. By default, Spark standalone cluster manager uses FIFO ordering to run the submitted applications. Further, Spark actions inside an application (SparkContext instance) are scheduled in a FIFO fashion. In a recent version, Spark provides fair-share scheduling between jobs, which schedules jobs tasks in a round-robin fashion.

The same as Hadoop JobTracker, Spark standalone cluster manager is a sequential centralized scheduler. Spark creator designed it specifically for processing resilient distributed datasets.As a result, it cannot serve other data-parallel programming frameworks, HPC jobs, and service jobs. Furthermore, they provide interactive data-analysis capabilities by decreasing I/O overhead through caching input data in memory rather than high-throughput scheduling.

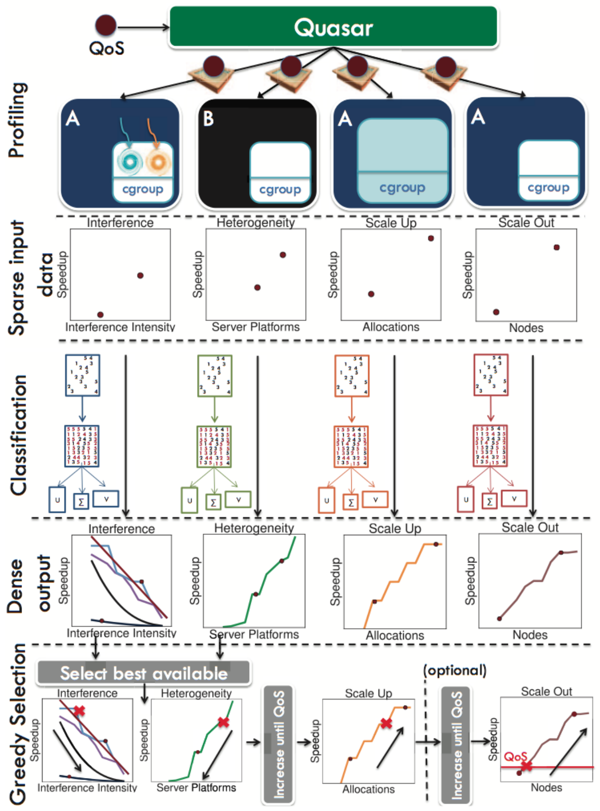


Figure 2.2: Quasar Scheduler [23]

Quasar

The main design goal of Quasar is solving resource underutilization in a cloud and cluster computing environment which is the result of resource reservation policy. To this end, Quasar employs three techniques: a) Instead of asking users the required amount of resources for their computation, the system requires the performance constraints for each workload; based on that, Quasar determines the right amount of resource to meet the constraints. b) To find impacts of amount and type of resources, Quasar uses a set of classification techniques based on profiling of incoming workloads on a few servers. c) Finally, by the means of a greedy algorithm, it combines the result of classification to jointly perform resource allocation and assignment. Furthermore, through monitoring jobs during their lifetime, the system adjusts the allocation and assignment in the case of workload change or classification fault [23].

Quasar employs scale-up, scale-out, heterogeneity, and interference for classification of incoming workloads. In scale-up classification, the system explores how the number of cores, the amount of memory and storage capacity within a server change the workload performance. Scale-out classification only applies to workloads that require multiple processing nodes. This classification investigates the effects of using a different number of nodes on workload performance. Heterogeneity classification estimates the workload performance across all server types by the means of collaborative filtering [63]. Finally, interference classification measures the sensitivity level of a workload to shared resource's interference such as CPU, cache hierarchy, memory, and storage. After classification, a greedy scheduler allocates the least amount of resources needed to satisfy target of a workload's performance by ranking the available servers based on decreasing resource quality (high-performance platforms with minimal interference first) and then selecting from the sorted list. Figure 2.2 shows the structure of Quasar scheduler.

The same as Hadoop JobTracker and Spark, Quasar is also a sequential centralized scheduler. Quasar workload profiling works well for long running jobs with a low level of elasticity in their resource consumption. However, its scheduling delay for short interactive jobs is not viable. Also, its workload profiling does not work for elastic service jobs. Quasar tends to serve various jobs from different programming frameworks, but using a centralized scheduler with profiling overhead prevents it to be scalable and high-throughput.

2.4.2 Parallel Centralized Schedulers

A genuine solution for high-latency in sequential schedulers is parallel scheduling. We can divide parallel centralized scheduling into partitioned state and shared state scheduling. Partition state scheduling can be further divided into static partitioning and dynamic partitioning. Usually, static partitioning is based on cluster administration policy while dynamic partitioning depends on cluster condition and demands in each scheduler. Partitioning is not a proper solution for presented cluster environments (Section 2.1) and cluster workloads (Section 2.2). This is due to the following limitation [38] : I) Fragmentation of resources: this is rooted in heterogeneous resource demands and heterogeneous resources in cluster which can cause starvation for jobs with large resource demands and resource underutilization; II) Restriction of goodness-of-fit especially in case of accessing data since the scheduling domains must be selected before the scheduler performs task-resource assignments. Consequently, we do not present partition state schedulers like Quincy [35] in this survey.

Omega

To achieve a high-throughput scheduling, Omega provides parallel scheduling architecture which is built around shared state and use of lock-free optimistic concurrency control. Omega grants each scheduler a full access to the entire cluster state [60]. To mediate decision conflicts which are results of concurrent scheduling, Omega utilizes optimistic concurrency control. In Omega, there is a common cluster state maintained by a meta-agent which is a resilient master copy of cluster resources state. Each of the parallel schedulers synchronizes its cluster state with this master copy before making a scheduling decision. After synchronization, the scheduler takes the first job from the shared queue and tries to match its tasks to the available resources. When the scheduler completes the task-resource assignments, it attempts to commit a transaction from its cluster state to the master copy. In this stage, it is the responsibility of meta-agent to detect conflicts among transactions and regulates them based on the meta-scheduling policies. Consequently, if the meta-agent rejects a transaction, the scheduler tries to reschedule the tasks.

To manage the conflicted transactions, Omega employs either a coarse-grained conflict detection or all-or-nothing scheduling. In the coarse-grained conflict detection, a scheduling transaction would be rejected if a change has happened to any of the selected machines in the transaction since the private cluster state synchronization. This is simple to implement by sequence numbering of the machines' states. In all-or-nothing scheduling, which is designed to support jobs with gang-scheduling requirements, an entire transaction would be rejected if it causes over-commit in any machine. All-or-nothing helps to avoid resource hoarding for gang-scheduling which causes resources underutilization.

Omega is capable of scheduling a different kind of jobs and achieving high-throughput scheduling through parallelism. However, it has following disadvantages: a) Since multiple schedulers share the same view of the cluster state, they tends to schedule tasks to the same lightly loaded server. Consequently, as the number of schedulers increases, the probability of conflict increases, thus scheduling overhead and latency increases [10]. b) Parallel scheduling in Omega makes enforcement of global properties such as capacity/fairness very hard. A system without these global properties is practical for a closed-world environment where there is a coordinated development of various frameworks, but it is not practical for an open environment with independent users [67]. c) The same as the other centralized approaches, Omega is subjected to the single

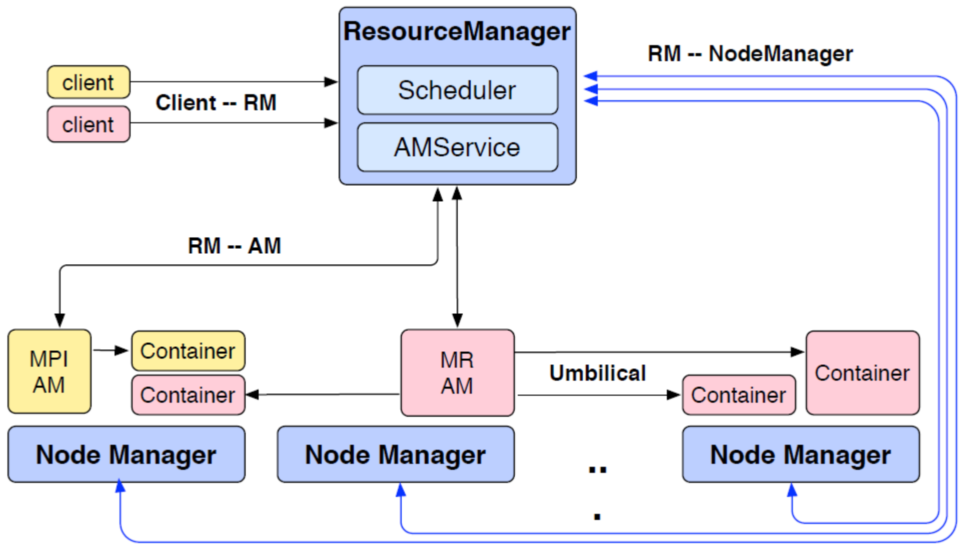


Figure 2.3: YARN Architecture. Blue parts are system components. Yellow and pink are two running applications [67]

point of failure issue.

2.4.3 Sequential Distributed Schedulers

Scalability and fault tolerance are two major problems that all centralize schedulers should cope with. This is even worse in a cluster computing environment since there is a high rate of failure (Section 2.1) and a large number of resources and computations (Section 2.2). Regarding these facts, managing all scheduler responsibilities that are mentioned in Section 2.3 by the means of a monolithic agent appears to be a barrier to scalability, availability, fault-tolerance, and resource sharing. As a result, designers and engineers tend to divide the scheduling responsibilities and delegate parts to the different agents [67, 26, 32]. Usually in these systems, there are per-job or per-framework schedulers and there is a resource manager which serves resource request of the schedulers. Although parts of scheduling happen concurrently in this design, we call these systems sequential since all the activities depend on a sequential resource manager.

YARN

For a large number of companies, Hadoop is the de-facto place to share and access data and computing resources. This high number of usage and wild adoption pushed Hadoop initial design beyond its intended target. During years of employment, several new requirements emerged for using cluster resources and processing shared data that were not regarded in Hadoop design. To address these requirements, the next generation of Hadoop computing platform emerged. YARN has been developed to extend Hadoop in different directions such as scalability, multi-tenancy, serviceability, and reliability/availability [67]. YARN decouples the programming model and scheduling functionalities (such as fault-tolerance) from resource management concerns. This design delegates the sooner to per-job agents while there is a resource manager (RM) per-cluster in charge of the later. RM is a central authority which arbitrates resource competitions among applications. RM has a global view of the cluster resources and enforces properties such as fairness, capacity and locality. Complementary to RM, there is a node manager (NM) on each worker machine which manages and monitors containers on that machine.

For each submitted job, RM provides a container which is a place for application master (AM) of the job. AM is responsible for coordinating the execution of the job's tasks in the cluster. AM manages all lifecycle aspects of the job such as applying optimizations, tracking progress, handling resource demand fluctuation and reacting to failures. AM harnesses the required resources to complete the job by issuing resource requests to the RM. These resource requests are submitted along with periodic heartbeats used to report liveness of AM to RM. YARN's resource requests can be formatted to project containers with special properties such as data locality. After receiving resource requests, RM tries to satisfy them based on resource availability and scheduling policies which can be fair-share, FIFO, or capacity scheduling. Once the AM discovers that requested containers are available, it can manage the execution of tasks inside those containers. If needed, the AM can communicate directly with tasks executing inside the containers, but it should be specified in the application.

Corona [26], Facebook cluster manager, is another system that uses two-level scheduling like YARN. The same as YARN, Corona has a central resource manager and dedicated per-job JobTrackers which are in charge of tracking and monitoring jobs' progress. Compared to YARN, which utilizes heartbeat based control plane framework, Corona uses push-based communication approach. This design decision trades off latency against scalability and fault-tolerance which is non-trivial [67]. Figure 2.3 shows the architecture of YARN and how different entities communicate with each other in this architecture.

Mesos

YARN and Corona are designed to separate resource management from scheduling, and delegate scheduling to per-job agents. Mesos designers did the same separation; however, Mesos employs per-framework schedulers instead of per-job schedulers. As a result, Mesos is a meta-scheduler for framework schedulers who try to share data and resources in cluster computing environment. In Mesos, there is a master process that manages per-node slave daemons, while there are frameworks' schedulers that ask for running tasks on the slaves.

Mesos uses a novel resource offer mechanism instead of resource request mechanism that is employed in YARN and Corona. Each resource offer contains a list of available resources on slave machines. The master is in charge of making decisions according to the administration policies (such as fair sharing or priority) about the amount of resources to offer to each framework. Since the whole system depends on the master process, there should be fault-tolerance mechanisms to preserve the state of this entity. As a result, Mesos master has been designed to be a soft state so that in the case of failure, a new master from the pool of hot-standby masters can completely reconstruct the failed master. To this end, Mesos utilizes ZooKeeper [28].

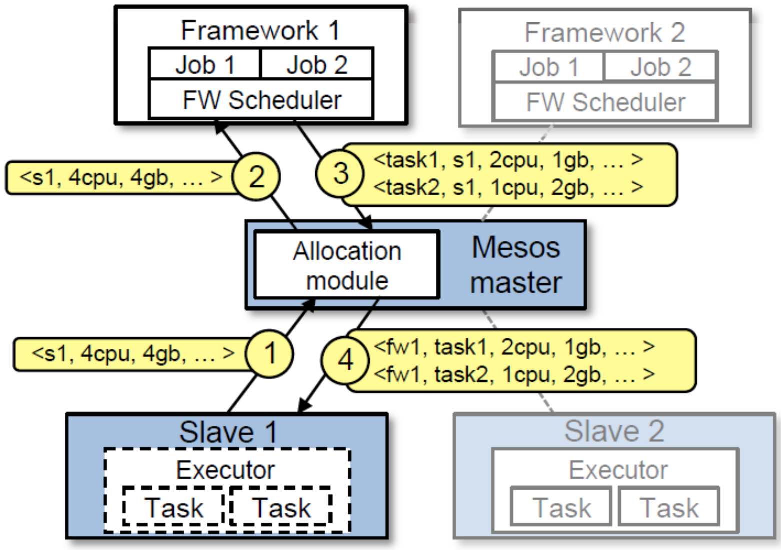


Figure 2.4: Resource offer example [38]

Figure 2.4 shows resource offer mechanism in Mesos. While the master decides how many resources offer to each framework, it is the responsibility of frameworks schedulers to choose among the offered resources for its internal usage. Therefore, the resource allocation mechanism contains following steps: a) A slave reports the master that it has free resources. b) Master selects one of the framework schedulers based on cluster internal policies to offer this resources. c) Master sends the description of resources as an offer to the selected scheduler. d) The selected scheduler responds to the master offer according to its requirements. e) Master informs the slave about the required allocation, which in turn launches the framework's tasks. Mesos's master does not require frameworks' resource requirements or constraints; therefore, frameworks can evolve independently from the master. Instead, frameworks are allowed to reject resource offers that do not match their constraints and wait for the future offers. To further improve the efficiency of resource offer mechanism, there is also a filtering mechanism in Mesos, which specifies certain resource that framework will always reject.

2.4.4 Concurrent Distributed Schedulers

Aforementioned systems with distributed schedulers achieve scalability, fault-tolerance, and high availability compared with centralized scheduling. However, their sequential resource managers impose limitations on their extensive usage. These systems are optimized for long-lasting jobs with complex resource constraints; therefore, they do not have proper service time or throughput for scheduling sub-second tasks [52]. Furthermore, their scheduling is suboptimal since neither the resource manager nor the schedulers have a complete view of tasks' characteristics and the cluster state respectively [24]. Consequently, concurrent distributed cluster schedulers designed to overcome these limitations.

Sparrow

Sparrow is a distributed cluster scheduling system which can schedule tasks with low latency and high throughput. Scheduling highly parallel jobs with short tasks, around hundreds of milliseconds, presents a difficult challenge for centralized or sequential resource managers. Sparrow addresses this challenge by exploring stateless concurrent schedulers that utilize the power of two choices technique [52]. The power of two choices is a load balancing technique which schedules each task by probing two randomly selected servers and placing the task on the server with a smaller queue. Task placement with this technique improves expected wait time exponentially compare with simple random placement [49]. Sparrow considers a cluster as a combination of worker machines that run tasks in a fixed number of slots and schedulers that assign tasks to those slots. Schedulers make decision concurrently and independently; also, each of them can schedule a submitted job.

To make the power of two choices more efficient for cluster computing environment, Sparrow introduces two techniques: batch sampling and late binding. Since parallel jobs are sensitive to tail task waiting time, completion of a job depends on the last task completion, Sparrow tries to address this concern by the means of batch sampling. In batch sampling, instead of performing sampling for each task independently, schedulers place the m tasks of a job on the least loaded set of $d.m$ randomly probed machines. Also, there are two other performance issues with the power of two choices: a) The queue length is not a good indicator of expected waiting time. b) Messaging delay causes race condition among multiple schedulers that sample concurrently. Sparrow addresses these two issues by applying late binding. In late binding, schedulers delay task assignment to the slots until the workers machines have free resources to run them. As a result, the response time of a job decreases compared to the power of two choices with batch sampling.

Sparrow is a primary prototype design to investigate the possibility of the power of two choices scheduling in cluster computing environment. As a result, it has several shortcomings that hinder its usage in production clusters. These limitations are: a) Static partitioning of resources and slot-based resource allocation which lead to an improper serving of resource demands. b) Solid distributed design which precludes Sparrow

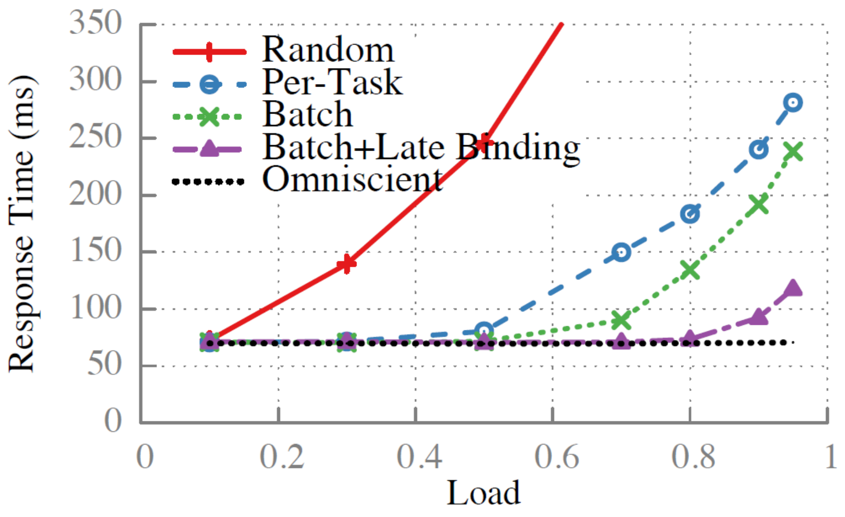


Figure 2.5: Comparison of scheduling techniques in a simulated cluster of 10,000 4-core machines running jobs with 100-tasks [52]

from applying global scheduling policies such as fair-share. c) Lack of a mechanism for handling worker failures which are very likely in cluster computing environment. d) No internal mechanism to address the exponential growth of scheduling latency when the cluster load rises to 80% and above (Figure 2.5). Moreover, e) Incapability of handling complex job's constraints since Sparrow's schedulers do not have cluster's resource view.

Apollo

As clusters' sizes grow continuously, it becomes more challenging to balance scalability and scheduling quality of resource managers. Apollo's designers explored distributed and loosely coordinated scheduling to address this challenge. Utilization of distributed schedulers makes Apollo a scalable resource management systems. These schedulers make decisions in an opportunistic and coordinated manner by using synchronized cluster information. Opportunistic scheduling increases cluster utilization and decreases job latency.

To implement opportunistic scheduling, Apollo introduces regular tasks and opportunistic tasks. Apollo provides low latency for regular tasks while trying to increase utilization by scheduling opportunistic tasks in the slack left by regular class [10]. To limit the number of regular tasks and ensure fairness, Apollo employs a token-based mechanism. Each token is a right to execute a regular task which consumes up to a limited amount of memory and CPU. Since opportunistic tasks are subject to starvation, these tasks can be upgraded to regular class after being assigned a token. To synchronize cluster information, Apollo presents server load and local queues advertisement mechanism. This mechanism provides a near-future view of available resources on each node for schedulers.

Apollo schedules each task on a server which minimizes the task's duration to achieve high-quality. To this end, Apollo introduces an estimation model based on completion time and the probability of failure. To estimate the task completion time when there is no failures, it considers following times: a) task initialization time for fetching the required files, b) task wait time which comes from a lookup in the wait-time matrix of the host server, and c) task runtime which consists bot I/O time and CPU time . To calculate probability of task failure, Apollo utilized an empirically determined value and upcoming and past server maintenance schedule. This model allows schedulers to perform weighted decision rather than just considering task placement requirements such as data locality. To manage unexpected cluster changes and suboptimal estimations, Apollo employs series of correction mechanisms to dynamically adjust previous decisions.

Providing cluster information for schedulers addresses Sparrow's limitations in handling worker failures and serving jobs with complex constraints. However, it limits system scalability by imposing a substantial overhead to synchronize schedulers. Designers of Apollo ignored this fact since they just targeted SCOPE [74] framework in their design. Since SCOPE applications do not have elastic resource demand similar to web services, there is no need for continuous resource allocation which requires resynchronization. Furthermore, schedulers are subject to conflicts, especially in a highly loaded cluster. This is because schedulers tend to schedule tasks to the same lightly loaded servers because they share similar cluster state. To summarize the presented cluster management systems, Table 2.1 shows and abstract view of the systems.

Table 2.1: The summary of the presented systems

Name	**Centralized/Distributed**	**Concurrent/Sequential**	**Frameworks**
Hadoop	Centralized	Sequential	MapReduce, Hive, Giraph [6]
Spark	Centralized	Sequential	Spark
Quasar	Centralized	Sequential	No restriction
Omega	Centralized	Concurrent	No restriction
YARN	Distributed	Sequential	No restriction
Corona	Distributed	Sequential	MapReduce, Hive
Mesos	Distributed	Sequential	No restriction
Sparrow	Distributed	Concurrent	No restriction
Apollo	Distributed	Concurrent	Scope

2.5 From Map-Reduce to MACRM

This section presents recent evolutions in cluster computing environment which motivate us to design a new cluster resource manager. Section 2.5.1 describes the advances in network technology which remove the disk locality requirement in task scheduling. Section 2.5.2 presents the scalability issues in existing systems which drive us to design a more modular resource management system.

2.5.1 Disk-locality Is Not a Scheduling Concern Anymore

Although cluster computing history goes back to the 1970s when Unix operating system and TCP/IP network protocols were shaped, it started to become the mainstream approach for supercomputing in early 2000s [48]. This is a time when Google revealed details of its cluster computing facility which supports Google search service. The first paper was released in 2003 which discussed the Google internal cluster architecture, and the second one in 2004 that presented their computing model for processing big data.

In the first paper, they showed how a large facility with 15,000 commodity-class PCs and network switches to support their search engine computation were built [8]. The second paper discussed Google's Map-Reduce programming model which automatically parallelizes and executes programs on a large cluster of commodity machines [22]. Map-Reduce is used for processing and generating large data sets by utilizing services from Google File System (GFS) [30] which is a distributed file system that divides each file into 64 MB blocks and stores three copies of each block on different servers.

Google's cost-effective computing platform and simple programming model attracted significant attention to cluster computing and big data analysis. Since then, several other programming models have been developed based on the Map-Reduce[2] programming model idea, such as Spark [73] and Dryad [15]. Although they are programming models, they are usually associated with cluster management and scheduling systems that can satisfy their processing requirements.

A common feature among all of the proposed scheduling systems is data-locality. Since Map-Reduce designers assumed that network bandwidth is a relatively scarce resource in the cluster computing environment, they attempted to schedule tasks on a machine which has a replica of the input data [22]. This was a fair assumption since Google used a tree-like network structure to connect the servers to each other. The servers were grouped into the racks consisting of 40+ machines and were interconnected via a 100-Mbps Ethernet switches.[3] Then all the racks were connected by one or two gigabit uplinks to a core gigabit switch. In this network topology, bisection bandwidth per node is much lower than that of the out link from each node. Consequently, achieving disk locality is a major factor in the design of cluster scheduler.

However, recent advances in network devices and network topology design has changed the conditions.

[2]From now to the end of this document, by Map-Reduce we mean Map-Reduce programming model and its open-source implementation Hadoop 1.x

[3]We use the term switch to refer to both layer two switching and layer three routing

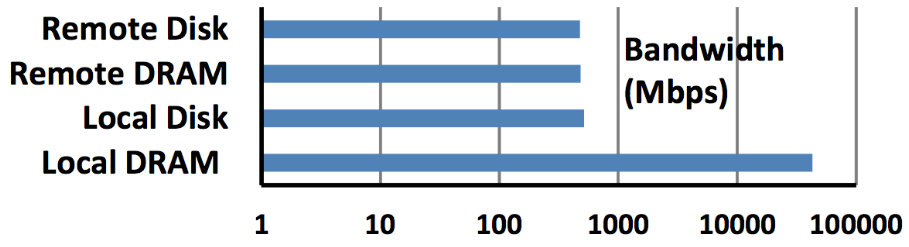


Figure 2.6: Comparison of disk and memory read bandwidths locally and across a network [3]

Currently, commercially available switches can support the aggregate link speed of 100Gbps, and server Network Interface Cards (NIC) can provide 10 to 25Gbps rate. By utilizing these new network devices, read bandwidth from a local disk is just about 8% faster (Figure 2.6) compared to the read from local network in a tree network topology [3].

Although utilizing new network devices has extended data-locality level from disk local to rack local, off-rack bandwidth oversubscription [22] still drives motivations for data-local scheduling in clusters with tree network topology. Oversubscription is a means of lowering the total cost of the design for a data center. An oversubscription of 1:1 means that all hosts can communicate with an arbitrary host at the full bandwidth which is their network interface bandwidth. Typical data center designs had oversubscription value of 2.5:1 which provided bisectional bandwidth of 400Mbps with 1Gbps network interfaces [2].

To decrease the oversubscription value from 2.5:1 to 1:1 with commodity network switches, Al-Fares et al. [2] developed a fat-tree network topology which is a special instance of a Clos network topology [18] (Figure 2.7 shows their network organization). In a k-ary fat-tree topology, there are k pods, and each has two layers of $\frac{k}{2}$ switches. $\frac{k}{2}$ of each k-port switch are connected to $\frac{k}{2}$ of hosts, and the remaining ports are connected to k ports in the aggregation layer of the tree. By utilizing commodity network switch and fat-tree topology, they achieved 1:1 oversubscription value with much lower cost compared to the tree topology with 2.5:1 oversubscription value. As a result, their method has been adopted by new data centers and is currently into use [13].

Rather than bandwidth oversubscription, another assumption that calls for disk locality is disk I/O constituting a considerable fraction of task's lifetime. Although this is a case in general Map-Reduce computation, two movements in data-parallel computing invalidate this assumption. The first movement is the development of different programming models like Pregel [44] and Dryad [34] for iterative data-parallel computations such as graph processing and machine learning applications. In these models, instead of using multiple two-level Map-Reduce jobs to imitate iterations, they create a graph of tasks which communicate through message passing. Consequently, the intermediate results between iterations or graph vertices are streamed on the

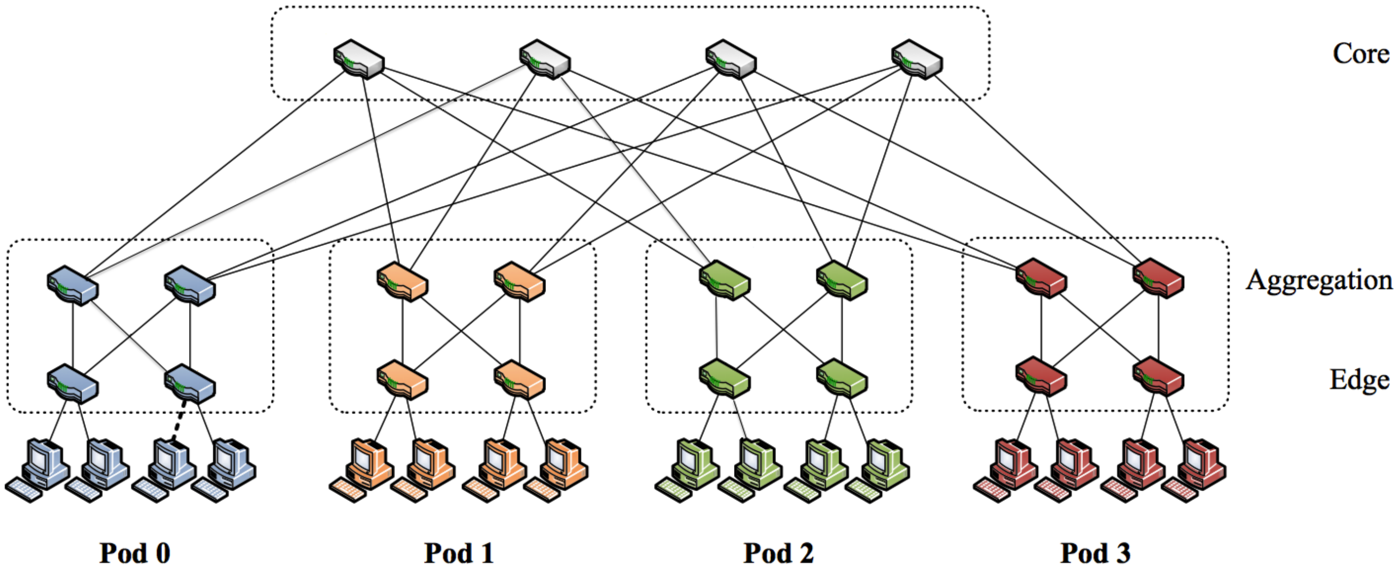


Figure 2.7: A fat-tree network topology [2].

network rather than I/O from disks.

The second movement is the utilization of tiny tasks for data-parallel computations. Using tiny tasks paradigm can mitigate stragglers and increase cluster utilization without sacrificing job response time and fairness [51]. In this paradigm, jobs are broken into tiny tasks operating on a small amount of input data and are completed in hundreds of milliseconds. This is a pattern for a large portion of Map-Reduce jobs; for example, statistics of Facebook production Map-Reduce cluster shows that on average 96% of jobs has less than 7MB input and output data while each job has more than ten tasks [16].

2.5.2 Does Not Scale

Although aforementioned changes reduce the need for disk locality and decrease the cluster scheduling complexity, there are still several shortcomings in primarily Map-Reduce design. By early 2011, Map-Reduce started reaching its limitation in large-scale production clusters such as Facebook's data warehouse facilities [26] and Yahoo!'s infrastructures driving its WebMap[4] application. The issues originate from the scheduling framework which consists of a Job Tracker and per-node Task Trackers.

Job Tracker's primary responsibilities are: a) Cluster resource management and monitoring. b) Scheduling submitted jobs of all users. Task Trackers are in charge of running the tasks that are assigned by the Job Tracker. Scalability limitations of this design become evident when the number of jobs increases and clusters

[4]A technology that powers Yahoo!'s search engine for building a graph of the known web [20]

grow up to nearly 4K nodes. As a result, Job Tracker cannot handle its responsibilities adequately and imposes a large scheduling overhead which results in an obvious drop in the cluster utilization. Another limitation of Map-Reduce framework is its resource-centric model which is not compatible with other application models. While Map-Reduce supports a broad range of use cases, it is not the proper choice for large-scale iterative computations and bulk-synchronous parallel models with message passing. Such tight coupling of the programming model with resource management pushes users to abuse the programming model by writing applications that break the built-in assumptions of Map-Reduce. As a result, this leads to poor utilization, potential deadlocks, and instability [67] since Map-Reduce's scheduler tries to thwart the assumptions.

Consequently, studies [67, 26, 32] have suggested separation of Job Tracker responsibilities to address scalability and flexibility issues. The common design pattern among all the suggested solutions is using a centralized resource manager for tracking resource usage, monitoring node liveness, enforcing allocation invariants, and arbitrating contention among users. This central entity deals with abstract descriptions of user requirements for resource allocation and ignores semantics of each allocation. The task of controlling the semantics of allocation and coordinating the logical plan of jobs are delegated to distributed schedulers. There can be per-job, per-framework, or per-user schedulers based on the system's design. These schedulers acquire resources by requesting from the resource manager or accepting the resource manager's offers. Then, they generate a plan for acquired resources and coordinates the execution of plan around possible failures.

The design with per-user schedulers limits the effects of a malfunctioning job to the job's owner. However, since these schedulers cannot be optimized for all kind of applications, they cannot properly manage lifecycle aspects of all possible jobs, unless, each user is inclined to submit same jobs with similar resource requirements, optimization strategy, and failure recovery policies. Using per-framework scheduler can provide better control over an application's lifecycle aspects since applications that are developed within the same programming model usually have similar features. However, a buggy application from a user can crash the schedulers and affect the other users and their applications. Therefore, design with per-framework schedulers hampers system multi-tenancy requirements. Furthermore, per-framework schedulers limit the developers to use specific programming frameworks and prevent them from applying specific optimizations or planning strategies for their jobs. Design with per-job schedulers addresses issues in per-user and per-framework schedulers by undertaking overhead of allocating a scheduler for each job. The fine-grained per-job scheduler provides flexibility, simplicity, and fault isolation for applications and their owners. Still, all the schedulers depend on the central resource manager which intensifies single point of failure concerns.

Moreover, all of these designs will eventually encounter scalability problems at different levels. Growth in the number of submitted jobs will result in an exponential increase in the number of tasks because of highly parallel jobs like in-memory spark queries. With per-framework or per-user schedulers, this exponential growth in the number of tasks that a scheduler should serve will repeat Map-Reduce scalability problems in schedulers level. In designs with per-job schedulers, growth in the number of tasks per job further justifies resource overhead of a scheduler for each job. However, this increase causes scalability issue at the resource manager level because it should server resource request with very high throughput. In the next chapter, we present architecture of an adaptive distributed/centralized resource management systems which address aforementioned issues in existing resource manager.

Chapter 3

System Design and Implementation

This chapter presents details of the design and implementation of the Multi-Agent Cluster Resource Management System (MACRM). Section 3.1 presents details of MACRM's design which considers recent advances in cluster computing and limitations of the existing resource management systems. Then, Section 3.2 describes implementation aspects of MACRM and how it has been developed based on the presented design.

3.1 MACRM Architecture

In MACRM design, we focus to address the shortcomings (Section 2.5.1 and Section 2.5.2) of the existing resource management systems. MACRM design concentrates on scalability, fault-tolerance, and average service time aspects of resource management. Considering each of these aspects separately, we had several choices for improvement. For example, to address single point of failure problem of a centralized resource manager, we could break the resource manager up to smaller components which are less expensive to recover, or replicate the resource manager and replace it with a replica in a case of failure, or create multiple instances of the resource manager which work concurrently. However, these aspects are connected; therefore, improving each of them imposes limitations on the possible choices for the two other aspects.

To come up with a comprehensive cluster resource manager with a satisfactory scalability, fault-tolerance, and service time, we revised possible choices based on our preliminary investigations. In other words, we considered viable solutions for improving each of the system's aspects in the beginning, then tried to chose the most proper ones based on our primary results. The next five subsections present our journey to find the best solutions. This process led to an adaptive distributed/centralized resource management system which is reactive to cluster load and has a scalable and modular structure.

3.1.1 Let's Divide It Again

In the first open-source implementation of Map-Reduce, there is a centralized job tracker which is in charge of all scheduling and resource tracking responsibilities. The job tracker tracks the cluster's resources using heartbeats, which are periodic messages from the cluster's nodes to the job tracker. Heartbeat is also a mechanism to monitor liveness of the cluster's nodes. In this design, the job tracker can only process 100 heartbeats per second, assuming each RPC is processed in 10 milliseconds [1]. As a result, in a cluster with 2000 nodes, each node can send a heartbeat only every 20 seconds. To address this scalability issue, job tracker's responsibilities are delegated to a resource manager and per-job schedulers in the second version of Map-Reduce. As a result, nodes can send heartbeats at shorter intervals. MACRM has a similar design to the second version of Map-Reduce. There is a centralized resource manager which tracks a cluster's resources and makes decisions about their allocation. Moreover, the resource manager serves resource requests of per-job schedulers. Per-job schedulers are responsible for coordinating execution of the jobs' tasks and manage lifecycle aspects of the jobs such as applying optimizations, tracking job progress, handling resource requirments fluctuation, and reacting to failures.

The centralized design of the resource manager and heartbeat mechanism for tracking cluster's nodes still causes scalability problem. It is evident that as the cluster size grows, the number of heartbeats that are received by the resource manager increases. Moreover, as the number of jobs rises, the number of resource requests from per-job schedulers increases. Therefore, this single resource manager becomes a bottleneck, limiting scalability. A simple solution for this scalability issue can be adaptively reducing the number of heartbeats in high loads by increasing heartbeat's intervals. However, it would cause high latency for jobs requiring available resources for their tasks' execution. This is because the resource manager serves the resource requests based on the resource availability information it receives through heartbeats. This happens as a result of using heartbeats for both cluster monitoring and resource allocation. To track the cluster state and monitor nodes, the resource manager should receive heartbeats from all the nodes. However, to allocate resources, the resource manager only needs to find nodes with free resources.

The design with a centralized resource manager also suffers from a single point of failure. Since most of the system progress depend on the functionality of the resource manager, failure of this entity will be catastrophic for the users and their applications. This problem can be addressed by replication of the resource manager using ZooKepper [28]. ZooKepper reconstructs the internal state of the resource manager in replicas and in the case of a resource manger's failures it replaces one of the replicas with it. Replicating the internal state of a resource manager has overhead which can be decreased by ignoring some of the unessential information. Suppose that the resource manager is an agent and its internal state changes based on the messages received from other entities. The resource manager has the following interfaces to communicate with the other entities: a) User Interface: to accept job submission from users and to provide controlling information to the users. b) Scheduler Interface: to receive resource requests from the schedulers. c) Node Interface: to receive heartbeats from cluster nodes and track their liveness. The communication pattern of the first two interfaces are event-driven, so their information needs to be persisted because the information cannot be

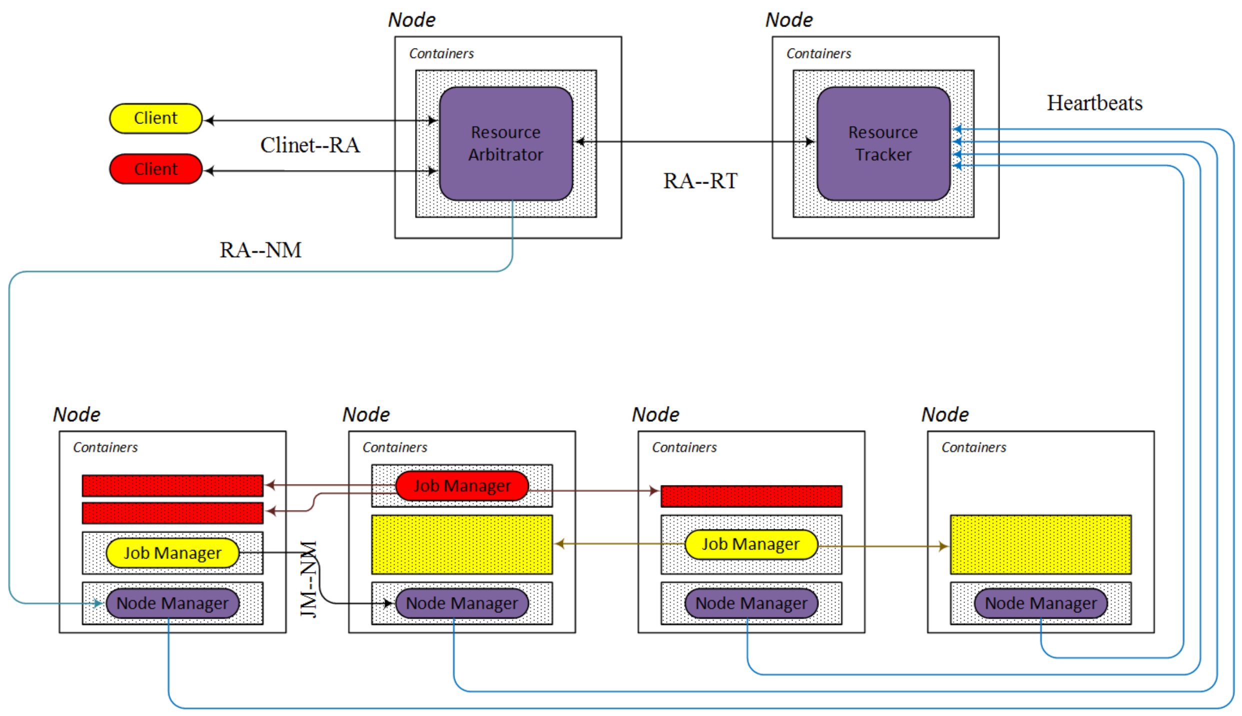


Figure 3.1: MACRM's system components and their communications

reconstructed and do not expire (hard-state information [17]). The communication pattern of the last interface is based on periodic heartbeats, which means the nodes' states can be reconstructed through the future heartbeats [43] (soft-state information [17]). Therefore, Zookeeper can ignore nodes' state in creating replicas.

Two aforementioned scalability problems in a centralized resource manager are related. For resource tracking, the resource manager needs to receive periodic heartbeats from cluster's nodes (soft-state information). While for resource allocation, it requires information about the submitted jobs and the resource requests from schedulers (hard-state information). As a result, to address the scalability problems and to decrease failure recovery overhead, MACRM breaks up the resource manager into two separate entities (Figure 3.1 shows MACRM's system components). Particularly, MACRM has a resource tracker agent and a resource arbitrator agent instead of a resource manager agent. These agents are described blow.

The resource tracker is responsible for monitoring a cluster by creating and preserving a complete view of the cluster. To this end, it receives periodic heartbeats through its node interface and updates the cluster view based on them. Afterward, the resource tracker filters the heartbeats from the nodes with free resources and forwards them to the resource arbitrator. There can be multiple instances of the resource tracker when the cluster size is extremely large (Section 3.2.3). This eliminates the resource manager's scalability issues when the number of nodes increases. Moreover, each resource tracker's view of the cluster is soft-state since it can easily be reconstructed from scratch by receiving a round of heartbeats. Therefore, they have tiny failure recovery overhead because a new resource tracker can immediately be replaced by the failed one.

The resource arbitrator is in charge of resource allocation. As shown in Figure 3.1 it has three interfaces to users, schedulers, and the resource tracker. On the first interface, it accepts job submission from users and provides information about jobs' status to them. Through the second interface, per-job schedulers can submit their resource requests to the resource arbitrator. And finally, the resource arbitrator receives information about nodes with free resources and each user's share of the cluster on the third interface. Furthermore, the resource arbitrator queries the resource tracker for the state of a particular job to serve users' monitoring requests.

Separation of resource tracking from resource allocation can provide a more scalable and modular system in which the resource arbitrator has a smaller and simpler internal state. Therefore, its internal state can be replicated and recovered faster compared with the resource manager's internal state. To recover resource arbitrator from a failure we use ZooKeeper [28] which preserves consistency of the internal state. However, this new design further aggravates scheduling delay of the centralized systems. In the next section, we investigate shortcomings of heartbeat based resource allocation and present MACRM's solutions.

3.1.2 Low Scheduling Latency in MACRM

Different resource monitoring modes have been proposed in distributed system's research [69] which can be categorized into four classes: periodic push mode, periodic pull mode, event-driven push mode and event-driven pull mode. Push mode refers to the case where every node in a distributed system actively reports its status to the monitor. Pull mode refers to the condition where the monitor actively queries a node's state. Periodic or event-driven aspects of the monitoring are determined by the timing pattern of monitoring occurrence [69].

MACRM adapts push mode rather than pull mode to scale up to clusters with thousands of nodes. This is because pull mode resource monitoring requires the double number of message transmission to acquire status of the same number of nodes compared to push mode. Furthermore, MACRM utilizes periodic monitoring rather than event-driven monitoring in order to discover node failure faster. Although, periodic push based monitoring causes scheduling delay which increases response time of applications and decreases resource utilization, this delay is inconsequential relative to long-running batch processes like Map-Reduce jobs [25].

However, scheduling delay associated with periodic push based monitoring approach or heartbeat is a substantial concern for interactive jobs like in-memory Spark queries. These jobs must be served with low latency and high scheduling delay is an intolerable overhead for them: for 100 milliseconds tasks, scheduling delay above tens of milliseconds is unacceptable [52]. Currently, a large portion of jobs in production clusters

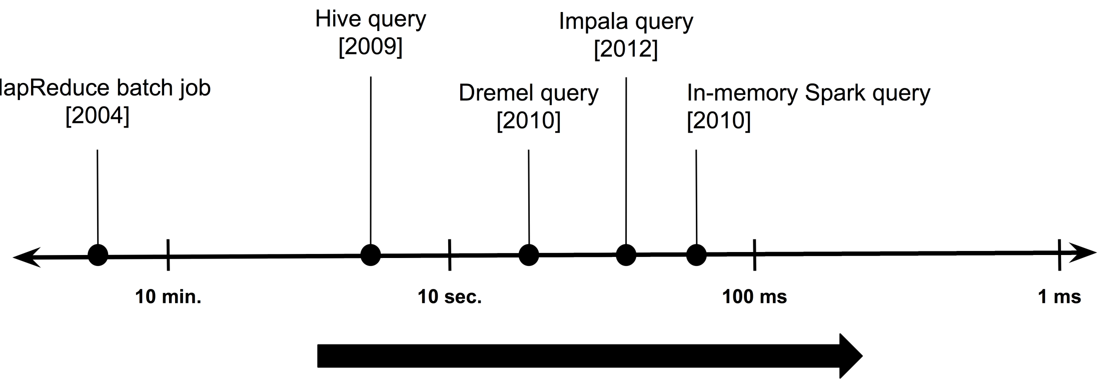


Figure 3.2: Average execution time for some of the data analytics frameworks.

are short interactive queries such as Dremel, Impala, and Spark (Figure 3.2). These jobs are ad-hoc queries for data analysis and data mining. For example, in 2012, Facebook clusters served more than 60K queries per day, which is more than 80% of their daily workload [26]. This workload distribution is broadly similar to what can be observed in cluster traces from other internet companies like Google [57] and Yahoo [36].

There are also two other timing concerns that become apparent in a centralize sequential scheduler. The first one is queuing delay to apply resource allocation policies. Cluster managers apply allocation policies, such as fair-share or capacity scheduling, to arbitrate contention when aggregate demand for resources exceeds computing capability. However, in low loads, this process imposes useless overhead and increase service time.

Another timing issue that is substantial in sequential scheduling is head-of-line-blocking wherein a large and high-priority job with complex scheduling constraint may cause scheduling delay for all subsequent jobs [38]. For example, scheduling a task with several hard-constraints involves considering multiple independent groups of machines that may nest and overlap, which is an NP-complete constraint satisfaction problem. The same as the scheduling delay from periodic push based monitoring approach, these delays also decrease cluster utilization and directly affects companies' revenues.

We propose adaptive distributed/centralized resource allocation to address the aforementioned timing delay and to increase in cluster utilization. In low loads, MACRM lets per-job schedulers to sample cluster nodes to find free resources for their tasks. This eliminates all the aforementioned timing delay since this approach does not depend on heartbeats and does not include the application of allocation policies. Moreover, since each job has its scheduler, "head of line blocking" is irrelevant. The next two sections present our approach for distributed resource allocation. In high loads, MACRM falls back to centralized resource allocation to be able to apply allocation policies. However, it uses a novel event-driven/periodic heartbeats to avoid timing delay of heartbeats. Also, it utilizes parallel scheduling to address the "head of line blocking" problem. Section 3.1.5 presents this resource allocation mode in details.

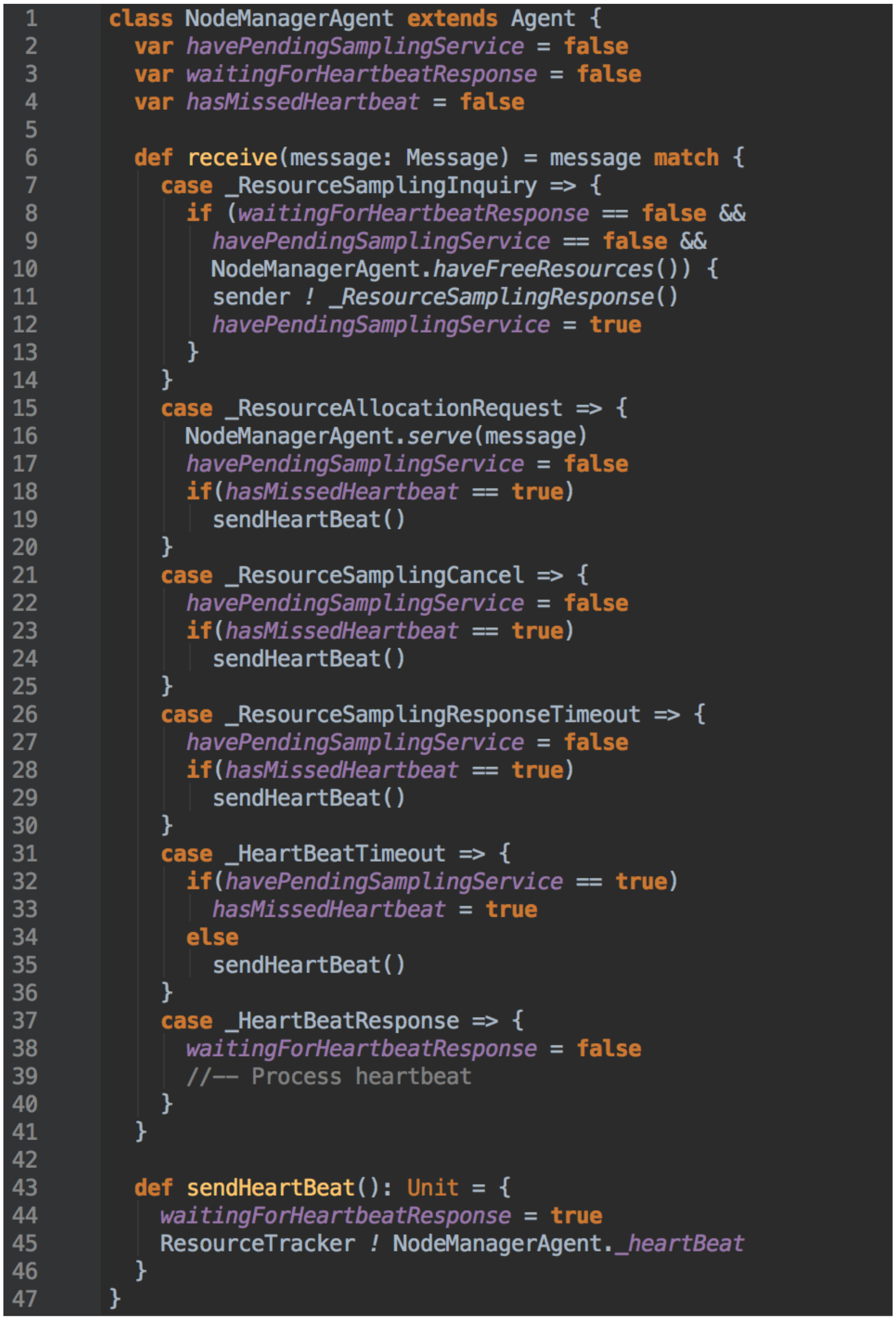
3.1.3 Power of Two Choices

As mentioned earlier, there is a dedicated scheduler for each job in MACRM's design which executes inside a container.[1] Since these schedulers have computing and communication resources, they can independently search cluster nodes for free resources. A straightforward way to implement resource lookup in per-job schedulers is sharing the whole cluster state with them. Then, the schedulers can concurrently allocate resources based on free resources in the cluster state. However, this approach imposes a considerable communication overhead to share and update the cluster state. Moreover, sharing the same cluster state leads to allocation conflict since schedulers tend to allocate resources to lightly loaded nodes to decrease resource interference between tasks on a single node [10].

To avoid the overhead of cluster state sharing and decrease conflicting allocation we chose to use random resource sampling techniques. Finding free resources and allocating them for tasks' execution is a load balancing problem. In this problem, we are looking for a balanced task distribution among a set of machines. This problem can be represented as a balls and bins problem [55] in which balls are tasks and bins are cluster servers. Suppose that n balls are placed into n bins by choosing bins independently and uniformly at random. It is well known that with high probability, the maximum number of balls in each bin will be $\frac{\log n}{\log \log n}$. Azar et. al. [7] show that if we choose two bins independently and uniformly at random for each ball, and place the ball into the less full bin, the maximum bin load with high probability drops to $\frac{\log \log n}{\log 2} + O(1)$, which is exponential improvement. Therefore, by utilizing the power of two choices, we can achieve close to $O(1)$ load in each bin with a stateless approach. The network overhead of this method is very small compared with sharing cluster state since it only requires sending very small sampling messages. Also, the sampling messages starting points and destinations are spread across network while cluster state should be served by a centralized entity which eventually leads to scalability issue.

Consequently, when a user submits a job to the resource arbitrator, it allocates a container to execute a job manager agent for the submitted job. Then, the job manager starts sampling cluster nodes to find free resources, instead of sending resource requests to the resource arbitrator. The job manager should send a message to a node's node manager to sample the node. Each worker machine has a node manager which monitors resources and containers on that machine. To this end, the job manager needs to know the IP addresses of the nodes. The resource arbitrator preloads this information into a job manager configuration

[1]Containers are system-level virtualization method that can limit a process tree access to computational resources and devices.



```
class NodeManagerAgent extends Agent {
  var havePendingSamplingService = false
  var waitingForHeartbeatResponse = false
  var hasMissedHeartbeat = false

  def receive(message: Message) = message match {
    case _ResourceSamplingInquiry => {
      if (waitingForHeartbeatResponse == false &&
        havePendingSamplingService == false &&
        NodeManagerAgent.haveFreeResources()) {
        sender ! _ResourceSamplingResponse()
        havePendingSamplingService = true
      }
    }
    case _ResourceAllocationRequest => {
      NodeManagerAgent.serve(message)
      havePendingSamplingService = false
      if(hasMissedHeartbeat == true)
        sendHeartBeat()
    }
    case _ResourceSamplingCancel => {
      havePendingSamplingService = false
      if(hasMissedHeartbeat == true)
        sendHeartBeat()
    }
    case _ResourceSamplingResponseTimeout => {
      havePendingSamplingService = false
      if(hasMissedHeartbeat == true)
        sendHeartBeat()
    }
    case _HeartBeatTimeout => {
      if(havePendingSamplingService == true)
        hasMissedHeartbeat = true
      else
        sendHeartBeat()
    }
    case _HeartBeatResponse => {
      waitingForHeartbeatResponse = false
      //-- Process heartbeat
    }
  }

  def sendHeartBeat(): Unit = {
    waitingForHeartbeatResponse = true
    ResourceTracker ! NodeManagerAgent._heartBeat
  }
}
```

Figure 3.3: Node Manager logic for managing node samplings.

file during container allocation process. Compared with sharing the whole cluster state, the size of cluster structure or IP address range is typically very small. For example, when the cluster has a continuous range of IP addresses on its nodes, size of the pre-loaded information is eight bytes, four bytes for the first IP address in range and four bytes for the last IP address.

To further adapt the power of two choices technique for jobs with multiple tasks, we utilize the recently developed multiple choices approach [55]. In this approach, instead of performing sampling for each task independently, a job manager places the n tasks of the job on the least loaded of $d \times n$ randomly selected nodes, where $d \geq 2$. Arranging and putting samples together improves per-task sampling by sharing information across all the samplings. However, it is still possible that the job manager cannot find enough free resources and the sampling fails since the job manager does not hear back from some of the sampled nodes.

A sample node does not respond back to a sampling request in any of the following situations: a) The sampled node is not available due to a hardware failure or crash of its node manager. b) The node manager has sent a heartbeat and is waiting for its response. In this case, the node manager ignores any sampling request because it may receive a resource allocation from the resource manager in the response. c) The node manager has responded to another sampling request and is waiting for its response. d) The node does not have free resources. Figure 3.3 shows the algorithm of a node manger for handling resource sampling inquiries.

To recognize a sampling failure, a job manager should initiate a timer after the submission of resource sampling requests. Then, it should wait for responses from the sampled nodes. If the job manager cannot acquire enough resources from the received responses and the timer times out, the job manager should repeat the sampling by querying a new set of nodes. However, since the sampling is based on random selections, it is still possible that the job manager cannot find enough resources after several retries. This leads to a considerable latency in tasks execution and increases a job's response time. To address this issue, we have devised different techniques to decrease resource allocation latency through random sampling which will be presented in the next section.

3.1.4 Adaptive Sampling Rate

To further investigate the power of two choices with batch sampling and the effects of sampling retires on the response time of a job, we performed a simulation experiment. In this simulation, we created a cluster of two hundred 4-core machines. We set up network round trip to one millisecond between simulated nodes with standard deviation of 0.02 millisecond. In the simulation, jobs arrive as a Poisson process with mean inter-arrival times of 1 second. Each job is composed of 9 tasks, and all the tasks have the same duration, which is chosen from an exponential distribution with a mean of one second. When tasks have different durations, short tasks can wait longer without affecting job response time; therefore, we use tasks of the same duration to put the most stress on the distributed resource allocation techniques. Also, using tasks with same duration helps us to focus the investigation just on the effects of sampling retries. Each task requires one core, and each job requires ten cores in total to complete its computation.

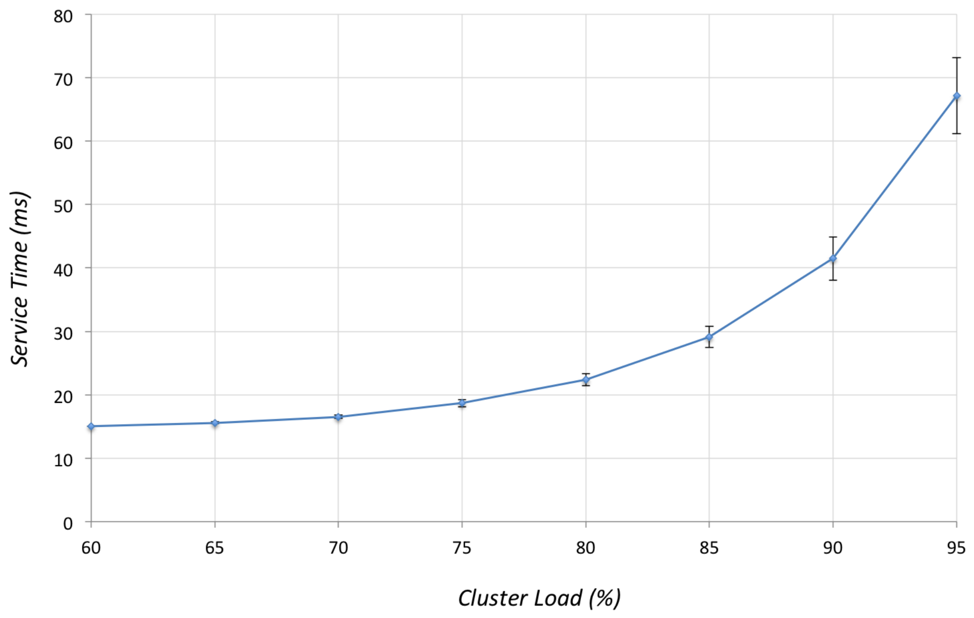


Figure 3.4: Average service time with the power of two choices

Figure 3.4 shows the average service time for one hundred jobs in different loads. We set the initial cluster load to 60% since the average memory and CPU usage in one-hour windows is around 50% and 60% respectively in production clusters [57]. We calculated the service time by subtracting duration of a task from the job's time span in the system. In this simulation, the sampling rate was two per task and the job managers applied batch sampling to acquire resources for nine tasks. Furthermore, the sampling timeout was set to five milliseconds and each node submitted heart beats every 1 second. As shown in Figure 3.4, the service time increases very fast as the cluster load increases. This has two main reasons: a) jobs encounter higher queuing delay inside the resource arbitrator during resource allocation for their job managers, and b) tasks experience delay as a result of several resource sampling failures.

To decrease the overhead of sampling retries we double the sampling rate after each failure. Figure 3.5 shows the results of simple sampling with the power of two choices compared to the sampling with doubling the sampling rate after each failure. While in the middle load (60% - 75%) there is not much difference, doubling the sampling rate after each failure decreases the service time more than 25% as the cluster load reaches 95%. This is because the doubling of the sampling rate increases the probability of finding free resources or decreases the likelihood of another sampling failure. In spite of this improvement, the sampling retries still happened several times, especially in high load, since the initial sampling rate was fixed and equal to two. To cope with this situation, we adaptively change the initial sampling rate.

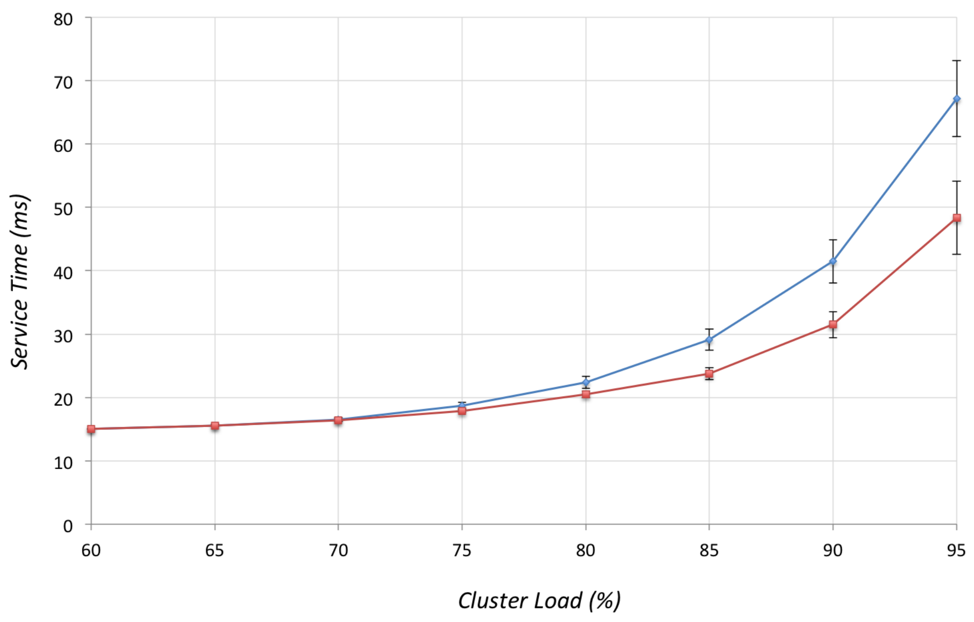


Figure 3.5: Comparison of the power of two choices (in blue) vs. the power of two choices with doubling sampling rate after each failure (in red).

To find out the other effective parameters for adaptively changing the initial sampling rate rather than cluster load, we performed another round of simulation. In this simulation, we focused on some potential parameters such as cluster size, number of tasks in each job, and jobs inter-arrival time. To this end, we arranged three cluster sizes with 100, 200, and 400 nodes. The total number of cores set to 16000 and the total memory set to 1600 gigabyte in all the three configurations. We used jobs with four or eight tasks with exponential distribution ($1/\lambda = 1sec$) durations. Furthermore, we kept the base load fixed and set it to 85% for all the tests. And finally, jobs were submitted as a Poisson process with mean inter-arrival times of 800 or 1200 milliseconds. We chose these two values to reflect both a shorter and a longer inter-arrival times compared with a production workload.[2]

As Figure 3.6 shows, compared with cluster load (Figure 3.5) other potential parameters do not have a considerable impact on the average service time and so the number of sampling retries. Although using more fine-grained nodes decreases the service time, this reduction is negligible (less than 4%) compared to doubling the number of nodes. Moreover, as the number of nodes increases, the arrival rate of heartbeats to the resource arbitrator increases, since we kept heartbeat's inter-arrival time fixed. Therefore, this reduction is also a result of smaller queuing delay in the resource arbitrator because it could allocate resource for job managers faster. Consequently, cluster load is the only significant criterion for adaptively changing the initial

[2]For further information about the jobs' inter-arrival times in a production workload check Section 4.2.1.

sampling rate.

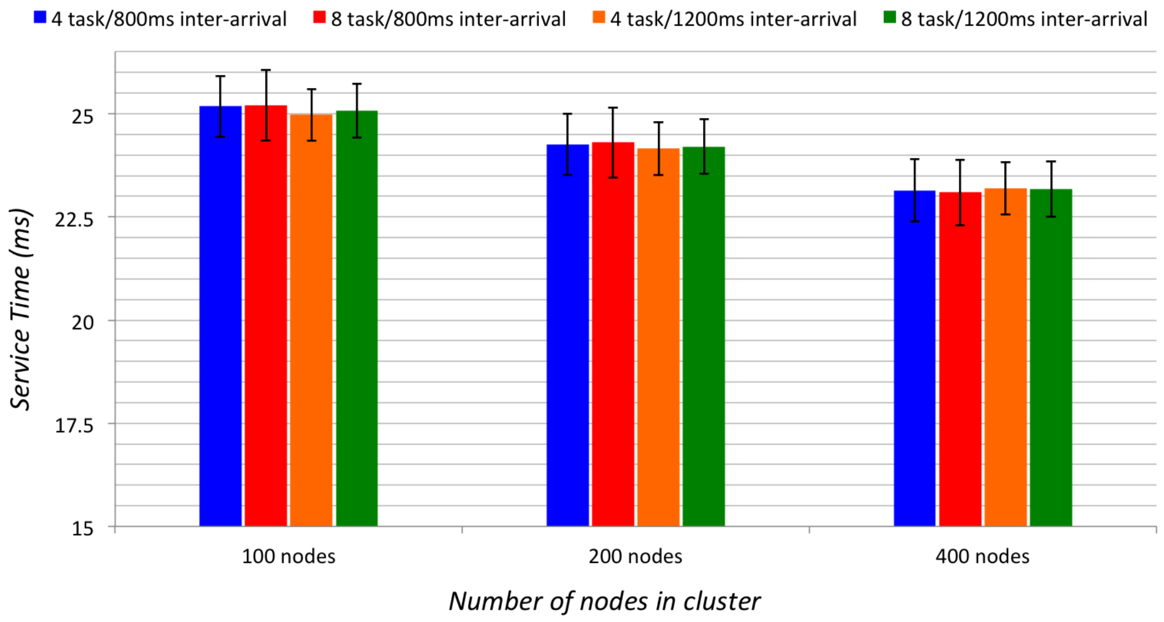


Figure 3.6: Effects of different parameters on average service time in sampling with the power of two choices at 85% load

To find a formula that calculates the best initial sampling rate based on the cluster load, we used interpolation methods. We repeated the previous simulation for 75%, 80%, 85%, and 90% cluster loads and used the best initial sampling rates for function interpolation. We chose these loads since failure in sampling with the rate of two starts to cause a notable increase in service time in 80% load (Figure 3.5). We observed that sampling rates of 2, 3, 5, and 8 were the best initial rates that minimize the number of sampling retries for 75%, 80%, 85%, and 90% cluster loads. By trying linear, quadratic, and exponential curve fitting methods, the best function found to fit the achieved results is:

$$x = clusterLoad \times 100 \Rightarrow initialSamplingRate = 0.02x^2 - 2.9x + 107 \tag{3.1}$$

Figure 3.7 shows the results of applying adaptive initial sampling rate on service time compared to the simple power of two choices and sampling by doubling the sampling rate after each failure. Utilizing adaptive sampling rate improves service time by up to 21% and 43% respectively compared with these two methods when the cluster has 5% free resources. Although applying aforementioned sampling techniques to find free resources decreases service time and increases system throughput, the service time still increases exponentially as the cluster load reaches a very high load. Furthermore, as mentioned in Section 3.1.2, in a highly loaded cluster the aggregated resource demands can exceed available resource which raises concerns about applying allocation invariants. To tackle these issues we have added another level of adaptation in our system which will be presented in the next section.

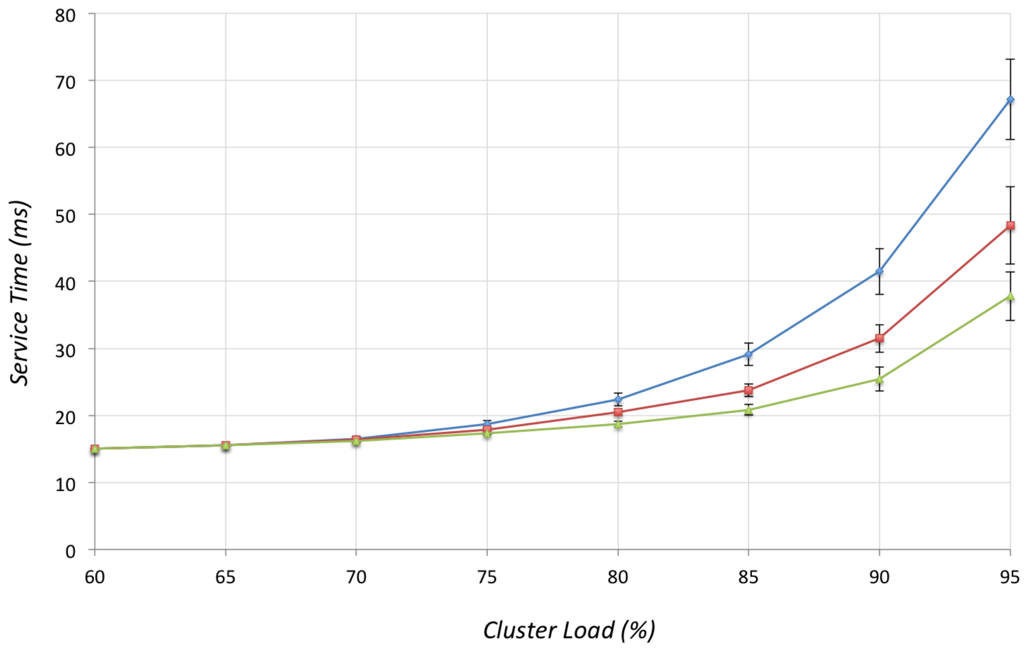


Figure 3.7: Average service time comparison of the adaptive initial sampling rate method (green) with the power of two choices (blue) and the power of two choices with doubling sampling rate after failure (red).

3.1.5 Switch to Centralized Resource Management

As mentioned in Section 3.1.2, the application of allocation policies, using periodic push based heartbeats for resource allocation, and head-of-line-blocking are three primary sources of delays in a centralized resource manager. To address these issues, we have utilized a distributed concurrent sampling approach to allocating free resources during low load conditions. However, as free resources inside the cluster drop to less than 10% of the total resources, two issues emerge: a) Overhead of sampling techniques increases as the initial sampling rate and the number of sampling retries grow. b) Conducting allocation policies becomes unavoidable which is very hard to manage in a distributed mode. Considering these facts, MACRM adaptively switches to centralized resource allocation when the cluster is highly loaded. Consequently, the job managers send their resource requests to the resource arbitrator instead of discovering free resources by sampling.

Changing from the distributed mode to the centralized mode enables MACRM to easily manage the application of allocation policies. The same as the other centralized resource manager, MACRM employs different queuing methods inside the resource arbitrator. However, adhering to the periodic heartbeats for resource allocation not only increases service time but also causes resource under-utilization when the aggregate resource demand is higher than available resources. Resource under-utilization happens because free resources in nodes remain unused because the resource arbitrator is not immediately notified. A proper solution to eliminate this latency is the utilization of an event-driven heartbeat. However, using event-driven heartbeats requires an alternative mechanism to find node failures. To solve both the latency issue and node failures discovery, we have developed a hybrid event-based/periodic heartbeats system when the MACRM switches to its centralized mode.

In this method, as soon as a container's execution finishes and its resources become available, the node manger cancels the heartbeat timer, immediately sends a heartbeat to notify the resource arbitrator, and schedules another heartbeat timer. Also, if the heartbeat timer finishes and the node does not have free resources, the node manager avoids sending heartbeat and schedules a new timer for half of heartbeat interval. In this case, if the timer expires again and the node does not have free resources, the node manager sends a heartbeat and schedules the next heartbeats. On the other side, the resource arbitrator starts preserving a cluster view of the nodes with free resources. This is important since there may not be any resource requests when the heartbeats arrive at the resource arbitrator, but immediately after that, a burst of resource requests arrives which can be served by previously received free resources.

To address the head-of-line-blocking, we devised a parallel request serving method. To this end, multiple agents pick resource requests from the resource arbitrator's queue and try to match the requests with resources inside the cluster view. The implementation details of this method is presented in Section 3.2.2. Our experiments show that utilizing hybrid event-driven/periodic heartbeat beside parallel request serving considerably decreases the service time compare with the random sampling in high load. Figure 3.8 shows the achieved result in a simulations.

Although switching to the centralized mode in high load decreases the service time compared to the adaptive sampling rate, using this method in lower load is not a viable solution due to the following facts:

- Scalability concerns: as the cluster load decreases, overhead of using event-driven heartbeats increases and it limits the system scalability. The overhead of this method is the number of messages that are received by the resource arbitrator and the required memory to preserve the view of the free resources in a cluster.
- Fault-tolerance concerns: Relying on centralized resource management makes all the computations dependent on the resource arbitrator for their progress which intensifies the single point of failure problem. Furthermore, recovering the resource arbitrator from failures becomes more expensive and prolonged since its internal state, view of the free resources, grows larger.
- Marginal improvements: When the cluster load is less than 90%, the achieved speedup in serving requests is low compared with the adaptive sampling method as Figure 3.8 shows.

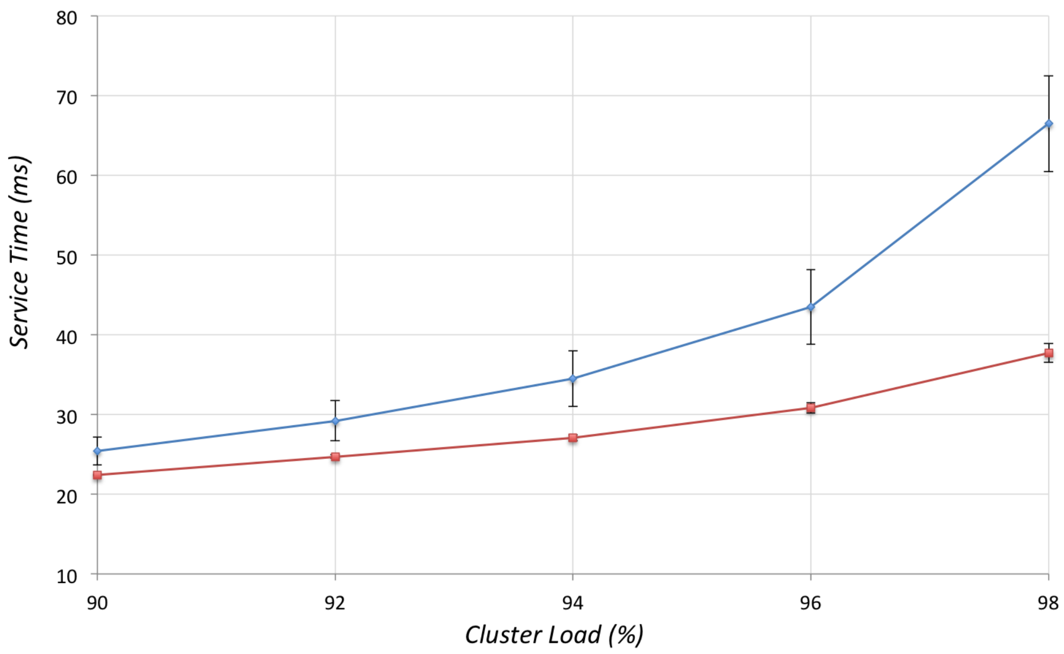


Figure 3.8: Average service time comparison of the hybrid event-driven/periodic heartbeat + parallel request serving (red) with the power of two choices + adaptive initial sampling rate (blue).

3.2 MACRM Implementation

The same as the other cluster resource management systems, MACRM is a message-oriented system. However, its components have been implemented by the means of actors to be more reactive to errors and crashes. This section presents MACRM's implementation using actor model of concurrency in details.

3.2.1 Actor-based System Design

As Figure 3.1 shows, MACRM has four major components. To provide their functionalities and perform corresponding responsibilities, each of them relies on several sub-components. In other words, a component's responsibilities are divided among the sub-components, and each of them can execute as a separate computational process. A sub-components requires information or services from the other sub-components or the other components to perform its responsibilities. As a result, designing concurrent sub-components increases throughput and responsiveness of a component and in a higher level the whole cluster management system.

There are four main approaches to build a concurrent system [27]: a) Concurrency based on threads, locks and shared state; b) Concurrency via software transactional memory; c) Event-driven concurrency; and d) Actor-based concurrency. The first two concurrency approaches result in procedure-oriented systems that are based on shared mutable data [41]. In thread-based systems, the software developers are in charge of controlling access to the shared data while in software transactional memory applications the development framework provides mutual exclusion.

Event-driven and actor-based concurrency approaches lead to message-oriented systems. Although both models are duals of each other and have identical performances [41], all of the existing cluster management are message-oriented systems based on event-driven concurrency approach. This has two primary reasons: a) A cluster management system and its internal structure are more message-oriented in nature rather than procedure-oriented. b) Message-oriented systems are more robust compared to procedure-oriented systems since they do not grapple with deadlocks, live-locks, and race conditions.

Considering these facts, we have chosen message-oriented concurrency design for development of MACRM components. However, instead of building event-driven components, we have developed components by composing several actors as sub-components. This design preference is on account of novel supervisor hierarchy with let-it-crash semantic for fault-tolerance. This fault management approach has been used with a large success by telecommunication industry to build self-healing application and system that never stop [68].

Throwing an exception in a concurrent code simply blows up the thread that executes the code. In this case, the only way to find the causing problem or the bug is the inspection of the stack traces. However, linking actors in a hierarchical fashion provides a clean way of getting notifications on errors and do something about them. Let-it-crash semantic encourages non-defensive programming in which programmers do not try to prevent errors or failures since they will eventually happen. Instead, the semantic expects failure as a natural state in applications' life cycles. Therefore, it lets the components crash early and another component with the whole picture of context deals with the failure.

In an actor system, a supervisor actor is responsible for starting, stopping and monitoring its child actors. The basic idea is that the supervisor should keep its child actor alive by restarting them when errors happen. There are three different restart strategies: a) One for one: Only the process that terminates will be restarted. b) One for all: all the child processes will be restarted in a case of a child failure. c) Rest for one: if a child process terminates, the 'rest' of the child processes (i.e. the child processes after the terminated process in start order) are terminated. Then the terminated child and the rest of the child processes are restarted. [50]. Depending on coupling semantics among child actors, the programmer should choose one of the aforementioned strategies. The rest of this chapter presents the implementation of resource arbitrator and resource tracker by the means of Scala programming languages and its actor library Akka.

3.2.2 Resource Arbitrator Structure

The resource arbitrator is a service which executes on a dedicated machine and acts as a central authority of the cluster. As mentioned in pervious sections, it is in charge of receiving user submitted jobs and finding free resources to execute job managers. Besides, in a highly loaded cluster, it is responsible for receiving resource requests from job managers and allocating resource for tasks execution. It allocates tasks' containers according to the allocation policies in order to arbitrator contention among users and jobs. To accomplish these responsibilities, we build the resource arbitrator by composing five types of actors as shown in Figure 3.9. The cluster-manager-actor is supervisor actor that creates the other actors in Figure 3.9 and monitors their liveness. Also, it holds the interface to the resource tracker of MACRM on which it receives information about nodes with free resource and updates of the cluster state.

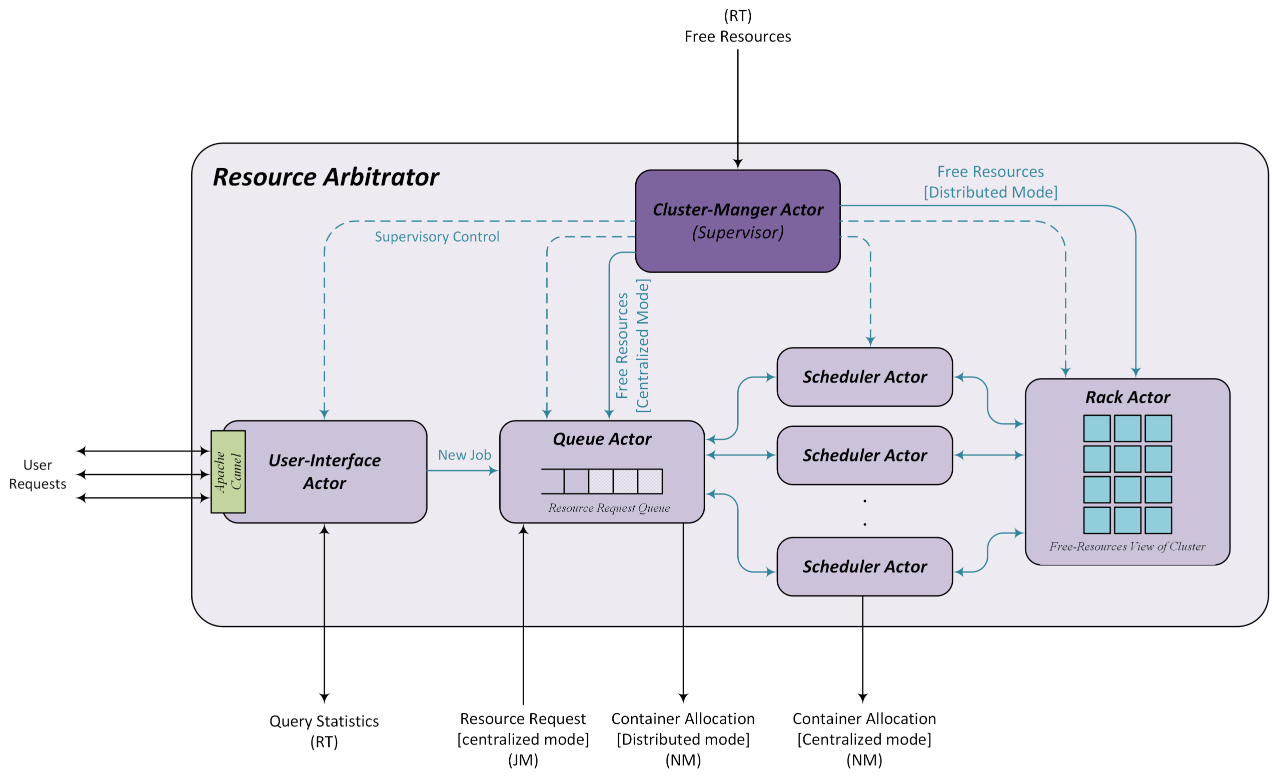


Figure 3.9: Resource Arbitrator architecture (Parentheses represent destination/source of the messages).

The user-interface-actor is in charge of accepting user submitted jobs. Since users' agents are not part of a cluster management system, we can only assume that they are software systems. Therefore, the API of the user-interface-actor should be comprehensive to provide a communication channel for all different kind of users' agents. To this end, we used Apache Camel [33] which focuses on making software integration easier by providing: a) Concrete implementations of all the widely used Enterprise Integration Patterns (EIP). b) Connectivity to a great variety of APIs and transports protocols. c) Easy to use Domain Specific Languages to wire EIPs and transports together. Consequently, the user-interface-actor can receive users' jobs which are capsulated inside Camel messages by listening on a predefined port. Following receipt of a message for job submission, it authenticates the user, examines the job description, and then prepares the job for scheduling.

To serve a job, MACRM expects that a user specifies tasks' hard constraints in the job description. Therefore, if the cluster does not have proper resources, such as nodes with a particular software package that is required by a task, the user-interface-actor can reject the job submission. Otherwise, based on the specified constraints, the user-interface-actor creates a configuration file that contains a list of nodes that can serve each constraint.

For building the configuration file, the user-interface-actor uses the information inside the cluster state. Cluster state is a data structure, here is a list, that contains contact information for the node mangers and the software and hardware capabilities of their corresponding nodes. A job manger utilizes the configuration file for sampling when the cluster is in low load. Since the job inter-arrival rate may be so high for an extremely large cluster, we parallelized the creation process of the configuration file by the means of Future property in Scala. After building the configuration file, the user-interface-actor creates a unique id for the job and then submits it to the queue-actor.

The queue-actor has different responsibilities based on the current load of a cluster and MACRM centralized/distributed resource management mode. In low loads, the main responsibility of the queue-actor is container allocation for execution of the job managers. To this end, it holds a FIFO queue[3] of the submitted jobs and whenever information about a node with free resources arrives from the resource tracker, it allocates containers on that node. In high loads, the responsibility of the queue-actor shrinks to queuing resource requests and applying allocation policies. Also, it does not receive information about nodes with free resource anymore. The cluster-manger-actor routes this information to the rack-actor. Therefore, the queue-actor sends the queued requests to scheduler-actors instead of allocating resource for them itself.

As mentioned in Section 3.1.5, MACRM has parallel scheduler-actors for serving resource requests in high load to address the head-of-line-blocking issue. Figure 3.10 presents the workflow of the Schedulers actors. Since there are multiple concurrent scheduler-actors, it is possible that their resource allocations lead to conflicts. To discover these conflicts and resolve them, scheduler-actors should send changes for their allocations in their local cluster views to the rack-actor (Step 7). Updating of the cluster view in the rack-actor is an atomic commit since an actor processes messages one at a time without any overlap. Therefore, in the case of conflict at most one of the commits will succeed. To deal with conflicts, we utilize two approaches:

[3]Depending on the administration rules and allocation policies of a cluster, the type of queue can be changed.

```
1: resourceRequest ← **SendMessage**(QueueActor, Dequeue from the resource requests queue)
2: clusterViewCopy ← **SendMessage**(RackActor, Get a copy of the view of free resources)
3: (isMatched, changesToClusterView) ← **FindMatching**(resourceRequest, clusterViewCopy)
4: if isMatched = false then
5:     goto $step2$
6: else
7:     isSucceeded ← SendMessage(RackActor, changesToClusterView, Try update cluster view)
8:     if isSucceeded = $true$ then
9:         for nodeMangerAddresses in changesToClusterView do
10:            SendMessage(nodeMangerAddresses, Allocate container)
11:        end for
12:    else
13:        goto $step2$
14:    end if
15: end if
```

Figure 3.10: A scheduler actor pseudo code

a) All-or-nothing: the entire matching process is repeated if any of the container placement causes conflict b) Incremental: the matching process is repeated only for the conflicting changes while successful placements are scheduled. Although the first approach may lead to additional conflicts and increases the scheduler-actors busyness, it is necessary for requests with gang-scheduling constraints. Pseudo code in Figure 3.10 shows the all-or-nothing approach.

3.2.3 Resource Tracker Structure

The same as the resource arbitrator, MACRM's resource tracker is composed of different actors. There are three types of actor to accomplish the resource tracker responsibilities. The supervisor actor is named resource-tracker-actor which is only responsible for creating and monitoring other actors. Therefore, it does not have any responsibilities related to the cluster resource management process. The two other actors are cluster-state-reader and cluster-state-writer.

A cluster-state-reader is responsible for monitoring cluster's events by analyzing the cluster view. It tracks the nodes liveness and discovers both node failures and node arrival to a cluster based on heartbeats' arrival times. After discovering changes in a node, it sends a message to update the cluster state of the resource arbitrator. Furthermore, it gathers statistic about the cluster such as each user share of the resources, overall resource usage, and jobs' progress. This information are used for applying allocation policies, adjusting sampling rate, switching between centralized and distributed mode, and responding to users' queries about their jobs progress.

A cluster-state-writer is in charge of receiving heartbeats from both node managers and job managers and updating the cluster view. Furthermore, if it receives a heartbeat from a node with free resources, it forwards the heartbeat to the resource arbitrator. Although the cluster-state-writer has simple responsibilities, it should process each heartbeat in less than 0.1ms for a cluster with 10,000 nodes and heartbeat interval of one second. If the cluster-state-writer cannot provide this throughput, heartbeat arrival will overflow its mailbox and causes memory crash. As mentioned in section 3.1.1, MACRM can have multiple distributed resource trackers to scale for an extremely large cluster.

To create a distributed mode, MACRM increases the number of cluster-state-writer and partitions the cluster nodes among them. In this case, a job manager sends heartbeats to the same cluster-state-writer actor to which its host node sends heartbeats. Instances of the cluster-state-writer are distributed on several nodes by utilizing Akka Clustering [19] which automatically distributes and balances actors across multiple nodes. In Akka clustering, the number of cluster nodes and the number of executing actors can change without any downtime. Therefore, this design can easily scale with cluster size. With this design, the resource tracker robustness to failure increases dramatically since the cluster-state-writers are distributed on several nodes and work independently.

When there are multiple distributed cluster-state-writers, the cluster-state-reader should gather all the partial cluster views from the cluster-state-writers. However, instead of transferring the partial cluster views to the cluster-state-reader, cluster-state-writers just send the processing results. This model matches well with the information that the cluster-state-reader requires since most of them are aggregated information such as average cluster utilization or total share of each user. For the other kind of information such as node failure or arrival, the cluster-state-reader periodically queries the cluster-state-writers, and they look into their cluster views.

Chapter 4
Experimental Evaluation

This chapter presents the evaluation of our MACRM system described in Chapter 3. MACRM is a hybrid distributed/centralized system which tries to offer advantages of both worlds. In the absence of a similar hybrid system comparing it with both a centralized and a distributed system is the best way to evaluate MACRM. Among the major systems presented in Chapter 2, only YARN [67], Mesos [32], and Spark [73] have open-source implementation. These are all centralized resource managers with fine-grained schedulers; YARN and Spark utilize per-job schedulers while Mesos uses per-framework schedulers. Among them, we chose YARN for this evaluation since it uses fine-grained per-job schedulers like MACRM and it is a general purpose system compared to Spark which is designed to be an in-memory resource manager.

We apply a change to YARN to make it more competitive in the evaluation. In YARN, when a task finishes its computation, it notifies YARN's Resource Manager and then the Resource Manger notifies the corresponding Job Manager about the task completion. This increases a job's timespan in the system at least by a network RTT. Therefore, instead of notifying the Resource Manager about a task completion, we directly notify the related Job Manager.

Based on our description in Section 2.4.4, we have two candidates for the distributed system in our evaluation: Sparrow [52] and Apollo [10]. There are no open-source implementations for these systems, so we had to implement either of them. We chose to use Sparrow instead of Apollo for the following two reasons. The first reason is that Apollo relies on task classification (into regular and opportunistic tasks) to perform; however, the paper lacks a detailed description of this classification, and it appears the system relies on users to provide this information. The second reason is the complexity of the algorithm for synchronizing distributed schedulers in Apollo compared to the simplicity of power of two choices mechanism in Sparrow. Furthermore, this synchronization mechanism decreases jobs' service times by imposing network overhead which is hard to formulate as a parameter in our comparison.

In a way similar to what we do with YARN, we apply a change to Sparrow to make our evaluation fair. Although Sparrow uses a constant number of schedulers for all jobs and their tasks, we create a scheduler for each submitted job like what MACRM and YARN do. As a result, the jobs are submitted to a queue that allocates containers for their schedulers (like job managers in YARN) by random sampling, and then the schedulers try to allocate resources for the tasks again by the means of random sampling. This way, the total amount of required resources for a job execution is equal in all three systems; also, the constant number of schedulers in Sparrow does not degrade its throughput.

4.1 Experiment setup

4.1.1 Google Cluster Trace

The Google cluster is a set of servers that are packed into racks and are connected by a high-bandwidth cluster network. These servers share a common cluster-management system that allocates resources to jobs [58]. The cluster trace released in October 2011 consists of what could be considered a month of activity of a single cluster with +12500 machines. The trace contains +668000 of jobs submitted by 679 users. Each job is composed of several tasks, which are Linux programs possibly consisting of multiple processes. These tasks are not gang-scheduled and are usually executed simultaneously [57].

We chose to use the Google cluster trace because it is the only available complete cluster trace; others lack important information for creating a comprehensive simulation. The Google cluster trace contains detailed information about nodes' configurations, tasks' resource requirements, and tasks' durations. In comparison, the other existing traces do not have information about the configuration of cluster nodes. Furthermore, some of them do not provide resource requirements of tasks and just present the total resource requirement of jobs. Although this trace goes back to the end of 2011, Google's cloud engineering team still supports and maintains it and has not released any new traces. As of early 2016, it is still in wide use as a valid and credible trace for research in the field of cluster computing.

Unlike traditional Grid and supercomputing environments, this trace represents a much higher diversity in different aspects. This cluster is constructed from a variety of machine classes at different points in the configuration space from processing to memory to storage (Table 4.1). Furthermore, there is a large heterogeneity in submitted jobs. Job durations span from seconds to more than the duration of the entire trace. Tasks have a wide diversity of resource demands. Each task specifies estimated maximum RAM and CPU requirements, and some have placement constraints (e.g., do not run on a machine without an external IP address).

Some information in the trace has been obfuscated for confidentiality reasons. For example, free-text fields have been randomly hashed, and resource sizes (for both tasks and machines) have been linearly scaled. This obfuscation has been done in a consistent way so that the data can still be useful for research studies [58]. The trace lacks precise information about the jobs' purposes; however, we can identify the distribution

Table 4.1: Configuration of machines in Google's cluster. Resources are linearly scaled so that the maximum value for CPU and memory is one.

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

of jobs, users, and machine characteristics.

To estimate some of the critical features of the Google cluster trace, we searched for available hardware technology during 2010-2011. At the time, the highest number of cores per machine was for Opteron AMD 6100-series with 12 cores. However, most of the machines in cluster trace belong to platform B, which has half of the number of cores compared to platform C. Considering popularity of Intel xenon processor for server machines and the fact that this architecture had at most 6 cores during 2010 to 2011, we think platform B is likely Intel Xenon processors. There were two models of this platform, Westmere-EP and Beckton. These models could support up to 12 and 16 Gigabyte of memory respectively. As a result, we think one unit of memory in the presented trace match 16 Gigabyte of actual memory. Table 4.2 presents our calculated assumption of the obfuscated information.

Table 4.2: Interpretation of Table 4.1

Obfuscated data	Meaning
A	Intel Itanium processor series
B	Intel Xenon processor series
C	AMD Opteron processor series
1.00 × CPUs	12 cores
1.00 × Memory	16 Gigabyte

4.1.2 Simulation Setup

To conduct a real experiment, we utilize a cluster's configuration and workload information from the Google cluster trace. Since access to a production cluster with thousands of machines is expensive, we decided to do simulation for testing MACRM performance. We simulated a cluster with 500 nodes; the cluster has the same distribution of machine configuration like Google's cluster. For example, since 6732 out of 12583 (53.5%) machines in the Google cluster use platform B with 0.5 CPU and 0.5 memory, we set 268 out of 500 machines of our simulated cluster with six cores and eight Gigabyte of memory (Based on our interpretation in Table 4.2). As a result, the simulated cluster has 3770 Gigabyte of memory and 3177 cores in total.

To run our experiments, we utilize 26 workstation machines.[1] Each of these machines has 16 Gigabyte of RAM and an Intel Core i7 processor; also, they are all connected through a single switch. We run the resource arbitrator and the resource tracker on one of these nodes, and the rest of nodes are used for hosting cluster nodes. Therefore, each workstation hosts 20 nodes of the simulated cluster. Since we do not have actual computing hardware, we've chosen to avoid actual execution of tasks. Instead of that, we simulate execution of a task on a simulated node by decreasing available resources of the node for the duration of the tasks execution. As a result, our experiment is partly simulation and partly emulation of a real cluster with 500 nodes.

Before running the experiment, we restart all the machines to make sure that there is no interfering program.[2] For each round of simulation, we take following steps: a) Run the resource manager (in case of MACRM, run the resource tracker and the resource arbitrator) on one of the machines. b) Run twenty simulated nodes on each workstation, one at a time. c) Make sure that all the simulated nodes have sent at least one heartbeat to the resource manager (or resource tracker in the case of MACRM). d) Start the experiment by submitting jobs to the cluster.

The mean Round Trip Time (RTT)[3] between actual machines during simulation execution is 0.8 millisecond with a standard deviation of 0.02. Since messages between simulated nodes that are located on the same machine do not encounter this network delay, we utilized the message-scheduling feature of Akka's dispatcher library to apply random delay on these messages. As a result, if the destination of a message has the same IP address as the current actor but a different port number, we postpone the submission of the message based on a normal random distribution with mean 0.8 and standard deviation of 0.02.

[1]These are iMac "Core i7" (Late 2014/Retina 5K) features a 22 nm "Haswell" 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 TB "Fusion Drive" (1 TB hard drive and 128 GB SSD), and a AMD Radeon R9 M290X graphics processor with 2 GB of dedicated GDDR5 memory. They are part of a lab in the department of computer science.

[2]All the simulations have been performed between 12AM to 4AM while the computers were not in use for any other purpose.

[3]Found it out by using ping command of UNIX

The other important aspect of the experiment that we take from Google cluster trace is workload distribution. As mentioned earlier, there are diverse types of jobs, with a different number of tasks, various durations, and a diverse range of resource requirements in this trace. To manage and utilize this workload, we put all the jobs and their related tasks information (around 25 million tasks) into a relation database. Table 4.3 presents some of the workload's statistics of the trace based on our query to the database. To create workloads for our evaluations, we randomly select some of the jobs and create workload files as inputs for simulation. For the random selection, we created a "SELECT" query with "ORDER BY RAND()" filter which has a uniform distribution. The next section presents detail of this random selection.

Table 4.3: The workload's statistics of Google cluster trace

	Min	**Max**	**Average**
# tasks per job	1	5000	36.56
Required # core per task	0	0.5	0.0338
Required memory per task (GB)	0.95×10^{-6}	0.955	0.0284
Task duration (Sec)	1×10^{-6}	N/A	3000

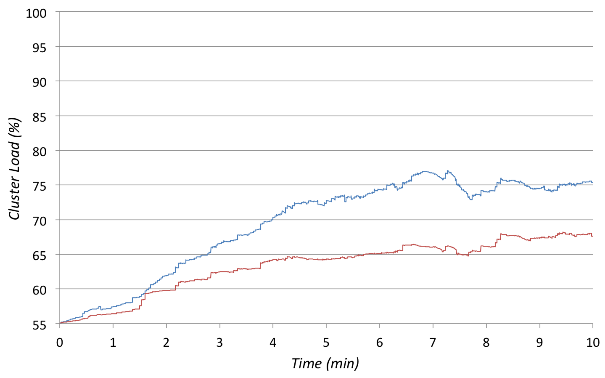
4.2 Evaluation And Results

We compared MACRM, YARN, and Sparrow along four different dimensions: system service time (4.2.1), scheduling throughput (4.2.2), scalability (4.2.3), and fault-tolerance (4.2.4).

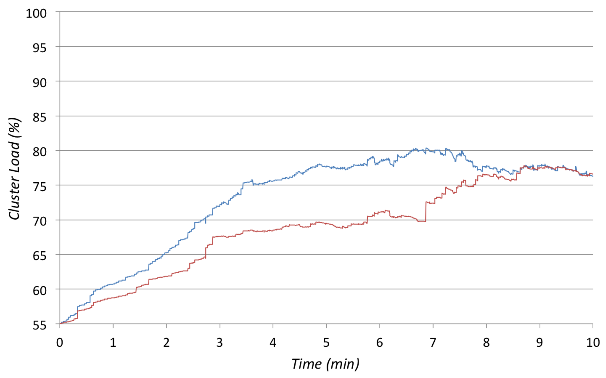
4.2.1 System Service Time

The first factor for which these systems are compared average service time for a job. Since there is no execution order among the tasks for a job in Google's trace, we measure the average service time by subtracting the duration of the longest task from the job's timespan in the system. The resulted number contains both the time that a job is waiting for allocation of its job manager and the time required to find free resources for tasks execution. Since these times depend on the cluster load, we try to compare these three systems at different load levels.

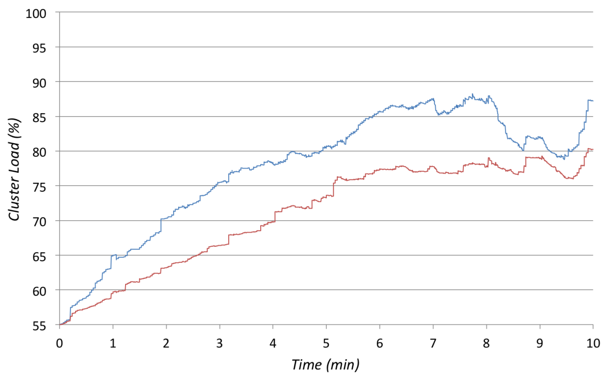
To create different load levels from jobs within Google trace, we create different sets of randomly selected jobs from the created database. Each of these sets contains 1200 jobs, and we use them as input for our evaluation. The other important point is the initial load of the simulated cluster. We set the initial cluster load to 55% since the average memory and CPU usage in one-hour windows is around 50% and 60% respectively in the Google trace [57]. To set this initial load, we randomly allocate resources on the simulated cluster, and those resources stay occupied during the simulation time. Then, we feed each set of randomly selected jobs



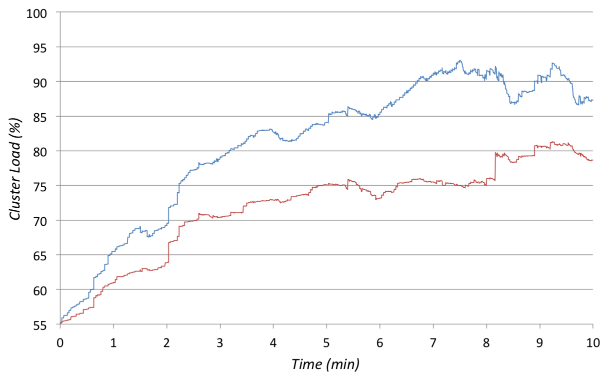
(a) 70%-load



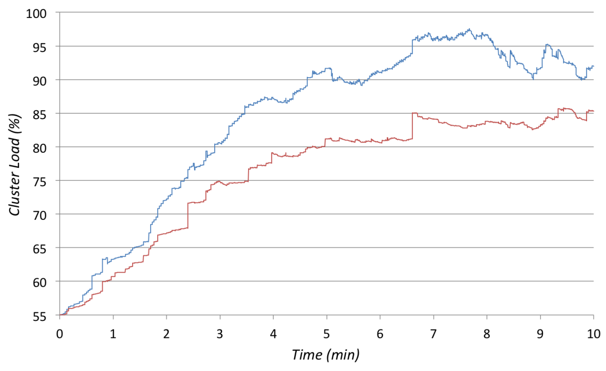
(b) 75%-load



(c) 80%-load



(d) 85%-load



(e) 90%-load

Figure 4.1: Cluster load changes for a sample from each group of the loads. Blue lines present CPU load and red lines present memory load.

to measure average service time for the systems. For each set, we change the initial random load. However, these initial loads are the same when we feed a specific set to each of MACRM, YARN, or Sparrow. Figure 4.1 shows the cluster load changes for some of the sets.

As Figure 4.1 shows, it is very hard to keep the load at a specific point with randomly selected jobs. Therefore, for comparison of the systems, we categorize sets based on the time they keep load inside a specific range. For example in Figure 4.1a the cluster load is close to 70% for most of the simulation time (more that 60% of simulation time), so we categorize this set in the group of 70% load. We make five different groups for 70%-load, 75%-load, 80%-load, 85%-load, and 90%-load and each group contains four to five sets of jobs. It is worth notify that we measure the service time just for the jobs which arrive at the cluster when the load is between -5% and +5% of group load. For example to measure service time for the 70%-load group we only measure service time when the cluster load is in range 65% to 75%.

We run the simulated cluster with each of MACRM, YARN, and Sparrow serving as the cluster manager and feed into the cluster one set at a time. Since the goal of this experiment is to measure the average service time for a job we set the inter-arrival time between the jobs to 500 milliseconds in order to measure the minimum possible service time regardless of possible competition among jobs to acquire resources. Considering Google trace duration, the number of submitted job, and assuming eight hours of work per day, 500 milliseconds inter-arrival time is less than half of the average inter-arrival time in the trace.

Figure 4.2 shows the average service time in each group for each of the systems. Considering the result, the centralized resource manager has the highest service time. This is because YARN's resource manager should wait until nodes inform it about their available resources, and this leads to sequential resource allocation. Therefore, a job's tasks encounter queuing delay, even without any competitor job, because they cannot be served concurrently. However, this is not the case for both MACRM and Sparrow since the Job mangers in these systems simultaneously query multiple nodes to find free resources for serving tasks.

As we expected, increasing the cluster load increases the average service time in all the system. Sparrow has the highest growth compared to the other systems as a result of a reduction in the chance of finding free resources with random sampling. But, as Figure 4.2 shows, this is not the case for MACRM, which also use random sampling. This is the result of using adaptive initial sampling rate and doubling sampling rate after each failure. Furthermore, MACRM keeps the average service time low by switching to its centralized mode in very high loads.[4]

In this experiment, we tried to measure the average service time in a usual condition where the inter-

[4]Appendix A contains the detailed value of this experiment.

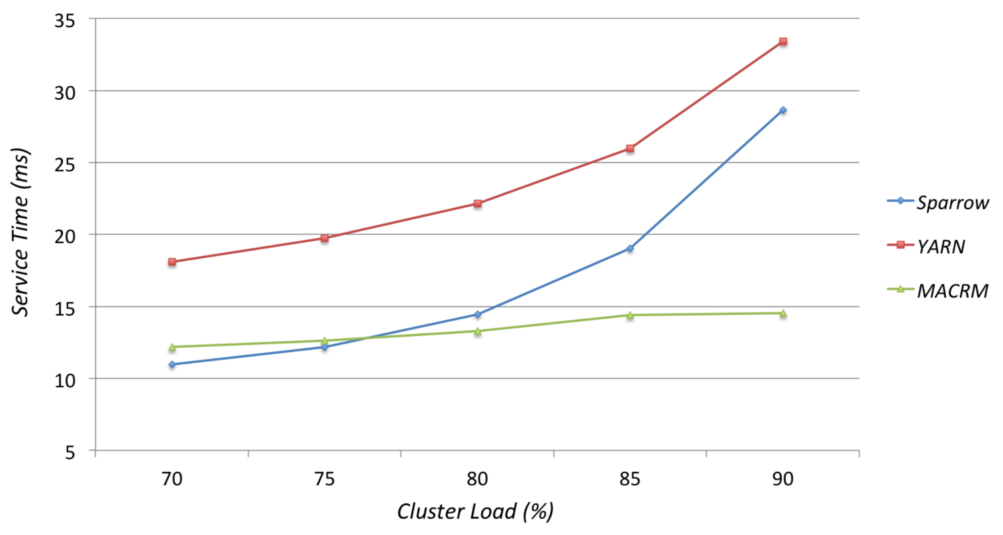


Figure 4.2: Average service time for YARN, MACRM, and Sparrow in a cluster with 500 nodes. The heartbeat inter-arrival time is 2000ms and the timeout for sampling retry in both MACRM and Sparrow is 3ms.

arrival between the jobs is larger than the average service time. However, in real world scenarios, there are times in which the cluster encounters a burst of job arrival. Especially when the number of users is high, it is probable that multiple users submit jobs all at the same time. In this case, serving the job or the task that is in front of resource allocation queue affects service time for the other jobs or tasks inside the queue. This is the scheduler's throughput which indicates the amount of the delay. To find out which of the aforementioned system performs better in these scenarios we did the following experiment.

4.2.2 Scheduling Throughput

To measure throughput of resource allocation for each of the aforementioned systems, again we used our simulated cluster. To simulate different levels of burst in job arrival, we created sixteen groups of inputs. Each group contains five different input files and each file has the information for specific number of jobs which has specific number of tasks. For example, the first group is for simulating the situation when five jobs each have ten tasks are submitted to the cluster simultaneously. To create this group, we queried our database of jobs from Google trace for jobs with ten to fifteen tasks. Then, if a retrieved job has more than ten tasks, we randomly drop their extra tasks. We repeated this process for the burst of ten, fifteen and twenty jobs that each has ten tasks. And the rest of the groups are created in the same way but for twenty, thirty and forty tasks.

Besides the size of the burst, cluster load was another variable factor in this experiment. We set the initial cluster load to 70%, 75%, 80%, 85%, and 90% and fed each burst into the cluster. For example, to set the initial load to 70%, we utilized the random class of java and its "nextGaussian" function to generate a normal distribution with mean 70 and standard deviation of 15 (which is (100 - 70)/2). Then, for each node, we generated two numbers one for CPU and one for memory and allocated CPU and memory according to the generated numbers. Finally, to measure the capability of the systems for serving burst of job arrival, we submit all the jobs of an input file simultaneously to the cluster

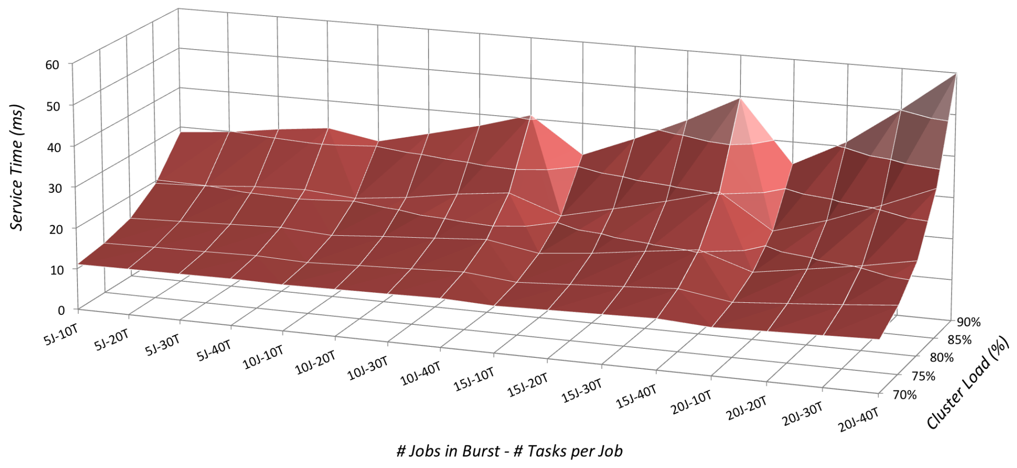
For comparison, we averaged the maximum service time of jobs for all the input files of a group. Figure 4.3 shows the results of these experiment for YARN, MACRM, and Sparrow. As we have expected, YARN has the worst results compared to the two other system as a result of its sequential resource allocation that imposes high queuing delay on jobs and their tasks. Sparrow and MACRM perform better since they make decision concurrently. In average loads, Sparrow outperforms MACRM since even for allocating job manager it makes decision concurrently while MACRM should wait for nodes' heartbeats to allocate job managers. However, as load increases, MACRM overall makes decisions faster because of adaptive initial sampling rate and double sampling rate. And finally, when MACRM switch to its centralized mode it achieves a very good result since it can quickly serve resource request by searching its cluster view of free resources.[5]

4.2.3 System Scalability

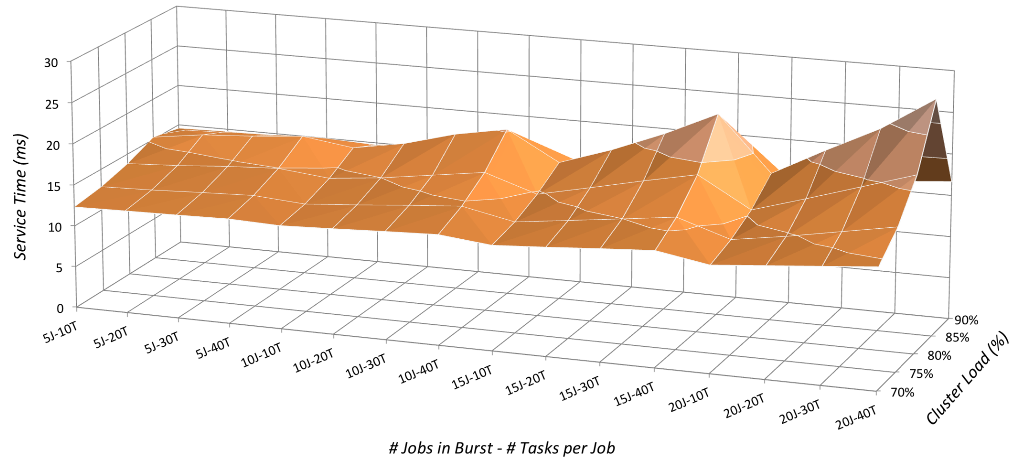
Although achieving low service time and high throughput are important parameters for resource managers, scalability of these systems is even more important in cluster computing environment. Like an operating system in a stand-alone computer, a cluster resource manager should monitor and control the cluster's resources. As a result, one aspect of scalability in cluster computing environment is the number of resources. However, there are two possible ways to increase resources of a distributed system compared with a stand-alone system. The first way is scaling up the system, which is the only option in stand-alone systems, that refers to adding more CPU, RAM, and storage to the existing. The second approach is scaling out, which is exclusive to distributed systems, by adding more nodes to the cluster.

In addition to the number of resources, large-scale systems such as clusters, supercomputers, and Grids have other scalability aspects that are not relevant to stand-alone systems. One of these sides is the number of users. Although, current operating systems designed to serve tens of users, large-scale systems like cluster usually face requests from hundreds of users. This is a challenging requirement to engineer inside a cluster resource manager because this large number of users usually access resources simultaneously, and they compete. To better define this dimension we can further break down it into two different parts based on user competition to access resources.

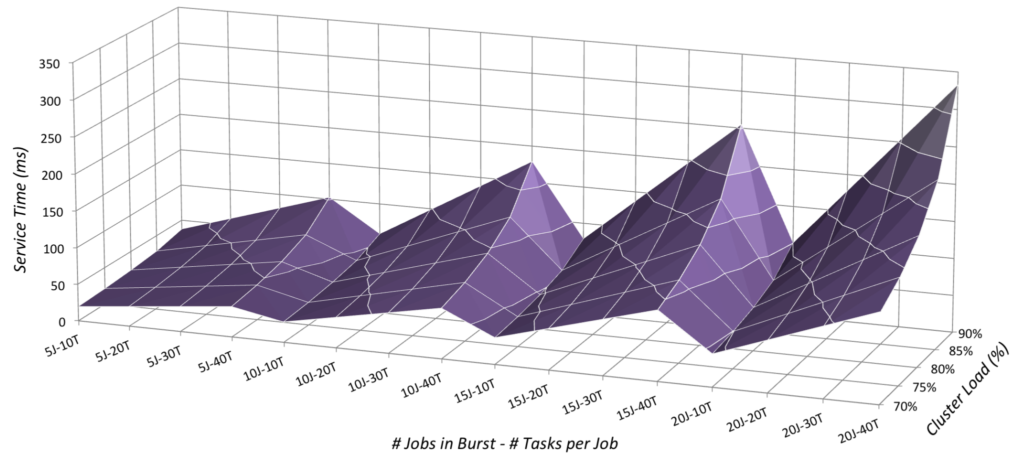
[5]Appendix I contains detail value of this experiment.



(a) Sparrow



(b) MACRM



(c) YARN

Figure 4.3: Throughput comparison of Sparrow, MACRM, and YARN.

The first part is when free resources are rare, and the resource manager should control user access to the resources by applying administration policies. Lack of these mechanisms causes starvation for some of the users and is a hindrance to serving even larger number of users. The second situation is when the aggregate available resources in cluster's nodes are more than requested resources from users. In this case applying administration policies and controlling users access to resources leads to resource under-utilization which again limits the number of users that can be served simultaneously. To sum up, managing large number of competing users based on the availability of resources is another dimension of scalability for clusters' resource managers.

The last aspect of scalability for a resource manager in cluster computing environment is the number of jobs. Like an operating system, a cluster resource manager should control and monitor life-cycle aspects of jobs such as resources fluctuation. This is a challenging task since a large cluster may host thousands of jobs with an enormous number of tasks in total. Although in a cluster computing environment, each node has an operating system which monitors its local processes, there should be another mechanism beyond single nodes because jobs mainly have multiple tasks that are distributed across the cluster nodes. Considering aforementioned scalability aspects, the following table compares MACRM with YARN and Sparrow from the scalability point of view.

Table 4.4: Scalability comparison of MACRM, YARN, and Sparrow

	#Resources		#Users		#Jobs
	Scaling up	**Scaling out**	**Competition**	**No competition**	**Jobs**
Sparrow	✓	✓	×	✓	×
YARN	✓	—	✓	×	✓
MACRM	✓	✓	✓	✓	✓

All the resources managers can handle scaling up in a cluster's resources as checks show in the first column of Table 4.4. This is because each node has an operating system that manages resources of its local node, and cluster resource managers access and control a node's resources through communication with node's operating system.

By scaling out resources, we add more nodes to the cluster. For resource managers like MACRM and YARN, which track available resources by the means of heartbeats, this means a higher number of heartbeats. For example in our simulated cluster, we had 500 nodes with 2000ms heartbeat inter-arrival time. As a result, the resource managers receives a heartbeat every 4ms on average. Suppose we increase the number of nodes to 5000, which is still a small production cluster size, while keeping the heartbeat inter-arrival time constant. Then, the resource manager receives a heartbeat every 0.4ms on average. This will eventually make the resource manager a scalability bottleneck. MACRM addresses this problem by breaking up the resource manager into a resource arbitrator and multiple resource trackers. Therefore, as number of nodes increases, MACRM increases the number of resource trackers. However, YARN cannot handle large increase in the number of nodes unless by increasing heartbeat inter-arrival time from nodes which will directly affect the average service time and scheduling throughput of YARN. Considering Sparrow, it can manage changes in number of nodes with no concern since it does not monitor cluster nodes.

Using hybrid centralized/distributed design in MACRM leads to a scalable resource management regarding the number of users. Considering number of competing users, because both MACRM and YARN have centralized entities they can regulate the competition. In YARN, the resource manager can apply administration policies while in MACRM the resource arbitrator manages the competition when system switch to centralized scheduling in high loads. However, since Sparrow trades simplicity of design for the accuracy of fair sharing by using aggregate fair sharing [52], it cannot manage this aspect. In aggregate fair-share, each user's share of resources on a node is proportional to the user priority or weight, regardless of the other nodes. As a result, if we have 400 users with a same priority, and each of them ask for resources on a node with 4 Gigabyte of RAM, each of them will receive 10 Megabyte of RAM which may not be enough for any of the tasks and lead to a job or user starvation.

Considering overhead of administration policies when the available resources are more than requested resources, both Sparrow and MACRM can serve users with low under-utilization. A Sparrow's scheduler starts resource allocation as soon as it receives a job or a tasks since Sparrow does not have a centralized administration policies application and does not require heartbeats from nodes to allocate resources. In MACRM, although the sequential centralized resource arbitrator handles the allocation of job managers by receiving heartbeats from nodes; however, it uses concurrent random sampling in low load for allocation of the container for tasks which leads to low resource under-utilization like Sparrow's scheduling. Contrary to MACRM, YARN always utilizes its centralized resource manager regardless of the cluster load. Therefore, to allocate resources for both job manager and tasks, it takes a sequential approach based on heartbeat arrivals from nodes. This leads to a resource under-utilization since resource requests should wait for application of administration policy and heartbeat arrival while there are lots of free resources in cluster nodes (Figure 4.3c).

The last aspect of scalability is the number of jobs in a cluster. In MACRM and YARN, since each job has a job manager who is responsible for managing the life-cycle aspect of it, number of jobs is not a scalability challenge for the resource manager. Although the allocated container for a job manager is a resource overhead, considering the average number of tasks per job and aggregation of their required resources, this is a viable overhead. Compared with having a job manger for each job, using a constant number of schedulers in Sparrow eventually limits the total number of jobs that can be managed inside a cluster. Specifically, since there is no defined method in Sparrow to scale the number of schedulers based on the number of jobs. Furthermore, each of Sparrow's schedulers require plenty of memory and network bandwidth to hold the states of all the scheduled jobs. To sum up, MACRM's design approach lets it easily handle different aspects of system scalability in a complicated environment of a cluster.

4.2.4 Fault-tolerance

As we have mentioned in Section 2.1, one of the inherent feature of cluster computing environment is failures. This is a result of using commodity hardware and serving a diverse range of jobs which are not the case in supercomputers and Grids respectively. Therefore, hardware crashes and job failure are ordinary events in cluster computing environment. Consequently, all the software components of a cluster should be designed considering this fact. In this section, we compare YARN, MACRM, and Sparrow in the context of failures and crashes.

In a cluster, hardware crashes can happen in different places. However, we can divide them into node crashes and network partition. A node crash also leads to failure of its containers which can contain users' processes or cluster management system's processes. Also, these processes can fail as a result of exceptions, high load, inaccessible packages, etc. Therefore, we provide the following list of failure for our compassion:

- **Task failure**: This is a failure in one of the users' processes. Task failure can be a result of an internal problem or exception inside the process. Also, the cluster management software may kill a process, for example when the process uses more that its allocated resources.
- **Job manager failure**: As its name implies this is a failure of a job manager. The same as task failure, it can be a result of both internal problems and external decisions.
- **Resource manager failure**: This failure happens when one the components of the resource manager fails. For example, when either resource arbitrator or any of resource tracker of MACRM fails.
- **Node failure**: This is either hardware failure in one the cluster node or failure of node manager.
- **Network Partition**: It occurs when a network switch inside a cluster crashes, and part of the network cannot communicate with the other part. Although in networks with fat-tree structure [2] this event is very rare but it is still possible.

YARN and MACRM react to a task failure more efficient than Sparrow since there is a job manager for each job in these systems which holds the internal logic of the related job. Therefore, the job manager makes an optimized decision in the case of a task failure based on the job's internal logic. For example, the job manager may only re-run the failed task instead of re-run all the tasks from the beginning. This is not true for Sparrow since it utilizes a constant number of general purpose schedulers which are responsible for all different types of jobs in a cluster.

Since Sparrow does not support per-job schedulers or job managers, failure of job manager is not applicable to it. However, both YARN and MACRM can recover from a job manager failure. YARN and MACRM rely on heartbeats from job managers to recognize failures or problems in them. In YARN, the centralized resource manager receives the heartbeats of the job managers while multiple distributed resource trackers are in charge of receiving the heartbeats in MACRM. As a result, job managers can send heartbeats to the resource trackers' of MACRM in very shorter interval while the same heartbeat interval hardly stresses the centralized resource manager of YARN. For example, MACRM can handle a second heartbeat interval from job managers while the default value for YARN is six minutes [67].

While YARN and MACRM outperform Sparrow for recovering a task failure, for resource manager's failure the case is reverse. In Sparrow, if any of its schedulers fails the user can easily switch to the other schedulers without any interrupt in the jobs' progress. However, YARN and MACRM require specific recovery mechanisms for this failure. YARN utilizes Zookeeper [28] to replicate the resource manager's internal state and when it fails the Zookeeper tries to replace it buy a replica. This process is more expensive and takes longer compared with MACRM.

In MACRM, we have broken down the resource manager to a resource arbitrator and multiple resource trackers. MACRM's resource tracker is also replicated by a Zookeeper. However, the replication and replacement process of it is faster and less expensive compared with YARN' resource manager since it has a very small logic and internal state. Moreover, MACRM's resource arbitrators do not require replications since their internal state is soft. Therefore, after a resource tracker failure, MACRM immediately instantiates another resource tracker which will contain the same state after a round of heartbeat arrival without any interrupt in any of services. Also, since MACRM operates in distributed fashion in low and middle loads (0%-90%), jobs which already have a job manager can continue working and allocate resources in the case of failure in the resource arbitrator while this is not the case for YARN.

Sparrow does not handle nodes' failures [52] since there is no monitoring mechanism for workstations, like heartbeats, in Sparrow design. This is one the biggest shortcomings of the Sparrow design. However, YARN and MACRM discover this failure through a timeout in a node's heartbeat arrival. The same as a job manager failure, MACRM can discover node failures faster than YARN. This is because MACRM's distributed resource trackers can receive heartbeats in a shorter interval compared with YARN's centralized resource manager. This is a big advantage for MACRM since a fast recovery from a node failure is very important when jobs are interactive and require fast turnaround.

Table 4.5: Comparison of MACRM, YARN, and Sparrow in the case of failures. The first part indicates how fast a system can recognize and recover from a failure and the second part shows recovery overhead compared with the other two systems.

	Failures				
	Task	**Job Manager**	**Resource Manager**	**Node**	**Network**
Sparrow	Moderate/High	N/A	Fast/Low	No mechanism	Fast/Low
YARN	Moderate/Low	Moderate/Low	Slow/High	Moderate/Low	Slow/High
MACRM	Moderate/Low	Fast/Low	Moderate/Middle	Fast/Low	Moderate/Middle

The last type of failure is network fragmentation. All the system can recover from this failure but with different costs. Network fragmentation is very costly for YARN since all the job managers that are not located on the same partition as the YARN resource manager cannot progress if they need resources. This is only true for MACRM in high loads (¿90%) when MACRM is in a centralized decision-making mode and Sparrow schedulers can still progress because they are independent. Moreover, if the failure lasts for a long time, YARN and MACRM will reschedule all the jobs and tasks that are fallen apart from the resource manager because they cannot track the liveness of those processes. This conservative approach trades efficiency with QoS. Table 4.5 summarizes the aforementioned discussions about fault-tolerance of YARN, MACRM, and Sparrow.

Chapter 5
Conclusion and Future Works

5.1 Conclusion

The cost effective design of clusters and the ease of application development for them make clusters an attractive option for many different types of computations. This leads to a diverse range of demands on the cluster resource management systems which has been evolved and increased during last years. To cope with this diversity, a cluster resource manager should be scalable, fault-tolerant, have high throughput, and low decision-making overhead. Survey of existing cluster resource management systems (Chapter 2) shows that they lack some of these features.

MACRM is a distributed/centralized resource management system which adapts to cluster load and satisfies the aforementioned requirements. Specifically, it adaptively changes to distributed scheduling when the cluster load is low and switches to centralized scheduling when the cluster load is high. Also, it has a multi-agent design to track the available resources, make scheduling decisions, monitor nodes liveness, and track job execution. Moreover, it has been implemented based on the actor model of concurrency to be scalable and fault-tolerant.

In its centralized mode, MACRM distributes resources by implementing a sharing principle like fair-share or capacity scheduling. Particularly, it is a scalable resource management system as a result of its modular and multi-agent design. MACRM's fine-grained scheduling and resource tracking, plus its actor-based implementation, lead to a robust system which can tolerate different types of failures. Moreover, concurrent decision-making and utilization of event-driven resource tracking in MACRM lead to high throughput resource allocation with low overhead.

5.2 Contributions

This study has several contributions to the field of cluster computing which we can categorize into three major parts. The first contribution of our work is providing a taxonomy of resource managers in Chapter 2 to explain the current state of the art in cluster's resource management. This classification, besides survey of the mainstream cluster scheduling systems in Chapter 2, establishes a structure for the future works in the field of cluster scheduling and resource management.

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

The last but not the least contribution of this work is using actor base model of concurrency for implementing a cluster resource management system. We have prototyped MACRM based on reactive systems manifesto which promotes systems which 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, MACRM is very flexible to future changes and improvements. Furthermore, MACRM components are loosely-coupled which improves its fault-tolerance significantly. Finally, we have utilized message passing feature of Akka, which is Scala actor library for asynchronous message passing.

5.3 Future Works

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 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 MACRM switches to centralized resource allocation in high load, there is no preemption mechanism in it. Exploring preemption mechanisms for MACRM and implementing them would be an interesting topic for future works.

Another unexplored area in this study is serving jobs with gang-scheduling requirement. Tasks of a gang-scheduling job require to be started all at the same time and usually a failure in one of them leads to complete job failure. Although this kind of jobs is rare, there were no jobs with gang-scheduling requirement in Google's trace, but 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 which requires lots of trade off in the scheduler. Currently, job managers in MACRM 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 resource on reliable machines introduces new challenges.

References

[1] Hadoop 1.x Configuration File. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[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*, pages 12–12. USENIX Association, 2011.

[4] InfiniBand Trade Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[5] Cluster Managment at Google by John Wilkes. https://www.youtube.com/watch?v=0zfmlo98jkc.

[6] Ching Avery. Giraph: Large-scale graph processing infrastruction on hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[7] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[8] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *Micro, Ieee*, 23(2):22–28, 2003.

[9] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[10] 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*, pages 285–300. USENIX Association, 2014.

[11] Rajkumar Buyya. High performance cluster computing: Architecture and systems, volume i. *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[12] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *International Journal of High Performance Computing Applications*, 15(2):124–135, 2001.

[13] Datacenter Energy Efficiency by Dileep Bhandarkar. http://bit.ly/e3lvu8.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[15] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[16] Yanpei Chen, Sara Alspaugh, and Randy H Katz. Design insights for mapreduce from diverse production workloads. Technical report, DTIC Document, 2012.

[17] David Clark. The design philosophy of the darpa internet protocols. *ACM SIGCOMM Computer Communication Review*, 18(4):106–114, 1988.

[18] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[19] Akka Clustering. http://doc.akka.io/docs/akka/snapshot/scala/cluster-usage.html.

[20] Brian F Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J Kistler, PPS Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, et al. Building a cloud for yahoo! *IEEE Data Eng. Bull.*, 32:36–43, 2009.

[21] Peter F. Corbett, Dror G Feitelson, J-P Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, et al. Parallel file systems for the ibm sp computers. *IBM Systems Journal*, 34(2):222–248, 1995.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 127–144. ACM, 2014.

[24] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tracil: Reconciling scheduling speed and quality in large, shared clusters. Technical report, Stanford University, Novermber 2014.

[25] Khaled Elmeleegy. Piranha: Optimizing short jobs in hadoop. *Proceedings of the VLDB Endowment*, 6(11):985–996, 2013.

[26] Facebook Engineering. Under the hood: Scheduling mapreduce jobs more efficiently with corona, 2012.

[27] Benjamin Erb. Concurrent programming for scalable web architectures. In *Informatiktage*, pages 139–142, 2012.

[28] Apache Software Foundation. Apache zookepper, 2010.

[29] Armando Fox, Rean Griffith, A Joseph, R Katz, A Konwinski, G Lee, D Patterson, A Rabkin, and I Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28:13, 2009.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[31] Amazon EC2 Cluster Placement Group. http://aws.amazon.com/ec2/faqs/.

[32] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[33] Claus Ibsen and Jonathan Anstey. *Camel in action*. Manning Publications Co., 2010.

[34] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[35] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

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

[37] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 398–407. IEEE, 2010.

[38] Andrew David Konwinski. Multi-agent cluster scheduling for scalability and flexibility. 2012.

[39] Marcel Kornacker and Justin Erickson. Cloudera impala: real-time queries in apache hadoop, for real, 2012.

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

[41] Hugh C Lauer and Roger M Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

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

[43] John CS Lui, Vishal Misra, and Dan Rubenstein. On the robustness of soft state protocols. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 50–60. IEEE, 2004.

[44] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[45] David Meisner, Brian T Gold, and Thomas F Wenisch. Powernap: eliminating server idle power. *ACM SIGARCH Computer Architecture News*, 37(1):205–216, 2009.

[46] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[47] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[48] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. The 500 most powerful commercially available computer systems, 2014.

[49] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[50] Jan Nyström and Bengt Jonsson. Extracting the processes structure of erlang applications. 2001.

[51] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 14–14. USENIX Association, 2013.

[52] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[53] Mayur R Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64. ACM, 2008.

[54] Gahyun Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 297–298. ACM, 2011.

[55] Gahyun Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 297–298. ACM, 2011.

[56] Gregory F Pfister. *In search of clusters.* Prentice-Hall, Inc., 1998.

[57] 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 Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[58] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, 2011.

[59] Sherif Sakr, Anna Liu, and Ayman G Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46(1):11, 2013.

[60] 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*, pages 351–364. ACM, 2013.

[61] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[62] Thomas Lawrence Sterling. *Beowulf cluster computing with Linux.* MIT press, 2002.

[63] Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009:4, 2009.

[64] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[65] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[66] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 25. ACM, 2012.

[67] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[68] Christoph Woskowski, Mikolaj Trzeciecki, and Florian Schwedes. Robust by" let it crash". In *Safecomp 2013 FastAbstract*, page NC, 2013.

[69] Gang Yang, Kaibo Wang, and Xingshe Zhou. An adaptive resource monitoring method for distributed heterogeneous computing environment. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 40–44. IEEE, 2009.

[70] Chee Shin Yeo. Utility-based resource management for cluster computing, 2008.

[71] 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*, pages 265–278. ACM, 2010.

[72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[73] 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*, pages 10–10, 2010.

[74] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: parallel databases meet mapreduce. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(5):611–636, 2012.

Appendix A

DETAILED VALUES OF THE THROUGHPUT EXPERIMENT

Table A.1: Sparrow's results for different initial loads.

(a) For 70% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	11.051ms	11.157ms	11.231ms	11.330ms
10 jobs	11.183ms	11.349ms	11.529ms	11.772ms
15 jobs	11.268ms	11.580ms	11.860ms	12.185ms
20 jobs	11.432ms	11.823ms	12.188ms	12.594ms

(b) For 75% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	12.241ms	12.475ms	12.638ms	12.893ms
10 jobs	12.481ms	12.998ms	13.473ms	13.992ms
15 jobs	12.766ms	13.526ms	14.307ms	15.105ms
20 jobs	13.148ms	14.158ms	15.201ms	16.373ms

(c) For 80% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	14.474ms	14.923ms	15.418ms	15.888ms
10 jobs	15.129ms	16.093ms	16.932ms	17.852ms
15 jobs	15.709ms	16.983ms	18.717ms	20.346ms
20 jobs	16.204ms	18.221ms	20.298ms	22.760ms

(d) For 85% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	19.410ms	19.916ms	20.954ms	21.363ms
10 jobs	20.407ms	21.833ms	23.875ms	25.367ms
15 jobs	20.912ms	23.874ms	26.943ms	30.411ms
20 jobs	22.138ms	25.892ms	30.780ms	36.074ms

(e) For 90% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	28.683ms	29.862ms	31.678ms	33.108ms
10 jobs	30.960ms	33.989ms	37.144ms	40.886ms
15 jobs	32.197ms	37.370ms	43.063ms	49.602ms
20 jobs	34.589ms	41.124ms	50.131ms	59.986ms

Table A.2: MACRM's results for different initial loads.

(a) For 70% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	12.299ms	12.428ms	12.541ms	12.681ms
10 jobs	12.478ms	12.726ms	13.009ms	13.263ms
15 jobs	12.610ms	13.012ms	13.459ms	13.870ms
20 jobs	12.792ms	13.359ms	13.915ms	14.489ms

(b) For 75% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	12.843ms	13.034ms	13.321ms	13.563ms
10 jobs	13.123ms	13.620ms	14.186ms	14.681ms
15 jobs	13.395ms	14.181ms	14.983ms	15.807ms
20 jobs	13.764ms	14.763ms	15.942ms	17.109ms

(c) For 80% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	13.807ms	14.148ms	14.632ms	14.975ms
10 jobs	14.342ms	14.986ms	15.911ms	16.835ms
15 jobs	14.785ms	16.170ms	17.598ms	18.925ms
20 jobs	15.307ms	17.249ms	19.261ms	21.280ms

(d) For 85% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	15.253ms	16.118ms	16.523ms	17.083ms
10 jobs	16.225ms	17.327ms	19.054ms	20.219ms
15 jobs	16.735ms	18.867ms	21.408ms	24.366ms
20 jobs	17.756ms	21.130ms	24.304ms	28.401ms

(e) For 90% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	14.548ms	14.704ms	14.817ms	14.973ms
10 jobs	14.723ms	15.010ms	15.275ms	15.531ms
15 jobs	14.906ms	15.312ms	15.707ms	16.138ms
20 jobs	15.043ms	15.624ms	16.138ms	16.698ms

Table A.3: YARN's results for different initial loads.

(a) For 70% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	19.240ms	26.479ms	33.719ms	40.958ms
10 jobs	27.419ms	40.838ms	54.257ms	67.677ms
15 jobs	35.572ms	55.144ms	74.716ms	94.288ms
20 jobs	43.758ms	69.517ms	95.275ms	121.033ms

(b) For 75% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	21.107ms	29.795ms	38.482ms	47.170ms
10 jobs	30.567ms	46.714ms	62.861ms	79.008ms
15 jobs	40.013ms	63.606ms	87.200ms	110.793ms
20 jobs	49.526ms	80.632ms	111.738ms	142.844ms

(c) For 80% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	24.029ms	34.889ms	45.748ms	56.607ms
10 jobs	35.372ms	55.574ms	75.776ms	95.978ms
15 jobs	46.715ms	76.260ms	105.804ms	135.349ms
20 jobs	58.158ms	97.145ms	136.133ms	175.120ms

(d) For 85% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	28.199ms	42.678ms	57.157ms	71.636ms
10 jobs	42.632ms	69.543ms	96.455ms	123.367ms
15 jobs	57.078ms	96.435ms	135.793ms	175.150ms
20 jobs	71.657ms	123.593ms	175.530ms	227.467ms

(e) For 90% load.

	10 tasks/job	**20 tasks/job**	**30 tasks/job**	**40 tasks/job**
5 jobs	37.219ms	58.937ms	80.656ms	102.374ms
10 jobs	57.757ms	98.015ms	138.272ms	178.530ms
15 jobs	78.323ms	137.146ms	195.969ms	254.792ms
20 jobs	100.055ms	177.610ms	255.165ms	332.720ms