Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

DOCTORAL THESIS

Optimizing the reliability and resource efficiency of MapReduce-based systems

Author
Bunjamin Memishi

Supervisors
María de los Santos Pérez Hernández
Gabriel Antoniu

January 2016

Abstract

Due to the increase of huge data volumes, a new parallel computing paradigm to process big data in an efficient way has arisen. Many of these systems, called data-intensive computing systems, follow the Google MapReduce programming model. The main advantage of these systems is based on the idea of sending the computation where the data resides, trying to provide scalability and efficiency.

In failure-free scenarios, these frameworks usually achieve good results. However, these ones are not realistic scenarios. Consequently, these frameworks exhibit some fault tolerance and dependability techniques as built-in features. On the other hand, dependability improvements are known to imply additional resource costs. This is reasonable and providers offering these infrastructures are aware of this. Nevertheless, not all the approaches provide the same tradeoff between fault tolerant capabilities (or more generally, reliability capabilities) and cost.

In this thesis, we have addressed the coexistence between reliability and resource efficiency in MapReduce-based systems, looking for methodologies that introduce the minimal cost and guarantee an appropriate level of reliability. In order to achieve this, we have proposed: (i) a formalization of a failure detector abstraction; (ii) an alternative solution to single points of failure of these frameworks, and finally (iii) a novel feedback-based resource allocation system at the container level.

Finally, our generic contributions have been instantiated for the Hadoop YARN architecture, which is the state-of-the-art framework in the data-intensive computing systems community nowadays. The thesis demonstrates how all our approaches outperform Hadoop YARN in terms of reliability and resource efficiency.

Resumen

Debido al gran incremento de datos digitales que ha tenido lugar en los últimos años, ha surgido un nuevo paradigma de computación paralela para el procesamiento eficiente de grandes volúmenes de datos. Muchos de los sistemas basados en este paradigma, también llamados sistemas de computación intensiva de datos, siguen el modelo de programación de Google MapReduce. La principal ventaja de los sistemas MapReduce es que se basan en la idea de enviar la computación donde residen los datos, tratando de proporcionar escalabilidad y eficiencia.

En escenarios libres de fallo, estos sistemas generalmente logran buenos resultados. Sin embargo, la mayoría de escenarios donde se utilizan, se caracterizan por la existencia de fallos. Por tanto, estas plataformas suelen incorporar características de tolerancia a fallos y fiabilidad. Por otro lado, es reconocido que las mejoras en confiabilidad vienen asociadas a costes adicionales en recursos. Esto es razonable y los proveedores que ofrecen este tipo de infraestructuras son conscientes de ello. No obstante, no todos los enfoques proporcionan la misma solución de compromiso entre las capacidades de tolerancia a fallo (o de manera general, las capacidades de fiabilidad) y su coste.

Esta tesis ha tratado la problemática de la coexistencia entre fiabilidad y eficiencia de los recursos en los sistemas basados en el paradigma MapReduce, a través de metodologías que introducen el mínimo coste, garantizando un nivel adecuado de fiabilidad. Para lograr esto, se ha propuesto: (i) la formalización de una abstracción de detección de fallos; (ii) una solución alternativa a los puntos únicos de fallo de estas plataformas, y, finalmente, (iii) un nuevo sistema de asignación de recursos basado en retroalimentación a nivel de contenedores.

Estas contribuciones genéricas han sido evaluadas tomando como referencia la arquitectura Hadoop YARN, que, hoy en día, es la plataforma de referencia en la comunidad de los sistemas de computación intensiva de datos. En la tesis se demuestra cómo todas las contribuciones de la misma superan a Hadoop YARN tanto en fiabilidad como en eficiencia de los recursos utilizados.

Doctoral thesis committee:

Professor Antonio Pérez Ambite .....
Thesis Committee
Universidad Politécnica de Madrid, Spain

Professor Antonio García Dopico .....
Thesis Committee
Universidad Politécnica de Madrid, Spain

Professor Antonio Cortés Rosselló .....
Thesis Committee
Universitat Politècnica de Catalunya, Spain

Professor Alejandro Calderón Mateos .....
Thesis Committee
Universidad Carlos III de Madrid, Spain

Professor Luc Bougé .....
Thesis Committee
ENS Rennes, France

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

This chapter gives an overall description of the PhD thesis. First of all, we describe the motivation and context in which the thesis has been developed. Then, we define the problem addressed by this thesis, which is primarily fault-tolerant oriented. After that, we show the main objectives and the contributions derived from these objectives. In the next section we state the delimitations of the thesis, that is, the related issues that are out of its scope. Then, we describe the research methodology used in the thesis. Finally, we outline the rest of the thesis chapters.

1.1 Motivation

This thesis started as part of the Marie Curie Initial Training Network (MCITN) "SCALing by means of Ubiquitous Storage (SCALUS)" which aimed at elevating education, research, and development inside the storage area with a focus on cluster, grid, and cloud storage [94]. Arguing that the fault tolerance, due to its complexity, was an issue that had not been properly addressed in large-scale storage systems, this thesis focused on the development of strategies for providing enhanced fault tolerance capabilities to these distributed systems.

Due to the increase of huge data volumes, the computing systems community developed a new parallel computing paradigm to process big data in an efficient way. Data-intensive processing frameworks were arisen, taking Google's MapReduce [42] an important role, since it become the leading programming model. This framework and its alike models have covered many gaps from data processing requirements. However, after an exhaustive analysis, we concluded that fault-tolerance, although considered the strong side of these frameworks due to its fine-grain nature, can be enhanced and optimized.

1.2 Problem definition

The ever growing size of data (i.e., Big Data) has motivated the development of data intensive processing frameworks and tools. In this context, MapReduce [42] has become a relevant framework for Big Data processing in the clouds, thanks to its

remarkable features including simplicity, fault tolerance, and scalability. The popular open source implementation of MapReduce, Hadoop [11], was developed primarily by Yahoo!, where it processes hundreds of Terabytes of data on at least 10,000 cores, and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [10].

Undoubtedly, failure is a part of everyday life, especially in current data-centers which comprise thousands of commodity hardware and software [32, 84, 88]. Consequently, MapReduce was designed with hardware failure in mind. In particular, Hadoop tolerates machine failures (crash failures) by re-executing all the tasks of the failed machine by the virtue of data replication. Furthermore, in order to mask temporary failures caused by network or machine overload (timing failure) where some tasks are performing relatively slower than other tasks, Hadoop re-launches other copies of these tasks on other machines.

Foreseeing MapReduce usage in the next generation Internet [78], a particular concern is the aim of improving the MapReduce's reliability by providing better fault tolerance mechanisms. While the handling and recovery in MapReduce fault-tolerance via data replication and task re-execution seem to work well even at large scale [68, 14, 115], there is relatively little work on detecting failures in MapReduce. Accurate detection of failures is as important as failures recovery, in order to improve applications latencies and minimize resource waste. A new methodology to adaptively tune the timeout detector can significantly improve the overall performance of the applications, regardless of their execution environment. Every MapReduce job should have its proper timeout, because in this way it could be possible to efficiently detect failures. Due to this, we raise the following research question:

Research question 1. Is it possible to formalize a failure detector abstraction in order to model the timing assumptions in MapReduce-based systems?

Apart from the enhancement of the timeout, a major issue of the MapReduce environments is its dependence on daemons, which can become single points of failure. For instance, this is the case of the master. Every MapReduce execution is based on a special node, called master. The rest of the nodes are called workers. In Hadoop MapReduce 1.0, the master node, also called JobTracker, has special relevance in a MapReduce framework, since it is in charge of keeping several data structures, like the state and the identity of the worker nodes. The master node is also responsible for scheduling the tasks of a job, distribute these tasks among the workers, monitoring the tasks and re-executing them, if it is needed.

In the first version of the Hadoop MapReduce framework, the master was a single point of failure [22]. In order to enhance the reliability of this framework, a new architecture was proposed, YARN [101]. This project states that Hadoop YARN is only a resource management platform, that among other features provides greater scalability, higher efficiency and enables different frameworks to efficiently share a cluster [101]. In other words, this means that MapReduce is only one of the frameworks that run on top of YARN. While getting rid of the single JobTracker for all

the jobs, now each job has its own JobTracker, placed in the cluster as any other TaskTracker. However, the problem of a single point of failure is not solved, since a JobTracker has become a single point of failure for a particular job, which in some scenarios could also be important.

We think that it is possible to increase the reliability of YARN, and particularly the reliability of masters in YARN by means of a novel approach that goes beyond existing state-of-the-art approaches, such as standby masters and periodical checkpointing. Due to this, we raise the following research question:

Research question 2. Is it possible to extend or re-design the failure handling model for MapReduce single point of failure, a model that goes beyond existing methodologies?

When reliability is improved, it is reasonable to think that these improvements are made at the expense of additional resource consumption. For instance, the replication improves the reliability, but increases the cost, regardless of the kind of replicated data, final, intermediate or hotspot data. The same occurs with cloning or speculating a task, which implies the increase of the resource cost. It is very difficult to improve reliability and maintain the same resource utilization. Due to this, we raise the final question:

Research question 3. Can MapReduce reliability be improved by considering the minimal cost increase, by not increasing the cost at all, or even decreasing it?

This thesis is focused on answering these three research questions, which revolve around the reliability and resource efficiency of MapReduce systems.

1.3 Objectives

The previous research questions can be transformed into objectives to be achieved by this thesis. Namely, the four goals of this thesis, and at the same time, their four contributions, are:

- Analysis of the failure detection drawbacks and other fault-tolerant bottlenecks in MapReduce-based systems that could be object of improvement. Regarding this, we have (i) statistically and experimentally analyzed the failure detector shortcomings in different MapReduce workloads; (ii) we have shown that failure handling on single points of failure is still deficient in large-scale environments, and (iii) we have proven that data-intensive frameworks are not able to efficiently re-distribute resources at container level.

- Led by the Research question 1, we aim to formalize a failure detector abstraction for MapReduce-based systems, by introducing alternative timeouts, different from the default and static timeout of current frameworks. Regarding

this, initially (i) we propose the *high relax failure detector* (HR-FD), which represents a static timeout as alternative to the default timeout. This algorithm is able to estimate a completion time of the user workload, and adjust a unique timeout for it. This static adjustment is particularly efficient for small workload requests, since most of the user request of production clouds run these respective requests. Second, (ii) we propose *medium relax failure detector* (MR-FD), which represents the fundamental module behind our dynamic timeout service. The advantage of this module consists in its ability to change the timeout with respect to the progress score of each user workload; as long as the user workload comes into the finish state, the timeout gets smaller, in order not to harm the completion time performance. Finally, (iii) we propose the *low relax failure detector* (LR-FD), which is able to intersect the MapReduce dynamic timeout with an external monitoring system, which enforces more accurate failure detections. This last module is particularly important for some of the user requests which are strictly deadline-bounded, by consisting of higher accuracy requirements.

- Led by the Research question 2, we aim to formalize an alternative, but general failure handling model for any MapReduce-based single point of failure. Regarding this, we have proposed a novel failure handling framework, called Diarchy. Diarchy differs from classical and costly standby and checkpointing methodologies. Its principal aim is to enhance the MapReduce reliability, by means of the sharing of responsibilities between two master peers. Apart from the main Diarchy algorithm, we instantiate a case study with respect to its functioning, and experimentally address the application master single point of failure within Hadoop YARN.

- Led by the Research question 3, we aim to formalize an alternative and optimized resource allocation model for MapReduce-based systems, which is capable of adjusting proper containers for any workload request. Containers represent an encapsulation of a subset of computing resources, placed on a single node of a cluster. They are becoming the de facto standard as resource allocation facility. However, their current static configuration leaves huge room for improvement. Regarding this, we propose AdaptCont, a feedback system based approach, which is in charge of selecting the right amount of resources for any container request. This selection is dependent on many parameters, among others, the real-time request input, the number of requests, the number of users and the dynamic constraints of the system infrastructure, such as the set of resources available. AdaptCont can function in two modes, Dynamic AdaptCont and Pool AdaptCont. Unlike Dynamic AdaptCont, which calculates the exact amount of resources per each container, Pool AdaptCont chooses a predefined container from a pool of available configurations. In addition, we instantiate AdaptCont for a particular case study, the application master container of Hadoop YARN.

1.4 Delimitations

This thesis belongs to the area of data-intensive frameworks, a parallel computing paradigm that has grown in the last decade, due to the increase of huge data volumes. Although these frameworks are known as workhorses of the current production clouds, our contributions go beyond these specific infrastructure environments and can be applied to other infrastructures, such as clusters, grids or clouds.

All the thesis contributions try to improve the fault-tolerance mechanisms of these frameworks, with a reasonable resource efficiency. Regarding fault tolerance, we have addressed crash-stop and omission failures. Arbitrary (byzantine) failures, network failures, and security failures are out of the scope of the thesis.

On the cloud computing context, especially in scenarios where a user or a provider needs to access data, it is likely that privacy and security issues appear. Apart from ensuring high availability and reliability, privacy and security are considered first-class citizens of these environments. Nevertheless, these topics are also out of the scope of this thesis.

1.5 Methodology

In the early phase of this doctoral thesis, we have followed an exploratory research methodology. In other words, the research objectives of the thesis were not known beforehand, but extracted from the study of the state of the art through different research iterations.

Initially, the first scientific iteration was originated from the goal of enabling a *High-performance, Secure and Fault Tolerant Grid File System*. With respect to this issue, we systematically surveyed most of the literature in this direction, and identified the first set of problems which have not been solved yet.

The shift from grid to cloud environments motivated the second research iteration. In this second stage, we focused on a particular cloud-based framework, i.e., data-intensive computing systems. Concretely, we concentrated on the performance drawbacks of MapReduce-based systems when they are exposed to failures.

The third iteration constituted the definition of the first contribution of the thesis, derived from the identification of the necessity of a formal failure detector abstraction model within MapReduce-based systems. Regarding this first contribution, we applied a method based on modeling the system before formalizing the failure detector abstractions. As a result of this iteration, we developed three algorithms, which were evaluated by means of two metrics: the correctness and the performance of the mechanisms. The first metric evaluation is decomposed in the assessment of two other additional metrics: completeness and accuracy. The performance is evaluated by means of simulation.

In this stage, we identified the other two research problems. The second problem dug into the issue of single points of failure, present in MapReduce-based systems. For this contribution we made a statistical analysis of the impact of a single point of failure in MapReduce. Then, we defined an algorithm, which was again evaluated

from a probabilistic point of view. Finally, we made an experimental evaluation based on simulation.

On the other hand, the third problem was related to the inappropriate use of resources and its relation to the reliability of MapReduce frameworks. For this contribution we have followed a feedback-based approach, applied to a paradigmatic use case in YARN, the application master. The experimental evaluation has been made again by means of a round-based simulator.

1.6 Thesis outline

This dissertation is organized into three remaining parts: preliminaries, contributions and conclusions sections.

The preliminaries include two chapters. Chapter 2 explains the basic theoretical terminology in data-intensive frameworks, focusing on explaining the MapReduce approach, and its fault-tolerant mechanisms. Chapter 3 gives a general literature review in data-intensive frameworks, with particular emphasis on the contributions regarding their reliability improvements.

The contributions are described in three chapters. Chapter 4 is dedicated to formalize the failure detector mechanism in MapReduce-based systems, with particular focus on the omission failures. Chapter 5 introduces a novel failure handling framework that enhances the MapReduce reliability, by means of the sharing of responsibilities between two master peers. Chapter 6 introduces a novel resource allocation framework, that enhances the MapReduce resource utilization and reliability, by means of feedback-based resource allocation approach at the container level.

Finally, Chapter 7 summarizes the contributions, and highlights the future lines and the main publications of this dissertation.

Part II

Context

Chapter 2

Background

This chapter gives the essential theoretical preliminaries in order to easily follow the ongoing part of the thesis. Just at the beginning, we define the concept of reliability, and its importance in distributed systems. After giving a general overview about the data-intensive frameworks, we focus on the definition of the MapReduce framework. After this, we briefly describe its wide-spread implementation, Apache Hadoop, and its next-generation MapReduce implementation, commonly known as Hadoop YARN (Yet Another Resource Negotiator), which are classified as Hadoop MapReduce *1.0* and Hadoop MapReduce *2.0*, respectively. Furthermore, an important part is dedicated to fault-tolerant concepts of MapReduce, and their implementation mechanisms. Finally, we summarize the potential of this thesis proposal.

2.1 Reliable distributed systems

The key concept of a distributed system is the network, which enables communication and coordination between the machines of such a distributed system by means of exchanging messages. In comparison to a centralized system, distributed systems provide well know advantages, such as reliability, scalability and, in some contexts, performance enhancement. However, distributed systems also have many disadvantages, being the complexity of these systems one of the most noticeable. This was clearly stated by Leslie Lamport, a foundational researcher in the theory of distributed computing, who said that "a distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable".

In computer science, dependability is an already matured field. It is mainly considered as a property of a computer system that delivers what is intended to deliver. Avizienis et al., in [24], explain in detail this property through attributes, threats, and means. One of the most important attribute of dependability is the *reliability*. Reliability studies the continuity of a correct service. In other words, it represents the time function probability $R(t)$ that a specific computer system does not fail in a predicted time interval t, considering that the system was correctly functioning at the initial time $t = 0$ [91].

Reliability is ensured by means of different fault-tolerant mechanisms, such as

software/hardware replication [86, 57, 56, 54], state checkpointing [34, 47], agreement protocols [87, 69, 52, 55, 70], etc. The basis of these implementations is mostly guided and dependent by these fault-tolerant levels or steps: monitoring, detection, handling, and recovery. According to the system requirements, all these levels could work concurrently, or in sequence.

Reliability is an important attribute in both centralized systems and distributed systems. It is very common to measure the reliability attribute with the system uptime or downtime. Whereas the system uptime refers to the time period during which a system is available (operational), the downtime concept refers to the time period when a system in unavailable. The longer the uptime is, the better the system behaves. In mission-critical system infrastructures, e.g., power plants, military and air traffic control systems, etc, whose correct and consistent functioning is vital for human lives, the reliability is even more important and its downtime requirement is stricter.

2.2 Data-intensive processing frameworks

The emergence of very complex computing problems has required applying approaches which involve the division of the problem into smaller tasks. When these tasks are executed in a concurrent (or parallel) manner, we can speak about parallel computing.

Due to the increase of data volumes in current applications [27], data-intensive computing has become one of the most popular forms of parallel computing. Its main goal is to process large amounts of data in parallel. It is worth mentioning the difference between data-intensive versus compute-intensive parallel approaches. Whereas the former concept is linked to frameworks whose primary goal is devoted to I/O processing on large volumes of data, the later concept is primarily devoted to high computational requirements on small volumes of data.

These data-intensive frameworks are very relevant nowadays, due to the explosion of digital data we are living. This data expansion has mainly come from three sources: (i) scientific experiments from fields such as astronomy, particle physics, or genomics; (ii) data from sensors and (iii) citizens publications in channels such as social networks.

2.3 MapReduce programming model

The MapReduce programming model is one of the most widespread approaches of data-intensive computing. It represents a programming model for processing large data sets [42]. MapReduce has been discussed by researchers for more than a decade, including the database community. Even though its benefits have been questioned when compared to parallel databases, some authors suggest that both approaches have their own advantages, and there is not a risk that one could become *obsolete* [97]. MapReduce's advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs. Other advantages are simplicity, automatic parallelism and scalability. These features make MapReduce an

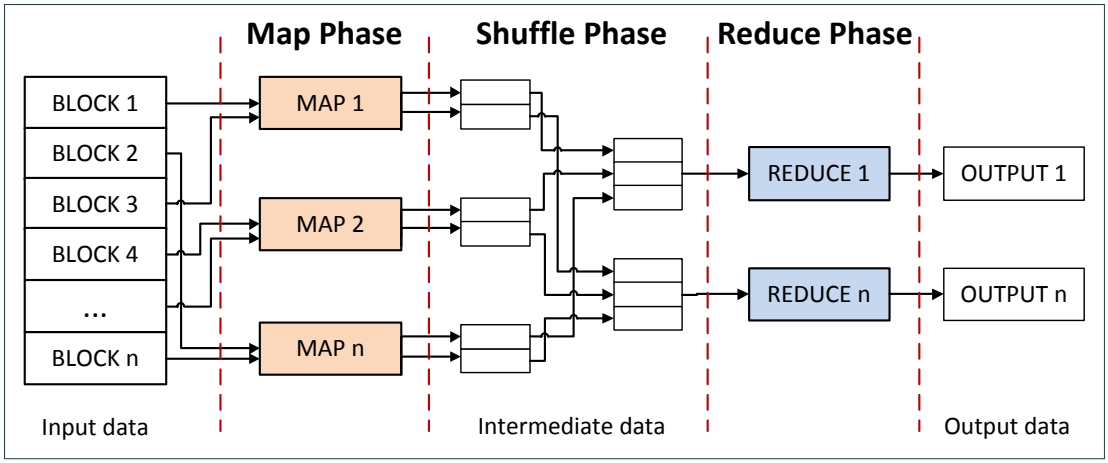


Figure 2-1: MapReduce logical workflow.

appropriate option for data-intensive applications, being more and more popular in this context. Indeed, it is used for different large-scale computing environments, such as Facebook Inc. [6], Yahoo! Inc. [12], and Microsoft Corporation [9].

By default, every MapReduce execution needs a special node, called *master*; the other nodes are called *workers*. The master keeps several data structures, like the state and the identity of the worker machines. Different tasks are assigned to the worker nodes by the master. Depending on the phase, tasks may execute two different functions: Map or Reduce. As explained in [42], users have to specify a Map function that processes a *key/value* pair to generate a set of intermediate *key/value* pairs, and a Reduce function that merges all intermediate values associated with the same intermediate key. In this way, many real world problems can be expressed by means of the MapReduce model.

A simple MapReduce data workflow is shown in Figure 2-1. This figure represents a MapReduce workflow scenario, from the input data to the output data. The most common implementations keep the input and output data in a reliable distributed file system, while the intermediate data is kept in the local file system at the worker nodes.

2.3.1 Hadoop MapReduce 1.0 versus Hadoop MapReduce 2.0

The most common implementation of MapReduce is part of the Apache Hadoop open-source framework [11]. Hadoop uses the Hadoop Distributed File System (HDFS) as the underlying storage backend, but it was designed to work on many other distributed file systems as well.

The main components of Apache Hadoop are MapReduce and HDFS. Hadoop MapReduce consists of a JobTracker and many TaskTrackers, which constitute the processing master and workers respectively. TaskTrackers consist of a limited number of slots for running map or reduce tasks. The MapReduce workflow is managed

by the JobTracker, whose responsibility goes beyond the MapReduce process. For instance, the JobTracker is also in charge of the resource management. Hadoop HDFS consists of a NameNode and many DataNodes, that is, the storage master and workers respectively. Whereas the NameNode manages the file system metadata, DataNodes hold a portion of data in blocks.

The traditional version of Hadoop (Hadoop MapReduce 1.0) has faced several shortcomings on large-scale systems, concerning scalability, reliability and availability. The YARN (*Yet Another Resource Negotiator*) project (Hadoop MapReduce 2.0) has recently been developed with the aim of addressing these problems [101].

As previously stated, in the classic version of Hadoop, the JobTracker handles both resource management and job scheduling. The key idea behind YARN is to separate concerns, by splitting up the major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate entities. In the new architecture, there is a global ResourceManager (RM) and per-application ApplicationMaster (AM). The ResourceManager and a per-node slave, the NodeManager (NM) compose the data-computation framework. The per-application ApplicationMaster is in charge of negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the progress of the tasks. The Resource Manager includes two components: a Scheduler and Application Manager. Whereas the Scheduler is in charge of resource allocation, the Application Manager accepts job submissions, and initiates the first job container for the job master (Application Master). This architectural change has as main goals to provide scalability and remove the single point of failure presented by the JobTracker. However, the resource scheduler, the application manager and the application master now become single points of failure in the YARN architecture.

2.3.2 Fault-tolerant mechanisms for MapReduce processing

In Figure 2-2 we show a big picture of the default fault-tolerant concepts and their mechanisms in MapReduce.

At the core of failure detection mechanism is the concept of heartbeat. Any kind of failure that is detected in MapReduce has to fulfill some preconditions, in this case to miss a certain number of heartbeats, so that the other entities in the system detect the failure. The classic implementation of MapReduce has no mechanism for dealing with the failure of the master, since the heartbeat mechanism is not used to detect this kind of failure. Workers send a heartbeat to the master, but the master's health is monitored by the cluster administrator. This person must first detect this situation, and then manually restart the master.

Because the worker sends heartbeats to the master, its eventual failure will stop this notification mechanism. From the worker side, there is a simple loop that periodically sends heartbeat method calls to the master; by default, this period has been adjusted to 3 seconds in most of the implementations. The master makes a checkpoint every 200 seconds, in order to detect if it has missed any heartbeats from a worker for a period of 600 seconds, that is, 10 minutes. If this condition is fulfilled, then a worker is declared as dead and removed from the master's pool of workers

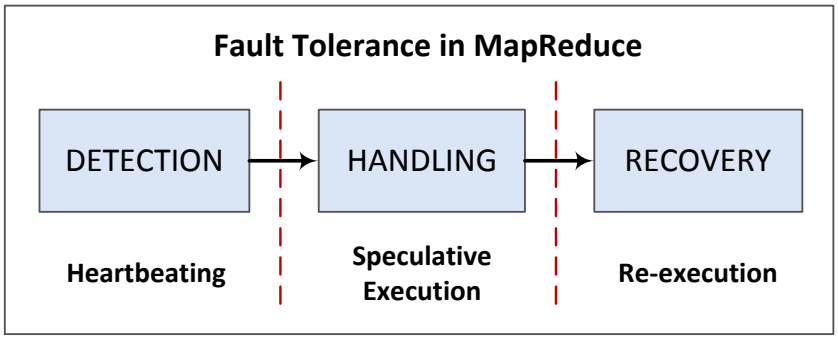


Figure 2-2: Fault tolerance in MapReduce: The basic fault tolerance definitions (detection, handling and recovery) with their corresponding implementations.

upon which can schedule tasks on. After the master declares the worker as dead, the tasks running on a failed worker are restarted on other workers. Since the map tasks that completed its work, kept their output on the dead worker, they have to be restarted as well. On the other hand, reduce tasks that were not completed need to be executed in different workers, but since completed reduce tasks saved its output in HDFS, their re-execution is not necessary.

Apart from telling to the master that a worker is alive, heartbeats also are used as a channel for messages. As a part of the heartbeat, a worker states whether it is ready to run a new task, and in affirmative case, the master will use the heartbeat return value for communicating the actual task to the worker. Additionally, if a worker notices that it did not receive a progress update for a task in a period of time (by default, 600 seconds), it proceeds to mark the task as failed. After this, the worker's duty is to notify the master that a task attempt has failed; with this, the master reschedules a different execution of the task, trying to avoid rescheduling the task on the same worker where it has previously failed.

The master's duty is to manage both, the completed and ongoing tasks on the worker to be re-executed or speculated, respectively. In the case of a worker failure, before the master decides to re-execute the completed and ongoing tasks so that may skip the default timeout of MapReduce (10 minutes), there is only one opportunity left, speculative execution.

The speculative execution is meant to be a method of launching another equivalent task as a backup, but only after all the normal tasks have been launched, and after the average running time of the other tasks. In other words, a speculative task is basically run for map and reduce tasks that have its completion rate below a certain percentage of the completion rate of the majority of running tasks.

An interesting dilemma is how to differentiate the handling and recovery mechanisms. A simple question arises: Does MapReduce differentiate between handling and recovery?

In some sense, both, speculative execution and re-execution try to complete a MapReduce job as soon as possible, with the least processing time, while execut-

ing its tasks on minimal resources (e.g., to avoid long occupation of resources by some tasks)[1]. However, the sequence of performing the speculative execution and re-execution is what makes them different, therefore considering the former one be part of the handling process, and the latter on part of the recovery. An additional difference to this is that, while the re-execution mechanism tries to react after the heartbeat mechanism has declared that an entity has failed, the speculative execution does not need the same timeout condition in order to take place; it reacts sooner.

Regarding to the nomenclature related to failures and errors, we consider a job failure when the job does not complete successfully. In this case, the first task that fails can be considered as an error, because it will request its speculation or re-execution from the master. A task failure can happen because the network is overloaded (in this case, this is also an error, because the network fault is active and loses some deliveries). In order to simplify this, we assume that in MapReduce, a task or any other entity is facing a failure, whenever it does not fulfill its intended function.

From the point of view of Hadoop's MapReduce, failures can happen in the master and worker. When the master fails, this is a single point of failure. But in the case of the worker, it may have a task fail (map or reduce task, or shuffle phase) or the entire worker. During a map phase, if a map task crashes, Hadoop tries to recompute it in a different tasktracker. In order to make sure that this computation takes place, most of reducers should complain for not receiving the map task output or the number of notifications is higher or equal to three [92]. The failed tasks have higher priority to be executed than the other ones; this is done to detect when a task fails repeatedly due to a bug and stop the job. In a reduce phase, a reduce task failure will have to be executed in a different tasktracker, having in mind that the three reduce phases should start from the beginning. The reduce task is considered as failed, if the majority of its shuffle attempts fails, the shuffle phase does not succeed to get five map outputs, or its progress stops for a long time. During the shuffle phase, a failure may also happen (in this case, a network failure), because two processes (in our case two daemons) can be in a working state, but a network failure may stop any data interchange between them. MapReduce implementations have been improved by means of the Kerberos authentication system, preventing a malicious reduce task from requesting another user's map output.

2.4 Summary

Data-intensive processing frameworks have facilitated the processing of large volumes of data. Nevertheless, the implementation of these programming models in large-scale clusters of commodity machines has arisen the importance of considering their reliability capabilities. This is especially necessary for production clouds that need to guarantee complex Service Level Agreements (SLAs) among its heterogeneous applications.

[1]This is not particularly true in the case of speculative execution, since it has proven to exhaust a considerable amount of resources, when executed on heterogeneous environments [115, 72] or when the system is going under failures [45]

The present chapter has briefly mentioned the basic functioning of reliable distributed systems, and then concentrated on describing the data-intensive processing frameworks, with particular emphasis on the MapReduce programming model, and its fault-tolerant mechanisms.

In order to place our proposals firmly in the field, and to familiarize the reader with similar contributions, we will present the state of the art in the next chapter.

Chapter 3

State of the art

This chapter gives a summary and analysis of the state of the art related to the main thesis contributions. First, we present a general analysis of the related work in data-intensive computing, and its well known applications. We then continue by analyzing the reliability optimizations of data-intensive applications, with particular emphasis on MapReduce. Part of this section covers the timeline of the most important Apache Hadoop software from a point of view of reliability customizations. Later, we discuss the state of the art in resource efficient optimizations of data-intensive applications, focusing this part again on MapReduce. At the end, we summarize the chapter.

3.1 Data-intensive processing frameworks

As mentioned before, MapReduce framework represents the de facto standard in the data-intensive computing community. However, there are many other projects, whose design and functionality differ from the basic MapReduce framework. Next, we present a collection of projects with significant impact in data-intensive computing.

3.1.1 Dryad/DryadLINQ

Knowing the benefits of Google's MapReduce, Microsoft designed its own data processing engine. In this way, Dryad [62] was introduced in 2007. After one year, Microsoft introduced a high level language system for Dryad, composed of LINQ expressions, and called it DryadLINQ [109].

Dryad represents a general-purpose distributed execution engine, whose main target is coarse-grain data-parallel applications. In order to form a dataflow graph, Dryad combines computational *vertices* with communication *channels*. An application is run in Dryad by executing the vertices of the graph on a set of available machines, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.

Whereas mainly inspired from the (i) graphic processing units (GPUs) languages, (ii) Google's MapReduce and (iii) parallel databases, Dryad is built also having in mind their disadvantages. As a consequence, Dryad as a framework allows the devel-

oper to have fine control over the communication graph, as well as the subroutines that live at its vertices. In order to describe the application communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared memory FIFOs) between the computation vertices, a Dryad application developer can specify an arbitrary directed acyclic graph (DAG). By directly specifying this kind of graph, the developer has also greater flexibility to easily compose basic common operations, leading to a distributed analogue of "piping" together traditional Unix utilities such as grep, sort and head.

Dryad graph vertices are enabled to use an arbitrary number of inputs and outputs. It is assumed that the communication flow determines each job structure. Consequently many other Dryad mechanisms (such as resource management, fault tolerance etc.) follow this pattern. A Dryad job is a directed acyclic graph where each vertex is a program and edges represent data channels. It is a logical computation graph that is automatically mapped onto physical resources by the runtime. At runtime each channel is used to transport a finite sequence of structured items.

Every Dryad job is coordinated by a master called "job manager" that runs either within the cluster or on a user's workstation, by having network access to the cluster. The job manager contains (i) the application-specific code, that allows to construct the job's communication graph, and (ii) library code, that allows to schedule the work across the available resources. Vertices transfer the data between them, therefore the job manager is only responsible for control decisions. DryadLINQ represents a very important extension of Dryad, since it is a set of language extensions and the corresponding system that can automatically and transparently compile SQL, MapReduce, Dryad and similar programs in a general-purpose language into distributed computations that can run on large-scale infrastructures. DryadLINQ does this in two ways, by (i) adopting an expressive data model of .NET objects; and (ii) by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language.

A DryadLINQ program is based on LINQ expressions that are sequentially run on top of datasets. The DryadLINQ main duty is to translate the parallelism portion of the program into a distributed execution, ready to be executed on the Dryad engine.

3.1.2 SCOPE

SCOPE is a scripting language for massive data analysis [31], also coming from Microsoft. Its design has a strong resemblance to SQL, which was intentionally decided. SCOPE is a declarative language. As in the case of MapReduce, it hides the complexity of the lower platform and its implementation.

A user SCOPE script runs the basic SCOPE modules, (i) compiler, (ii) runtime, and (iii) optimizer, before initiating the physical execution. In order to manipulate input and output, SCOPE provides respective customizable commands, which are, *extract* and *output*. The *select* command of SCOPE is similar to the SQL one, with the main difference that subqueries are not allowed. To solve this issue, a user should rewrite complex queries with outer joins.

Apart from the SQL functionalities, SCOPE provides MapReduce-alike commands,

which manipulate rowsets: *process*, *reduce*, and *combine*. The process command takes a rowset as input, and after processing each row, it outputs a sequence of rows. The reduce command takes a rowset as input, which has been grouped on the grouping columns specified in the ON clause. Then, it processes each group, and returns as output zero, one or multiple rows per group. The combine command takes two rowsets as input. It combines them depending on the requirements, and outputs a sequence of rows. The combine command is a binary operator.

Every SCOPE script resembles SQL, but its expression is implemented with C#, which needs to pass through the SCOPE compiler and optimizer, in order to be ready to run on parallel execution plan, which gets executed on the cluster as a Cosmos (Dryad) job.

According to different evaluation experiments, SCOPE demonstrates its powerful query execution performance, that scales in a linear manner with respect to the cluster and data sizes.

3.1.3 Nephele

In [104], authors present the basic foundations of Nephele, a novel research project at the time, whose aim was parallel data processing in dynamic clouds.

According to authors, state-of-the-art frameworks like MapReduce and Dryad are cluster-oriented models, which assume that their resources are a static set of homogeneous nodes. Therefore, these frameworks are not prepared enough for production clouds, whose exploitation of the dynamic resource allocation is a must. Based on this, they propose Nephele, a project which shares many similarities with Dryad, but providing more flexibility.

Nephele's architecture has a master-worker design pattern, with one Job Manager and many Task Managers. Each instance (aka VM) has its own Task Manager. As in Dryad, every Nephele job is expressed as a directed acyclic graph (DAG), where the vertices are tasks, and graph edges define the communication flow.

After writing the code for particular tasks, the user should define a Job Graph, consisting of linked edges and vertices. In addition, a user could specify other details, such as the number of subtasks in total, the number of subtasks per instance, instance types, etc. Each user Job Graph is then transformed into an Execution Graph by the Job Manager. Every specified manual configuration is taken into account by the Job Manager. Otherwise, the Job Manager places the default configuration according to the type of the respective job.

Compared to the default Hadoop on a small cloud infrastructure, the evaluation metrics are impressive and in favor to Nephele, showing better performance and resource utilization.

The main drawback of Nephele is that, due to its academic origin, it was not embraced by the research and industry community. One of the reasons could be its similarity with Dryad. An additional drawback of Nephele was its complexity, mainly compared to Hadoop MapReduce.

3.1.4 Spark

Spark is a novel framework for in-memory data mining on large clusters, whose main focus are applications that reuse the same dataset across multiple operations [112]. In this domain we found basically applications that are based on machine learning algorithms, such as text search, logistic regression, alternating least squares, etc. Spark programs are executed on top of the Mesos environment [59], where each of them needs its own driver (master) program to manage the control flow of the operations. Currently, Spark can also be run on top of other resource management frameworks, such as Hadoop YARN.

The main abstractions of Spark are:

- Resilient Distributed Datasets (RDDs) [111]. These are read-only collections of objects that are spread on cluster nodes.
- Parallel operations. These operations can be performed on top of the RDDs. Examples of these operations are *reduce*, *collect*, *foreach*, etc.
- Shared variables. These variables may be twofold: (i) broadcast variables, that copy the data value once to each worker; and (ii) accumulators, that can only "add" for being used as an associative operation, whose purpose (value) is readable by the driver only.

The most important abstraction of Spark are RDDs. Its primary use is to enable efficient in-memory computations on large-clusters. This abstraction evolves in order to solve the main issues of parallel applications, whose intermediate results are very important in future multiple computations.

The main advantages of RDDs are the efficient data reuse, which comes with a good fault tolerance support. Its interface is based on coarse-grained transformations, by applying the same operation in parallel to a large amount of data. Each RDD is represented through a common interface, consisting of:

- A set of partitions. These are atomic pieces of the dataset.
- Set of dependencies. These are dependencies on parent RDDs.
- A function for computing the dataset from its parent.
- Metadata about its (i) partitioning scheme, and (ii) data placement.

Examples of applications that can take advantage of this feature are iterative algorithms and interactive data mining tools. Spark shows great results on some of these applications, outperforming Hadoop by 10x [112, 111, 114]. As part of Spark, the research community has proposed different modules, such as D-Streams [114, 113], GraphX [53], Spark SQL [21], and many others.

D-Streams represents a stream processing engine, an alternative to live queries (or operators) maintained by distributed event processing engines. Authors argue that

it is better to have small batch computations by using the advantages of in-memory RDDs, instead of using long-lives queries which are more costly and complex, mainly in terms of fault tolerance.

GraphX is a graph processing framework, which is built on top of Spark. GraphX represents an alternative to the classical graph processing systems, because it can efficiently handle iterative processing requirements of graph algorithm, unlike the general-purpose frameworks, such as MapReduce. The advantage of GraphX with respect to the classical graph processing frameworks is that it enables wider range of computations, and preserves the advantages of general-purpose dataflow frameworks, mainly the fault tolerance.

Finally, Spark SQL is another Apache Spark module, which enables an efficient intersection between relational processing and Spark functional programming. It does this by introducing the (i) *DataFrame API*, which enables the execution of relational operations, and (ii) *Catalyst*, which is another module that optimizes queries, and in addition simplifies data sources additions, and optimization rules, among others.

The main idea behind Apache Spark is to use iterative queries that are main memory based, which is also its main drawback. If the user request is not related to the previous and recent RDDs, the query process should start from the beginning. In this scenario, if we have to go back to the first iteration, MapReduce usually performs better than Spark.

3.2 Reliability in data-intensive processing frameworks

Several projects have addressed different reliability issues in data-intensive frameworks, in particular for MapReduce. During this section, we have tried to collect those studies, adapting them to the most common failure type divisions in distributed systems [30, 90, 26]: crash, omission, arbitrary, network and security failures.

In [45], authors have evaluated Hadoop, demonstrating a large variation in Hadoop job completion time in the presence of failures. According to authors, this is because Hadoop uses the same functionality to recover from worker failure, regardless of the cause or failure type. Since Hadoop couples failure detection and recovery with overload handling into a conservative design with conservative parameter choices, it is often slow reacting to failures, exhibiting different response times under failure. Authors conclude that Hadoop makes unrealistic assumptions about task progress rates, re-discovers failures individually by each task at the cost of great degradation in job running time, and does not consider the causes of connection failures between tasks, which leads to failure propagation to healthy tasks.

In [29], authors have evaluated the performance and overhead of both the checkpointing-based fault-tolerance and the re-execution based fault tolerance in MapReduce through event simulation driven by Los Alamos National Labs (LANL) data. Regarding MapReduce, the fault tolerance mechanism which was explored is re-execution, where all map or reduce tasks from a failed core are reallocated dynamically to operational cores whether the tasks had completed or not (i.e., partial results are locally stored), and execution is repeated completely. In the evaluation of the performance of MapRe-

duce in the context of real-world failure data, it was identified that there is pressure to decrease the size of individual map tasks as the cluster size increases.

In [65], authors have introduced an analytical study of MapReduce performance under failures, comparing it to MPI. This research is HPC oriented and proposes an analytical approach to measure the capabilities of the two programming models to tolerate failures. In the MapReduce case, they have started with the principle that any kind of failure is isolated in one process only (e.g., map task). Due to this, the performance modeling of MapReduce was built on the analysis of each single process. The model consists of introducing an upper bound of the MapReduce execution time when no migration/replica is utilized, followed by an algorithm to derive the best performance when replica based balance is adopted. According to the evaluation results, MapReduce achieves better performance then MPI on less reliable commodity systems.

3.2.1 Crash failure

During a crash failure, the process crashes at time t and never recovers after that time. Since a crash failure involves process failing to finish its function according to its general definition, this means that in MapReduce a crash failure can lead to a node (machine), daemon (JobTracker or TaskTracker) or task (map, reduce) failure. These failures surge when a node simply crashes, and affects all of its daemons. But this is not only the case for a crash failure; there are many other cases when particular daemons or tasks crash due to Java Virtual Machine (JVM) issues, high overloads, memory or CPU errors, etc.

At this section point, we will summarize the master crash failures first, and then continue with other crash failures in MapReduce. An important contribution to the high availability of JobTracker is the work of Wang et al. [103]. Their paper proposes a metadata replication based solution to enable Hadoop high availability by removing single point of failure in Hadoop, regardless of whether it is NameNode or a JobTracker. Their solution involves three major phases:

- Initialization phase. Each standby/slave node is registered to active/primary node and its initial metadata (such as version file and file system image) are caught up with those of active/primary node.
- Replication phase. The runtime metadata (such as outstanding operations and lease states) for failover in future are replicated.
- Failover phase. Standby/new elected primary node takes over all communications.

A well known implementation of this contribution has been done by Facebook, Inc. [28], by creating the active and standby AvatarNode. This node is simply wrapped to the NameNode, and the standby AvatarNode takes the role of the active AvatarNode in less than a minute; this is because every DataNode speaks with both AvatarNodes all the time.

However, the above solution did not prove to give the optimum for the company requirements, since their database has grown by 2500x in the past four years. Therefore, another approach named Corona [48] was used. This time, for Facebook researchers it was obvious that they should separate the JobTracker responsibilities: resource management and job coordination. The cluster manager should look for cluster resources only, while a dedicated JobTracker is created per each job. As you can notice, at many points, the design decisions of Corona are similar to Hadoop YARN. Additionally, Corona has been designed to use push-based scheduling, as a major difference to the pull-based scheduling of the Hadoop MapReduce.

In the work [83], authors propose an automatic failover solution for the JobTracker to address the single point of failure. It is based on the Leader Election Framework [30], by using Apache Zookeper [5]. This means that multiple JobTrackers (at least three) are started together, but only one of them is the leader at a particular time. The leader does not serve any client, but receives periodical checkpoints from the remaining JobTrackers. If one of the NameNodes fails, the leader recovers its availability from the most recent checkpointed data. However, this solution within Yarn has not been explored for job masters [101] and only addresses other single points of failure, such as the resource manager daemon.

In case of a TaskTracker crash failure, its tasks are by default re-executed in the other TaskTrackers. This is valid for both, map and reduce tasks. Map tasks that completed on the dead TaskTracker are restarted because the job is still in the progress phase and did not finish yet, and contains n number of reduce tasks, which need that particular map output. Reduce tasks are re-executed as well, except for those reduce tasks that have completed, because they have saved its output in a distributed file system, that is, in HDFS.

MapReduce philosophy is based on the fact that a TaskTracker failure does not represent a drastic damage to the overall job completion, especially long jobs. This is motivated by large companies [41], which use MapReduce on a daily basis, and argue that even with a loss of a big number of machines, they have finished in a moderate completion time[1]. Any failure would simply speculate/re-execute the task in a different TaskTracker.

There are cases where TaskTrackers may be blacklisted by mistake from the JobTracker. In fact, this happens because the ratio of the number of the failed tasks in the respective TaskTracker is higher than the average failure rate on the overall cluster [105]. By default, the Hadoop's blacklist mechanism marks a TaskTracker as blacklisted if the number of tasks that have failed is more than four. After this, the JobTracker will stop assigning future tasks to that TaskTracker for a limited period of time. These blacklisted TaskTrackers can be brought to live, only by restarting them; in this way, they will be removed from the JobTracker's blacklist. The blacklisting issue could also go beyond this. This can be explained with one scenario. Let us assume that, at some point, reduce tasks that are running in the other TaskTrackers will try to connect to the failed TaskTracker. Some of the reduce tasks need the map

[1]Jeff Dean, one of the leading engineers in Google, said: (we) "lost 1600 of 1800 machines once, but finished fine."

output from the failed TaskTracker. However, as they cannot terminate the shuffle phase (because of the missing map output from the failed TaskTracker), they fail. Experiments in [45] show that reduce tasks die within seconds of their start (without having sent notifications) because all the conditions which declare the reduce task to be faulty become temporarily true when the failed node is chosen among the first nodes to connect to. In these cases, when most of the shuffles fail and there is little progress made, there is nothing left except re-execution, while wasting an additional amount of resources.

The main goal of the work presented in [40] is to represent a byzantine fault-tolerant (BFT) MapReduce runtime system that tolerates faults which are arbitrary. The idea behind the paper is doubling each task in execution. This means that if one of the tasks fails, the second backup task will finish on time, reducing the job completion time by using larger (intuitively, you may guess that doubling the tasks leads to approximately doubling the resources) amounts of resources.

A research paper presented in [75] describes Cloud MapReduce (CMR), a new fully distributed architecture to implement the MapReduce programming model on top of the Amazon cloud OS. The nodes are responsible for pulling job assignments and their global status in order to determine their individual actions. The proposed architecture also uses queues to shuffle results from map tasks to reduce tasks. Map tasks are meant to write results as soon as they are available and reduce tasks need to filter out results from failed nodes, as well as duplicate results. The preliminary results of the work indicate that CMR is a practical system and its performance is comparable to Hadoop. Additionally, from the experimental results it can be seen that the usage of queues that overlap the map and shuffle phase seems to be a promising approach to improve MapReduce performance.

In the early versions of Hadoop (including the Hadoop 0.20 version), a crash failure of the JobTracker involved that all active work was lost entirely when restarting the JobTracker. The next Hadoop version 0.21 gave a partial solution to this problem, making periodic checkpoints into the file system [11], so as to provide partial recovery.

In principle, it is very hard to recover any possible data after a TaskTracker's failure. That is why Hadoop's reaction is to simply re-execute the tasks in the other TaskTrackers. However, there are works which have tried to take the advantage of checkpointing [38], or saving the intermediate data in a distributed file system [67, 68].

Regarding [38], among the interesting aspects of the pipelined Hadoop implementation is that it is robust to the failure of both map and reduce tasks, introducing the "checkpoint" concept. It works on the principle that each map and reduce task notifies the JobTracker, which spreads or saves the progress, informing the other nodes about it. For achieving this, a modified MapReduce architecture is proposed that allows data to be pipelined between operators, preserving the programming interfaces and fault tolerance models of a full-featured MapReduce framework. This provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. The paper has also demonstrated the benefits for batch processing: by pipelining both within and across jobs, the proposed implementation can reduce the time to job completion. This study work can also be considered as an optional solution to an omission failure.

In [68], authors propose an intermediate storage system, with two features in mind: data availability and minimal interference. According to this paper, these issues are solved with ISS (intermediate storage system), which is based on three techniques:

- Asynchronous replication. This does not block the ongoing procedures of the writer. Moreover the strong consistency is not required when having in mind that in platforms like Hadoop and similar, there is a single writer and single reader for intermediate data.
- Rack-level replication. This technique is chosen, because of the higher bandwidth availability within a rack, taking into account that the rack switch is not heavily used as the core switch.
- Selective replication. It is used considering that the replication will be applied only to the locally-consumed data; in case of failure, the other data may be fetched again without problems.

This work is important to be mentioned, because for every TaskTracker failure, every map task that has been completed, it has already saved its output in a reliable storage system different from the local file system. In this way, the amount of redundant work for re-executing the map task that has been completed on the failed TaskTracker is reduced again.

3.2.2 Omission failure (stragglers)

An omission failure is a more general kind of failures. This happens when a process does not send (or receive) a message that it is supposed to send (or receive). In MapReduce terminology, omission failures are synonym for stragglers. Indeed, the concept of stragglers is very important in the MapReduce community, especially task stragglers, which could jeopardize the job completion time. Typically, the main causes of a MapReduce straggler task are: (i) a slow node, (ii) network overload and (iii) input data skew [18].

Most of the state of the art in this direction has intended to improve the job execution time, by means of doubling the overall small jobs [15], or just by doubling the suspected tasks (stragglers) through different speculative execution optimizations [42, 62, 115, 18, 35, 107].

In [115], authors have also proposed a new scheduling algorithm called Longest Approximate Time to End (LATE) to improve the performance of Hadoop in a heterogeneous environment, brought by the variation of VM consolidation amongst different physical machines, by preventing the incorrect execution of speculative tasks. In this work, authors try to solve the issue of finding the real stragglers[2] among the MapReduce tasks, so as to speculatively execute them, while giving them the deserved priority. As the node heterogeneity is common in the real-world infrastructures and

[2]It is important to mention that, differently from [115] which considers tasks as stragglers, in the default paper of Google [42], a straggler is "a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation."

particularly cloud infrastructures, the speculative execution in the default Hadoop's MapReduce implementation is facing difficulties to give a good performance. The paper proposes an algorithm which should in some way improve the MapReduce performance in heterogeneous environments. It starts giving some assumptions made by Hadoop, and how they are broken down in practice. Later on, it proposes the LATE algorithm, which is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. The paper has an extensive experimental evaluation, which proves the valuable idea implemented in LATE.

Mantri [18] is another important contribution related to omission failures, which are called outliers in this paper. The main aim of the contribution is to monitor and cull or relax the outliers, accordingly to their causes. Based on their research, outliers have many causes, but mainly are enforced by MapReduce data skew, crossrack traffic, and bad (or busy) machines. In order to detect these outliers, Mantri does not rely only on task duplication. Instead, its protocol enhances according to outlier causes. A real time progress score is able to separate long tasks from real outliers. Whereas the former tasks are allowed to be run, the real outliers are only duplicated when new available resources arise. Since the state-of-the-art contributions were mostly duplicating tasks at the end of the job, Mantri is able to make smart decision even before this, in case the progress score of the task is heavily progressing. Apart from data locality, Mantri places task based on the current utilization of network links, in order to minimize the network load and avoid self-interference among loads. In addition, Mantri is also able to measure the importance of the task output, and according to a certain threshold, it decides whether to recompute task or replicate its output. In general, the real-time evaluations and trace-driven simulations show Mantri to improve the average completion time for about 32%/

In [116], authors have proposed two mechanisms to improve the failure detection in Hadoop via heartbeat, but only in the worker side, that is, the TaskTracker. While the adaptive interval mechanism adjusts the TaskTracker timeout according to the estimated job running time in a dynamic way, the reputation-based detector compares the number of fetch-errors reported when copying intermediate data from the mapper and when any of the TaskTrackers reaches a specific threshold that TaskTracker will be announced as a failed one. As authors explain, the adaptive interval is advantageous to small jobs while the reputation-based detector is mainly intended to longer jobs.

GRASS [17] is another novel optimization framework, which is oriented to trimming the stragglers for approximation jobs. Approximation jobs are very common in the last period, because many domains are willing to have partial data in a specific deadline or error margin, instead of processing the entire data in an unlimited time or with 0% error margin. After the introduction of the MapReduce programming model, which came with a simple solution of speculative execution of slow tasks (stragglers), the research community proposed decent alternatives, such as LATE [115] or Mantri [18]. However, they were not meant to give near to optimal solution for the domain of approximation analytics. And this is the advantage of GRASS, which is basically formed of two algorithms:

1. Greedy Speculative Scheduling (GS). This algorithm is intended to greedily pick a task that will be scheduled next.

2. Resource Aware Speculative Scheduling (RAS). This algorithm is able to measure the cost of leaving an old task to run or schedule a new task, according to some important parameters (e.g. time, resources, etc.)

GRASS is a combination of GS and RAS.

Depending on the cluster infrastructure size, but also on other parameters, the scheduler could impose different limitations per user or workload. Among others, it is common to place a limit on the number of concurrent running tasks. The overall set of these simultaneous tasks per each user (or workload) is known as wave. If a GRASS job requires many waves, then it starts with RAS and finally, in the last two waves uses GS. If the jobs are short, it may use only GS. This switching is mostly dependent on:

- Deadline-error bound.
- Cluster utilization.
- Estimation accuracy for two parameters, t_{rem} (remaining time for and old job), and t_{new} (an estimated time for a new job).

Evaluations show that GRASS improves Hadoop and Spark, regardless of the usage of LATE or Mantri, by 47% and 38% respectively, in production workloads of Facebook and Microsoft Bing. Apart from approximation analytics, the speculative execution of GRASS also shows to be better for exact computations.

In [35], authors propose an optimized speculative execution algorithm called Maximum Cost Performance (MCP) that is characterized by:

- Apart from the progress rate, it takes into consideration the process bandwidth in a phase, in order to detect the slow tasks.
- It uses exponentially weighted moving average (EWMA), whose duty is to predict the process speed and also predict the task remaining time.
- It builds a cost-aware model that determines what task needs a backup based on the cluster load.

In addition, the MCP contribution is based on the disadvantages of previous contributions, which mainly rely on the task progress rate to predict stragglers, inappropriate reaction on input data skews scenarios, unstable cost comparison between the backup and ongoing straggler task, etc. Evaluation experiments on a small-cluster infrastructure show MCP to have 39% faster completion time and 44% improved throughput when compared to default Hadoop.

In [107], authors propose an optimized speculative execution algorithm that is oriented to solving a single-job problem in MapReduce. The advantage of this work

is that takes into account two cluster scenario, heavy and lightly loaded case. For the lightly loaded cluster, authors introduce two different speculative execution policies, early cloning, and later speculative execution based on the task progress rate. During the stage of heavily loaded cluster, the intuition is to use a later backup task. In this case, an Enhanced Speculative Execution (ESE) algorithm is proposed, which basically extends the work of [18]. Same authors have also introduced an additional extended work that assumes to work for multiple MapReduce jobs [108].

An important project related to Hadoop's omission failures is presented in [37]. In this work, authors have tried to build separate fault tolerance thresholds in the UpRight library for omission and commission failures, because omission failures are likely to be more common than commission failures. As we have mentioned before, during omission failures, a process fails to send or receive messages specified by the protocol. Commission failures exclude omission failures, including the failures upon which a process sends a message not specified by the protocol. Therefore, in the case of omission failures, the library can be fine-tuned in order to provide the liveness property (meaning that the system is "up") despite any number of omission failures.

The TaskTracker omission failures have also been addresses in some of the previous works we have mentioned [40, 38].

3.2.3 Arbitrary (byzantine) failure

The work discussing the omission failures in [37], is actually a wider review that includes the byzantine failures in general. The main properties upon which the UpRight library is based are:

- An UpRight system is safe ("right") despite r commission failures and any number of omission failures.
- An UpRight system is safe and eventually live ("up") during sufficiently long synchronous intervals when there are at the most u failures of which at most r are commission failures and the rest are omission failures.

The contribution of this paper is to establish byzantine fault tolerance as a viable alternative to crash fault tolerance for at least some cluster services rather than any individual technique. As authors say, much of their work involved making existing ideas fit well together, rather than presenting something new. Additionally, the performance is a secondary concern, with no claim that all cluster services can get low-cost BFT (byzantine fault tolerance).

The main goal of the work presented in [40] is to represent a BFT MapReduce runtime system that tolerates faults that corrupt the results of computation of tasks, such as the cases of DRAM and CPU errors/faults. These last ones cannot be detected using checksums and often do not crash the task they affect, but can only silently corrupt the result of a task. Because of this, they have to be detected and their effects masked by executing each task more than once. This BFT MapReduce follows the approach of executing each task more than once, but in particular circumstances. However, as the state machine approach requires $3f+1$ replicas to tolerate at the most

f faulty replicas, which gives a minimum of 4 copies of each task, this implementation uses several mechanisms to minimize both the number of copies of tasks executed and the time needed to execute them. In case there is a fault, from the evaluation results, it is confirmed that the cost of this solution is close to the cost of executing the job twice, instead of 3 times as the naive solution. Authors argue that this cost is acceptable for critical applications that require high level of fault tolerance. They introduce an adaptable approach for multi-cloud environments in [39].

In [106], authors propose another solution for commission failures called Accountable MapReduce. This proposal forces each machine in the cluster to be responsible for its behavior, by means of setting a group of auditors that perform an accountability test that checks the live nodes. This is done in real time, with the aim of detecting the malicious nodes.

3.2.4 Network failure

During a network failure, many nodes leave the Hadoop cluster; this issue has been discussed in different publications [72, 98, 36, 74], although for particular environments.

The work presented in [72] introduces a new kind of implementation environment of MapReduce called MOON, which is MapReduce on Opportunistic eNvironments. This MapReduce implementation has most of the resources coming from volunteer computing systems that form a Desktop Grid. In order to solve the resource unavailability, which is vulnerable to network failure, MOON supplements a volunteer computing system with a small number of dedicated compute resources. These dedicated resources keep a replica in order to enhance high reliability, maintaining the most important daemons, including the JobTracker. To enforce its design architecture, MOON differentiates files into reliable and opportunistic. Reliable files should not be lost under any circumstances. In contrast, opportunistic files are transient data that can tolerate some level of unavailability. It is normal to assume that reliable files have priority for being kept in dedicated computers, while opportunistic files are saved in these resources only when possible. In a similar way, this separation is also managed for read and write requests. MOON is very flexible in adjusting these features, based on the Quality of Service (QoS) needs. A reason for this is the introduction of a hibernate state and hybrid task scheduling. The hibernate state is an intermediate state whose main duty is to avoid having an expiry interval that is too long or short, which can incorrectly consider a worker node as dead or alive. A worker node enters in this state earlier than its expiry interval, and as a consequence it will not be supplied with further requests from clients. MOON changes the speculative execution mechanism by differentiating straggler tasks in frozen and slow lists of tasks, adjusting their execution based on the suspension interval, which is significantly smaller than the expiry interval. An important change to speculating tasks is their progress score, which divides the job into normal or homestretch. During the normal phase, a task is speculatively executed according to the default Hadoop framework; in a homestretch phase, a job is considered to have advanced toward its completion, therefore MOON tries to maintain more running copies of straggler tasks.

A later project similar to MOON is presented in [98]. Here, authors try to present a complete runtime environment to execute MapReduce applications on a Desktop Grid. The MapReduce programming model is implemented on top of an open source middleware, called BitDew [49], extending it with three main additional software components: the MapReduce Master, MapReduce worker programs and the MapReduce library (and several functions written by the user for their particular MapReduce application). Authors wanted to benefit from the BitDew basic services, in order to provide highly needed features in Internet Desktop Grid, such as "massive fault tolerance, replica management, barriers free execution, and latency-hiding optimization, as well as distributed result checking". The last point (distributed checking) is particularly interesting, knowing that result certification is very difficult for intermediate results which might be very large to send for verification on the server side. The introduced framework implements majority voting heuristics, even though it involves larger redundant computation.

The works presented in [36, 74] are related to cloud environments, with particular emphasis on Amazon cloud. They discuss the MapReduce implementation on environments consisting of Spot Instances (SIs)[3]

In [36], a simple model has been represented. This model calculates the n-step probability, the expected lifetime of a VM, and the cost of termination, that is, the amount of time lost compared to having the set of machines stay up until completion of the job. Using the spot instances, in cases when there is no fault, the completion time may be speed up. Otherwise, if there are failures, the job completion time may be longer than without using spot instances.

[74] is a more mature proposal than the previous work. Here authors have tried to prove that their implementation, called Spot Cloud MapReduce, can take full advantage of the spot market, proposed by Amazon WS. As the name suggests, this implementation has been built on top of Cloud MapReduce (CMR), with additional changes:

- Modifying the split message format in the input queues (adding a parameter which indicates the position in the file where the processing should start).
- Saving the intermediate work when a node is terminated.
- Changing the commit mechanism to perform a partial commit.
- Changing the way CMR determines the successful commit for a map split (electing a set of commit messages that is one more than the last key-value pair's offset).

The experimental evaluation shows that Spot CMR can work well in the spot market environment, significantly reducing cost by leveraging spot pricing.

[3]Spot instances are virtual machines resources in Amazon Web Services (WS), for which a user defines a maximum biding price that he/she is willing to pay. If there is no concurrence, the prices are lower and the possibility of using them is higher. But when the demand is higher, then Amazon WS has the right to stop your spot instances. If the spot instances are stopped by Amazon, the user does not pay, otherwise if the user decides to stop them before completing the normal hour, the user is obliged to pay for that consumption.

3.2.5 Security failure

The security concept is basically the absence of unauthorized access to, or handling of, system state [24]. This means that, authentication, authorization and auditing go hand in hand, in order to ensure a system security. Whereas authentication refers to the initial identification of the user, the authorization determines the user rights, after he or she has entered into the system. Finally, the audit process represents an official user inspection (monitoring) to check if the user behaves according to its role. In other words, we could equate these terms with the pronouns *who* (authentication), *what* (authorization), and *when* (audit).

MapReduce's security in Hadoop is strictly linked to the security of HDFS; as the overall Hadoop security is grounded in HDFS, this means that other services including MapReduce store their state in HDFS. While Google's MapReduce does not make any assumption on security [42], early versions of Hadoop assumed that HDFS and MapReduce clusters would be used by a group of cooperating users within a secure environment. Furthermore, any access restriction was designed to prevent unintended operations that could cause accidental data loss, rather than to prevent unauthorized data access [105, 73].

The basic security definitions that include authentication, authorization and auditing, were not present in Hadoop from the beginning. The authorization (managing user permissions) had been partially implemented. The auditing took place in the version 0.20 of Hadoop. The authentication was the last one, which came with Kerberos, an open-source network authentication protocol.

A user needs to be authenticated by the JobTracker before submitting, modifying or killing any job. Since Kerberos authentication is bi-directional, even the JobTracker authenticates itself to the user; in this way, the user will be assured that the JobTracker is reliable. Additionally, each task is seen as an individual user, due to the fact that tasks now are run from the client perspective, the one which submitted the job, and not from the TaskTracker owner. In addition, the JobTracker's directory is not readable and writable by everyone as it happens with the task's working directories. During the authentication process, each user is given a token (also called a ticket) to authenticate once and pass credentials to all the tasks of a job; the token's default lifetime is meant to be around 8 hours. While the NameNode creates these tokens, the JobTracker manages a token's renewal; token expiration is reasonably JobTracker dependent, in order not to expire prematurely for long running jobs.

At the same year when Kerberos was implemented in Hadoop, another proposal called Airavat [92] tried to ensure security and privacy for MapReduce computations on sensitive data. This work is an integration of mandatory access control (MAC) and differential privacy. MAC's duty is to assign security attributes to system resources, to constrain the interaction of subject with objects (e.g., subject can be a process, object can be a simple file). On the other side, differential privacy is a methodology which ensures that the aggregated computations maintain the integrity of each individual input. The evaluation of Airavat on several case studies shows flexibility in the maintenance of both accurate and private-preserving answers on runtimes within 32% of the default Hadoop's MapReduce.

Apart from the different improvements in Hadoop security[2, 105], the work for preventing the Hadoop cluster from eavesdropping failures, has been slow. The explanation from the Hadoop community was that encryption is expensive in terms of CPU and I/O speed [96].

At the beginning, the encryption over the wire was dedicated only to some socket connections. In the case of Remote Procedure Call (RPC), an important protocol for communication between daemons in MapReduce, its encryption was added only after the main security improvement in Hadoop (by integrating Kerberos [8]). Most of the other encryption improvements (for instance, the shuffle phase encryption) came in a very recent Hadoop version [1], taking into consideration that Hadoop clusters may also hold sensitive information.

3.2.6 Apache Hadoop reliability

Since its appearance in 2006, Apache Hadoop has undergone many releases [7]. Each of them has tried to improve different features of previous version, including fault tolerance. Table 3.1 shows Apache Hadoop 1.0 fault tolerance patches in a tree-like form, from the first Apache Hadoop 1.0 release (0.1.0) until release 1.2.1 which is the latest stable release up to the time of writing. These upgrades have played an important role in the later Hadoop evolution. An example of this is the introduction of speculative execution for reduce tasks, which caused many bugs in the previous days of its implementation. Therefore, the overall speculative execution mechanism was turned off by default later on, due to bugs in the framework. Actually the speculative execution mechanism was removed for some period, and later on, placed once again in the default functioning of the Apache Hadoop.

Year	Release	Patch
2006	0.1.0	The first release
	0.2.0	Avoid task re-run where it has previously failed (142); Don't fail reduce for a map impossibility allocation (169, 182); Five client attempts to JT before aborting a job (174); Improved heartbeat (186)
	0.3.0	Retry a single fail read, to not cause a failure task (311)
	0.7.0	Keep-alive reports, changed to seconds [10] rather than records [100] (556); Introduced killed state, to distinguish from failure state (560); Improved failure reporting (568); Ignore heartbeats from stale TTs (506)
	0.8.0	Make DFS heartbeats configurable (514); Re-execute failed tasks first (578)
	0.9.0	Introducing speculative reduce (76)
	0.9.2	Turn off speculative execution (827)
2007	0.10.0	Fully remove killed tasks (782)
	0.11.0	Add support for backup NNs, to get snapshotting (227, 959); Rack awareness added in HDFS (692)

Year	Release	Patch
	0.12.0	Change mapreduce.task.timeout to be per-job based (491); Make replication computation as a separate thread, to improve heartbeat in HDFS's NN (923); Stop assigning tasks to a dead TT (654)
	0.13.0	Distinguish between failed and killed task (1050); If nr of reduce tasks is zero, map output is written directly in HDFS (1216); Improve blacklisting of TTs from JTs (1278); Make TT expiry interval configurable (1276)
	0.14.0	Re-enable speculation execution by default (1336); Timed-out tasks counted as failures rather than killed (1472)
	0.15.0	Add metrics for failed tasks (1610)
2008	0.16.0	File permissions improvements (2336, 1298, 1873, 2659, 2431); Fine-grain control over speculative execution for map and reduce phase (2131); Heartbeat and task even queries interval, dependent on cluster size (1900); NN performance degradation from large heartbeat interval (2576)
	0.18.0	Completed map tasks should not fail if nr of reduce tasks is zero (1318)
	0.19.0	Introducing job recovery when JT restarts (3245); Add FailMon for hardware monitoring and analysis (3585)
2009	0.20.0	Improved blacklisting strategy (4305); Add test for injecting random failures of a task or a TT (4399); Fix heartbeating (4785, 4869); Fix JT (5338, 5337, 5394)
2010	0.20.202.0 (unreleased)	Change blacklist strategy (1966, 1342, 682); Greedily schedule failed tasks to cause early job failure (339); Fix speculative execution (1682); Add metrics to track nr of heartbeats by the JT (1680, 1103); Kerberos
2011	0.20.204.0	TT should handle disk failures by reinitializing itself (2413)
	0.20.205.0	Use a bidirectional heartbeat to detect stuck pipeline (724); Kerberos improvements
2012	1.0.2	A single failed name dir can cause the NN to exit (2702)
	1.1.0	Lower minimum heartbeat between TT and JT for smaller clusters (1906)
2013	1.2.0	Looking for speculative tasks is very expensive in 1.x (4499)
	1.2.1	The last stable release

Table 3.1: Apache Hadoop 1.0: timeline of its fault tolerance patches

The Hadoop community was very active at the beginning, but this changed drastically through the years. A crucial reason for this was the existence of parallel projects, which tested new proposed features, but that were in their early phases (alpha or beta). Finally, a new release, Apache Hadoop 2.0, widely known as Hadoop YARN, was created. Table 3.2 shows Apache Hadoop 2.0 fault tolerance patches in a tree-like form, from the first Apache Hadoop 2.0 release (0.23.0) until release 2.7.1,

which is the latest stable release up to the time of writing.

Year	Release	Patch
2011	0.23.0	The first release; Lower minimum heartbeat interval for TaskTracker (MR-1906); Recovery of MR AM from failures (MR-279); Improve checkpoint performance (HDFS-1458)
2012	0.23.1	NM disk-failures handling (MR-3121); MR AM improvements: job progress calculations (MR-3568), heartbeat interval (MR-3718), node blacklisting (MR-3339, MR-3460), speculative execution (MR-3404); Active nodes list versus unhealthy nodes on the webUI and metrics (MR-3760)
	0.23.3	Timeout for Hftp connections (HDFS-3166); Hung tasks timeout (MR-4089); AM Recovery improvement (MR-4128)
	0.23.5	Fetch failures versus map restart (MR-4772); Speculation + Fetch failures versus hung job (MR-4425); INFO messages quantity on AM to RM heartbeat (MR-4517)
	2.0.0-alpha	NN HA improvements: fencing framework (HDFS-2179), active and standy states (HDFS-1974), failover (HDFS-1973), standbyNode checkpoints (HDFS-2291, HDFS-2924), NN health check (HDFS-3027), HA Service Protocol Interface (HADOOP-7455), in standby mode, client failing back and forth with sleeps (HADOOP-7896); haadmin with configurable timeouts for failover commands (HADOOP-8236)
	2.0.2-alpha	Encrypted shuffle (MR-4417); MR AM action on node health status changes (MR-3921); Automatic failover support for NN HA (HDFS-3042)
2013	0.23.6	AM timing out during job commit (MR-4813)
	2.0.3-alpha	Stale DNs for writes (HDFS-3912); Replication for appended block (HDFS-4022); QJM for HDFS HA for NN (HDFS-3901, HDFS-3915, HDFS-3906); Kerberos issues (HADOOP-9054, HADOOP-8883, HADOOP-9070)
	2.1.0-beta	Reliable heartbeats between NN and DNs with LDAP (HDFS-4222); Tight DN heartbeat loop (HDFS-4656); Snapshots replication (HDFS-4078); Flatten NodeHeartbeatResponse (YARN-439); NM heartbeat handling versus scheduler event cause (YARN-365); NMTokens improvements (YARN-714, YARN-692); Resource blacklisting for Fifo scheduler (YARN-877); NM heartbeat processing versus completed containers tracking (YARN-101); AMRMClientAsync heartbeating versus RM shutdown request (YARN-763); Optimize job monitoring and STRESS mode versus faster job submission. (MR-3787); Timeout for the job.end.notification.url (MR-5066)

Year	Release	Patch
	2.1.1-beta	RM failure if the expiry interval is less than node-heartbeat interval (YARN-1083); AMRMClient resource blacklisting (YARN-771); AMRMClientAsync heartbeat versus runtime exception (YARN-994); RM versus killed application tracking URL (YARN-337); MR AM recovery for map-only jobs (MR-5468)
	2.2.0	MR job hang versus node-blacklisting feature in RM requests (MR-5489); Improved MR speculation, with aggressive speculations (MR-5533); SASL-authenticated ZooKeeper in ActiveStandbyElector (HADOOP-8315)
2014	2.3.0	SecondaryNN versus cache pools checkpointing (HDFS-5845); Add admin support for HA operations (YARN-1068); Added embedded leader election in RM (YARN-1029); Support blacklisting in the Fair scheduler (YARN-1333); Configuration to support multiple RMs (YARN-1232)
	2.4.0	DN heartbeat stucks in tight loop (HDFS-5922); Standby checkpoints block concurrent readers (HDFS-5064); Make replication queue initialization asynchronous (HDFS-5496); Automatic failover support for NN HA (HDFS-3042)
	2.4.1	Killing task causes ERROR state job (MR-5835)
	2.5.0	NM Recovery. Auxiliary service support (YARN-1757); Wrong elapsed time for unstarted failed tasks (YARN-1845); S3 server-side encryption (HADOOP-10568); Kerberos integration for YARN's timeline store (YARN-2247, HADOOP-10683, HADOOP-10702)
	2.6.0	Encryption for hftp. (HDFS-7138); Optimize HDFS Encrypted Transport performance (HDFS-6606); FS input streams do not timeout (HDFS-7005); Transparent data at rest encryption (HDFS-6134); Operating secure DN without requiring root access (HDFS-2856); Work-preserving restarts of RM (YARN-556); Container-preserving restart of NM (YARN-1336); Changed NM to not kill containers on NM resync if RM work-preserving restart is enabled (YARN-1367); Recover applications upon NM restart (YARN-1354); Recover containers upon NM restart (YARN-1337); Recover NMTokens and container tokens upon NM restart (YARN-1341, YARN-1342); Time threshold for RM to wait before starting container allocations after restart/failover (YARN-2001); Handle app-recovery failures gracefully (YARN-2010); Fixed RM to load HA configs correctly before Kerberos login (YARN-2805); RM causing apps to hang when the user kill request races with AM finish (YARN-2853)

Year	Release	Patch
	2.6.1 (unreleased)	Make MR AM resync with RM in case of work-preserving RM-restart (MR-5910); Support for encrypting Intermediate data and spills in local filesystem. (MR-5890); Wrong reduce task progress if map output is compressed (MR-5958)
2015	2.7.0	Block reports process during checkpointing on standby NN (HDFS-7097); DN heartbeat to Active NN may be blocked and expire if connection to Standby NN continues to time out (HDFS-7704); Active NN and standby NN have different live nodes (HDFS-7009); Expose Container resource information from NM for monitoring (YARN-3022); AMRMClientAsync missing blacklist addition and removal functionality (YARN-1723); NM fail to start with NPE during container recovery (YARN-2816); Fixed potential deadlock in RMStateStore (YARN-2946); NodeStatusUpdater cannot send already-sent completed container statuses on heartbeat (YARN-2997); Connection timeouts to NMs are retried at multiple levels (YARN-3238); Add configuration for MR speculative execution in MR2 (MR-6143); Configurable timeout between YARNRunner terminate the application and forcefully kill (MR-6263); Make connection timeout configurable in s3a. (HADOOP-11521)
	2.7.1	The last stable release

Table 3.2: Apache Hadoop 2.0: timeline of its fault tolerance patches

3.3 Resource efficiency in data-intensive processing frameworks

There are many contributions on data-intensive frameworks, whose goal is optimizing the MapReduce framework from different viewpoints. According to the title, at first, this section seems to be very narrow. We consider resource-efficient contributions all the state-of-the-art contributions which have tried to improve the resource utilization, but have maintained the completion time performance. However, there are other contributions of the state of the art, that have used additional resources in order to improve the completion time. This means that they have not really improved the resource utilization, because their priority has been the completion time. We also describe these approaches.

Most of these proposals belong to the area of improving the scheduling methodology with respect to data locality, dynamic resource allocation, autonomic parameters configuration, etc. Intentionally, we have not mentioned any contribution related to power efficiency. This is because our contributions have no direct connection with energy efficiency metrics, apart from side-effects of resource optimizations.

3.3.1 Data locality

In high performance computing environments, it is well known that computation takes the main role in the resolution of scientific problems. However, in data-intensive computing, the priority is given to data. Therefore, data-intensive processing framework usually transfer computations near the data in order to increase the throughput. In this way, data-intensive frameworks try to take advantage of data locality.

Among the first contributions in this direction is Quincy [63], which is a novel framework that addresses scheduling problems in environments where concurrent distributed jobs are norm. In order to do this, Quincy uses a min-cost flow network, where the edge weights encode the data locality issues and fair sharing, among others. Through evaluation experiments on a medium sized cluster infrastructure, authors have achieved to confirm Quincy as a good alternative when compared to the state of the art (mainly greedy algorithms), in terms of performance.

The contribution [110] works on the hypothesis of what to do when we have a shared Hadoop cluster between multiple users with different jobs over a common dataset. Authors present an algorithm called delay scheduling, which tries to combine two different features, fairness sharing and data locality, considering fairness sharing as the capability of a scheduler for distributing its resources among users (or workloads) in a fair manner (equally).

When the first concept of fair sharing is strictly followed, two problems appear:

- Head-of-line-scheduling, that is, very small jobs almost never consume data locality.

- Sticky slots, that is, when a cluster has large jobs, there is a tendency that a job uses the same slots repeatedly.

The delay scheduling algorithm proposes that a job waits for a limited amount of time, in order to get a node that has its data. Authors prove experimentally that by enabling this delay, they could bring locality close to 100%. Another crucial contribution to this paper is the Hadoop fair scheduler (HFS), which is based on the delay scheduling algorithm. HFS works on the principle of two-level scheduling hierarchy. At the top, it divides slots among pools by using weighted fair sharing. At the bottom, each user pool could allocate its slots with either FIFO or Fair sharing. In order to do task balancing, authors suggest to consider all jobs as long-task jobs, but change their status as short-task jobs if needed. Another issue are hot spots, which is the important data accessed by many users. Authors propose a kind of dynamic replication of these small, but hot files.

Another proposal called MapReduce Task Scheduler for Deadline Constraints (MTSD) algorithm that is also related to data locality issue is presented in [100]. The MSTD is actually composed of a node classification algorithm, whose aim is to improve the data locality of mappers and predict the remaining task execution time. These two modules are capable of allowing any user to specify a job request deadline and finishing its execution before the given deadline. It works on the principle of making different sets of nodes based on their performance, dynamically distributing

the workload on nodes with better performance, and estimating the remaining task progress rate for these respective nodes. According to the evaluation experiments, MTSD improves the data locality in comparison to the state of the art, demonstrates capability of meeting deadline constraints, and even improves the estimation of remaining task progress rate.

3.3.2 Dynamic resource allocation

A particular contribution related to hot files is Scarlett [14]. This paper addresses the problem of popular content in MapReduce, which may become a bottleneck when many users or jobs access its data concurrently. Due to this, authors present a novel approach called Scarlett, which replicates MapReduce input data, based on their access patterns. In addition, Scarlett distributes this data among machines and clusters, in order to avoid centralized hotspots. Production load analysis has enabled important lessons, such as knowing that concurrent access is an important metric to evaluate file popularity, or that larger files have the highest access number, etc.

Scarlett is based on two design choices:

- It replicates data on file granularity.
- It replicates these files based on predicted popularity.

In order to do this, it uses previous usage statistics, while the prediction is based on the most recent past (from 12h to 24h), and always considering the submitted jobs for execution in the queue. In order to minimize interference with running jobs, Scarlett puts a boundary of 10% as extra storage budget, which according to the experiments shows near to optimal results. Scarlett has an ability to replicate data lazily, by equally spreading replication traffic among machines of different rack[4], and using data compression as a good tradeoff between data processing (computational overhead) and network bandwidth. Simulations and experiments that are done in two MapReduce frameworks, Hadoop and Dryad, show that Scarlett removes efficiently hotspots and improves the job completion time by 20.2%, with 10% of extra storage and additional 1% of network resources.

In [64], an optimization system called Manimal was introduced. This system analyzes MapReduce programs by applying appropriate data-aware optimizations. The benefit of this *best-effort* system is that achieves to speedup these programs in an autonomic way, without human intervention.

FlexSlot [58] is a contribution to resize map slots of a workload and deliver additional slots, if needed, in order to improve the job execution time.

In [99], authors introduce DynamicMR, whose main contribution is to relax the slot allocation constraint between mappers and reducers, and modify the speculative execution mechanism in order to improve the performance efficiency. DynamicMR also addresses the data locality problem by scheduling slots in advance.

[4]While replicating, Scarlett reads from many sources.

In [71], authors introduce MRONLINE, which is able to configure relevant parameters of MapReduce online, by collecting previous statistics, and predicting the task configuration in fine-grain level.

[95] introduces MROrchestrator, whose main contribution is to dynamically detect resource bottlenecks, and according to this measurements, it can reallocate dynamic slot resources among workloads.

An automatic optimization of the MapReduce programs has been proposed in [25]. In this work, authors provide out of-the-box performance for MapReduce programs that need to be run using as input large datasets.

Cura [85] automatically creates an optimal cluster configuration for MapReduce jobs, by means of the framework profiling, reaching global resource optimization. In addition, Cura introduces a secure instant VM allocation to reduce the response time for the short jobs. Finally, it applies other resource management techniques such as cost-aware resource provisioning, VMaware scheduling and online VM reconfiguration. Overall, these techniques lead to enhance the response time and reduce the resource cost.

There are other optimizations which have been proposed from the database point of view, by optimizing the MapReduce queries [13], [50], or lower level optimizations, such as compiler optimizations [76].

3.4 Summary

As stated in this chapter, the MapReduce model has triggered many viewpoints from different communities. We have discussed the most relevant proposals for optimizing data-intensive frameworks reliability and resource efficiency. Many angles of these topics have been covered by the state of the art. However, as we described in Chapter 1, there are still many relevant research questions to be solved.

Among others, there is little work which has been dedicated to failure detection in MapReduce-based systems. Indeed, as far as we know, there is not any contributions which addresses a complete and heterogeneous infrastructure, covering different application profiles. Instead, every work establishes its own assumptions, solving this issue mainly by suspecting the slow tasks or doubling the entire set of tasks, whereas maintaining the static timeout. Being aware that a static timeout can jeopardize the workload performance of most requests, we define a general failure detector model, missing in the literature at the time of writing. Concretely we propose three fundamental modules, according to the time heterogeneity, that is, how much relaxation the system is capable to allow in detecting an omission failure. The first module leaves the static timeout for large jobs, adjusting the timeout for short jobs to the estimation of the job completion time. The second module adjusts dynamically the timeout, according to the progress score. The third module is oriented to deadline-bounded workloads, considering the timeout for the workers as an additional parameter.

Secondly, after analyzing the state-of-the-art methodologies for solving the single points of failure of MapReduce-based frameworks, we conclude that the transition from MapReduce 1.0 to MapReduce 2.0 did not solve this issue, except transforming

the problem nature from coarse-grained into fine-grained. Namely, the problems of scalability and fault tolerance of the JobTracker of the classic Hadoop version have given way to single points of failure in other entities of YARN, such as the resource scheduler, the application manager or the application master. The other proposed research alternatives did not solve either this problem. Due to this, we introduce an alternative solution to the single point of failure in these frameworks in order to increase the reliability of these systems.

Finally, taking into account the increase of resources needed for enhancing the reliability of MapReduce-based systems, we deal with the optimization of the intersection between reliability and resource efficiency of data-intensive computing systems. Particularly, we optimize the resource allocation at the container level of MapReduce systems. A container is an encapsulation of a subset of computing resources, placed on a single node of a cluster. A considerable amount of cloud solutions, not only MapReduce-based clouds, are using currently containers as resource allocation facility. Our approach allocates the amount of resources needed by a specific container, depending on several parameters, such as the real-time request input, the number of requests, the number of users and the dynamic constraints of the system infrastructure, such as the set of resources available.

These three research lines and their corresponding approaches outperform the existing solutions, as we will demonstrate in the next section related to the contributions of the thesis.

Part III

Contributions

Chapter 4

Formalization of the failure detector abstraction in MapReduce

4.1 Introduction

In dependability terms, an omission failure is a more general kind of failure. This happens when a process does not send (or receive) a message that it is supposed to send (or receive).

In asynchronous computing systems, in order to detect and marginalize the impact of slow processes, any framework needs to consider the failure detector abstraction [30]. Failure detectors are abstract devices that offer information about the operational status of processes in a distributed system. It is believed that the failure detector abstraction is fundamental and should sit as a first-class citizen library on any distributed computing framework. Additionally, failure detectors are important because of the possibility to classify problems in distributed computing [51].

Failures are often detected by using either static or dynamic timeout service, which is enforced by using a heartbeat mechanism. The static way is understood as tuning the timeout parameter when starting the job and not changing it until the job execution completes. It is well known that a static timeout value applicable to any application, infrastructure or networking environment does not exist [89]. This is due to its limitations: firstly, that value is not applicable to all the scenarios, and secondly, even if the timeout value would have been chosen well at the beginning, the application, infrastructure and the networking environment may suffer changes (e.g. failures, delays, etc.).

Failure detectors may be divided in perfect or eventual. Perfect detectors may report some process to have crashed, immediately with the first signs of unresponsiveness, while the eventual detectors report a level of suspect. In failure detection, there are two metrics that provide the correctness of the mechanism [33]: (i) *Completeness*, which requires that a heartbeat-based detector eventually suspects every process (task) that actually crashes; and (ii) *Accuracy*, which restricts the mistakes that a heartbeat-based detector can make.

In MapReduce terms, omission failures are synonym for stragglers. Indeed, the

concept of stragglers is very important in the MapReduce community, especially task stragglers, which could jeopardize the job completion time. When MapReduce is deployed on a large-scale infrastructure, consisting of commodity machines, it is better to work on the assumption that stragglers are more than a norm, appearing in huge numbers. Typically, the main cause of a MapReduce straggler task is a slow node, network overload and input data skew [18].

Currently, a static timeout based mechanism is applied for detecting fail-stop failure by checking the expiry time of the last received heartbeat from a certain machine. In Hadoop, each worker sends a heartbeat every 3 seconds, the master checks every 200 seconds the expiry time of the last reported heartbeat. If no heartbeat is received from a machine for 600 seconds, then this machine will be labeled as a failed machine and therefore the master will trigger the failure handling and recovery process. However, some studies have reported that the current static timeout detector is not effective and may cause long and unpredictable latency [44, 45]. Our studies in [77] report that, in the presence of single machine failure the applications' latencies vary not only in accordance to the occupancy time of the failure, similar to [45], but also vary with the job length (short or long).

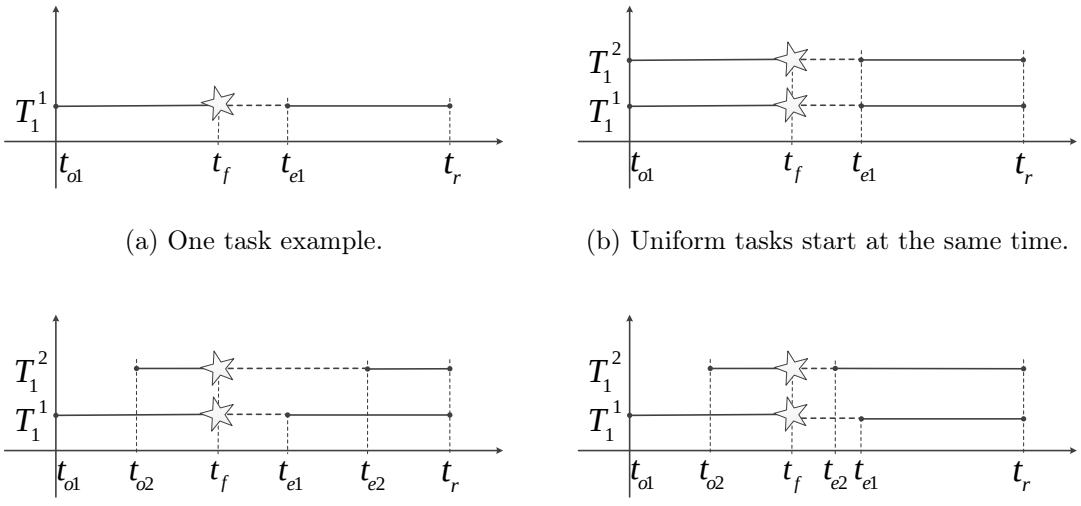
An additional important factor is the non-uniformity of straggler tasks. If we assume that a cloud infrastructure implements a shared Hadoop [11] cluster, it is very possible that some user may be willing to run a CPU heavy job request. If its workload is large, it may need many virtualized tasks, spread among many machines. If other users are running different tasks on these machines, at some point in time, the cluster will have a large number of dynamic stragglers. To worsen the scenario, the percentage of stragglers could be the majority set of job tasks (in meaning, the stragglers number is bigger than or equal to the number of normal tasks). In this case, it is unlikely that a speculative execution could work well, because it would starve cloud resources.

Let us explain the failure timeout problem with an example shown in Figure 4-1.

Case 4-1a. We consider a single task T_1^1 (subscript is the task number, whereas superscript is the job number) of job J_1 to start on a worker m at time t_{o1}. If a worker crashes at time t_f, according to its progress score, the task T_1^1 is assumed to have finished around 70% of the load at time t_f. However, after the failure, a task will need to wait 10 minutes of the arranged timeout to finish in time t_r, and not a time t_{e1}. Clearly, t_{e1} is proportional to the 30% left of the missing load in this case, and obviously less than a 10 minutes timeout.

Case 4-1b. We consider two uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m at the same time t_{o1}. If a worker crashes at time t_f, both T_1^1 and T_1^2 will enforce their respective jobs to wait the 10 minutes timeout, and consequently, to prolong their completion time equally, from t_e1 into t_r.

Case 4-1c. We consider two uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m, but at different times (t_{o1}, and t_{o2},



(a) One task example.

(b) Uniform tasks start at the same time.

(c) Uniform tasks start at different time.

(d) Non-uniform tasks start at different time.

Figure 4-1: Assumed timeout reaction to different task scenarios.

respectively). If a worker crashes at time t_f, both tasks will cause their respective jobs to respect the 10 minutes timeout, and both prolong their completion time equally, from different t_{e1} and t_{e2} into t_r. If we analyze in more detail the start time of these uniform tasks, it is clear that T_1^1 has started earlier than T_1^2, which normally would have finished earlier at time t_{e1}. But the present timeout adjustment harms the T_1^1 equally as T_1^2, by giving advantage to the second task T_1^2, which is not fair.

Case 4-1d. We consider two non-uniform tasks, T_1^1 and T_1^2 of different jobs, J_1 and J_2 respectively, to start on a single worker m, and at different times (t_{o1}, and t_{o2}, respectively). If a worker crashes at time t_f, both tasks will cause their respective jobs to respect the 10 minutes timeout, and both prolong their completion time equally, from t_{e1} and t_{e2} into t_r, as previously. If we analyze in more detail the start time of these non-uniform tasks, it is clear that T_1^1 has started earlier (t_{o1}) than T_1^2 (t_{o2}). However, T_1^2 is shorter, and normally it should have finished earlier at time t_{e2}. But the present timeout adjustment harms the T_1^2 more than T_1^1, by giving advantage to the first task T_1^1, which is not fair again.

The static timeout implemented in MapReduce, is not capable to address any of the cases above. An accurate timeout detector is important not only to improve application's latency but also to improve resource utilization, especially in the Cloud where we pay for the resources we use. Therefore, we state that a significant potential exists for performance improvement in applications, particularly MapReduce applications, when choosing the appropriate timeout failure detector. We believe that a new methodology to adaptively tune the timeout detector can significantly improve the overall performance of the applications, regardless of their execution environment. Every MapReduce job should and can have its proper timeout, because only this is

the proper way of detecting failures in real time.

If stragglers do not affect the time completion in a strong way, a possible proposal in this case would be to dynamically relax the timeout for jobs whose priority is low, and let them run. Another solution to this problem could be diverting priority jobs/users to dedicated machines with no other tasks on them. In this way, if CPU jobs need heavy computations, then you do not run other heavy CPU jobs therein. And the same is possible for memory or I/O jobs. Therefore, if a fault-tolerant abstraction (more precisely, a failure detector abstraction) is deployed for this case, it should consider that different tasks are run in most of the machines, and not all the tasks are the same, in meaning that some tasks are executed in bigger containers, and others need smaller containers. However, dedicated nodes are not an optimal solution, because they will be used for a specific set of job requests and stay idle for other different requests.

As we notice in the Chapter 3, as far as we know, none of the contributions covers a heterogeneous and complete infrastructure, but every work establishes its own assumptions. This is due to the difficulty of modeling a framework for an environment with strictly dynamic requirements. In the same way, we propose three fundamental modules, namely HR-FD (High Relax Failure Detector), MR-FD (Medium Relax Failure Detector) and LR-FD (Low Relax Failure Detector), according to the time heterogeneity, in other words, how much relaxation the system is capable to allow in detecting an omission failure. Before this, we define a system model, upon which these modules are based on.

4.2 System model

Our system model is an abstraction of a single MapReduce job in execution. We consider that each MapReduce request is composed of N_T limited number of identified processes (slave tasks, worker tasks, or tasks) to complete. One of these processes is the master process (leader process, master task, or master) T_M, which controls the other workers tasks T_W. During each MapReduce request, the framework's first duty is to initiate the master process. The master ensures the failure detector execution upon other slave processes. In this case, we consider that the master process is always alive, and correct. In the Chapter 5, we study the case of failure handling of masters. However, this is out of the scope of this contribution. Therefore, during the entire job execution, there is no leader election or any other algorithm that executes in the background to replace the master.

After master initiation from the application manager, the scheduler allocates $N_T - 1$ slave processes. At any time t, a master monitors and coordinates a set of D number of worker tasks ($D \subseteq N_T - 1$), by ensuring and enforcing the correct functioning of each worker task, until they finish the work partition it was assigned to them. During the job execution, if a slave has terminated its task, the scheduler decides whether to assign another task to the slave or terminate it; it is important to mention that, if a slave process is assigned to run another task, it will have another unique id, particular to the work partition upon which was executed on. After all the slaves have finished,

Normal	Suspected	Result
$\in$	$\in$	$suspected \setminus \{task\}$
$\in$	$\notin$	✓
$\notin$	$\in$	✓
$\notin$	$\notin$	suspected $\cup$ {task}

Table 4.1: The probable task intersection between normal and suspected set.

Suspected	Speculated	Result
$\in$	$\in$	✓
$\in$	$\notin$	speculated $\cup$ {task}
$\notin$	$\in$	–
$\notin$	$\notin$	✓

Table 4.2: The probable task intersection between suspected and speculated set.

the master delivers the output to the application manager, which delivers the same output in a readable form to the user, on behalf of the entire framework.

Unless explicitly stated otherwise, it is assumed that a cluster S consists of a limited amount of uniform computing machines n, that could execute a limited number of concurrent processes. We consider that each failure detector algorithm is aware and dependent on the timing assumptions, and not on the resource utilization. However, it is assumed that the more the algorithm relaxes its timeout, the less amount of resources will be requested.

For example, we expect that HR failure detector uses less amount of resources and lets many users concurrently respect or equally have the same infrastructure rights, proportional to their requests. On the other side, the LR-FD algorithm is assumed to request more resources, because the timing assumptions are stricter and therefore, more speculative executions will be needed.

By default, we consider that none of the slave tasks is considered for speculation, without its exclusion from the set of normal tasks. In other words, the failure detector should rearrange the suspected task from the normal set into the suspected set. Only after this, a task may get into the queue of tasks for speculation. All these possible intersections are stated in Table 4.1 and Table 4.2. According to these results, the failure detector mechanisms should react to two scenarios from Table 4.1, because the other two scenarios belong to the normal functioning of the MapReduce framework. If a task is included in both normal and suspected sets, this task should be deleted from the suspected set. On the contrary, if a task is not included in any of these sets, this should be added to the suspected set. Table 4.2 shows that only one scenario requires a solution, namely, the case when a task belongs to the suspected set, but still has not been speculated. In this case, the task should be added to the speculated set. Two other scenarios (1 and 4) are completely correct, whereas the third scenario is not possible, because the failure detector mechanism does not allow a task to directly switch from the normal set to the speculated set, before being included into the set of suspected tasks.

Finally, we assume that our system model does not have network failures. This implies that if we have an operation task, this accomplishes the heartbeat mechanism, and if a task is non operational, the heartbeat will be missed. In other words, network failures are out of the scope of this contribution.

4.3 High relax failure detector

In this module of the framework, called high relax failure detector (HR-FD), we extend the default functioning of the MapReduce failure detector mechanism. Particularly, since the default timeout of Hadoop MapReduce has a static based timeout mechanism of 10 minutes, we leave this value as it is, but only for large jobs, that is, those ones whose completion time is above this value. For other jobs, whose completion time is below the value of 10 minutes, the timeout should be adjusted according to the estimation of the job completion time.

In this way, the failure detector timeout will be fair to most of the user requests, regardless of the other parameters. This statement agrees with the state-of-the-art literature [68, 116, 45, 19, 16, 46], where it is stated that most large-scale MapReduce clusters run small jobs.

As discussed in Table 4.1, any failure detector algorithm should have in mind that, a normal task which is suspected, needs to exit from the normal set, and enter into the suspected set of tasks. According to the possible alternatives, we could derive the Algorithm 1.

Algorithm 1: A task evolution from normal to suspected and viceversa.

```
forall the task ∈ ⊓ do
    if (task ∉ normal) ∧ (task ∉ suspected) then
        suspected := suspected ∪ {task};
        trigger(task, SUSPECT);
    else if (task ∈ normal) ∧ (task ∈ suspected) then
        suspected := suspected \ {task};
        trigger(task, RESTORE);
    end
end
```

For the task which are in the suspected set, it is necessary to find alternatives for completing them. According to Table 4.2, a reasonable decision to make is speculating the suspected task, as a form of not jeopardizing the completion time of the overall job request. Algorithm 2 is used in this scenario. This algorithm is composed of a loop for all the suspected tasks. This algorithm also takes into account the resources available in the system. In particular, if a task is not speculated yet and there are resources available in different nodes to the node in which the task has been scheduled, the algorithm triggers this one as speculated.

A relevant question to solve in this scenario is the maximum number of speculations for a single suspected task. This could be taken into account by the Al-

Algorithm 2: Speculating the suspection.

```
forall the task ∈ suspected do
    if (task ∉ speculated) then
        if (availableResources) ∧ (availableResources ∉ worker_task ) then
            speculated := speculated ∪ {task};
            trigger(task, SPECULATE);
        end
    end
end
```

gorithm 2, substituting the sentence **if**($task \notin speculated$) by **if**($speculated_{task} \geq max\_number\_speculations$). This follows the guidelines provided for instance by [18], which suggests 2 as maximum number of speculations.

In addition, there may be nodes whose performance is causing general overhead on its tasks. In this case, a reasonable reaction would be to consider those nodes as harmful for future executions, and provide a solution to their suspected tasks. The procedure in Algorithm 3 is an example of this scenario. In this case, we are not allocating tasks to a node whose number of suspected tasks is equal or greater to 3.

Algorithm 3: Limiting the suspected task number in the same worker.

```
if (suspected_worker ≥ 3) then
    lost_workers := lost_workers ∪ {suspected_worker};
    trigger(suspected_worker, LOST);
end
```

From these algorithms, we have designed a complete one, called HR-FD, which implements an eventual failure detector algorithm on top of a partially synchronous system. Through this algorithm, we establish time boundaries on omission failures. This is important, since the causes of the stragglers are not only crashes, and the system must react to these issues that harm users and resource providers. In other words, although not sure that crash has happened, the system should decide whether to concurrently speculate the affected task or simply kill the straggler and re-execute it once again from the beginning. The algorithm is described below (Algorithm 4).

In the Algorithm 4, the master process maintains a list of normal, suspected and speculated tasks. In addition, it adjusts a timeout according to an estimated completion time increased with some probability margin of error (λ).

Whenever the failure detector triggers a timeout, it will manage those tasks which do not belong to normal and suspected sets. Accordingly, it will place the tasks in the suspected set, and triggers a suspect event for the respective task. If the task belongs to both sets, then this task will be removed from the suspected set. It is very important to remove it, since it will not request other resources in the next step.

All the suspected tasks are eventually speculated, if new resources, which are independent from the worker, are available. In this condition, we have not implemented

Algorithm 4: The High relax failure detector

```
Implements: HRTimeout
Uses      : ProgressScore
upon event (Init) do
    normal := ⊓;
    suspected := ∅;
    speculated := ∅;
    startTimer(InitialEstimatedTime);
end
upon event (Timeout) do
    forall the task ∈ ⊓ do
        if (task ∉ normal) ∧ (task ∉ suspected) then
            suspected := suspected ∪ {task};
            trigger(task, SUSPECT);
        else if (task ∈ normal) ∧ (task ∈ suspected) then
            suspected := suspected \ {task};
            trigger(task, RESTORE);
        end
    end
    forall the task ∈ suspected do
        if (task ∉ speculated) then
            if (availableResources) ∧ (availableResources ∉ worker_task ) then
                speculated := speculated ∪ {task};
                trigger(task, SPECULATE);
            end
        end
        if (suspected_worker ≥ 3) then
            trigger(suspected_worker, LOST);
        end
    end
    trigger(HeartbeatRequest, task, SEND);
    startTimer(InitialEstimatedTime);
end
upon event (HeartbeatReply, task, DELIVER) do
    normal := normal ∪ {task};
end
```

any speculated number for the task which is suspected, although this is possible.

The algorithm checks another condition in the same loop. Namely, it limits the number of speculated tasks within the same worker. This means, whenever the worker has a certain number of stragglers, it will be considered lost, in order to let the scheduler know that this node should not accept future tasks until the node recovers from this state.

4.3.1 Correctness

The *completeness* property is satisfied by the algorithm, because if a task is behaving as a straggler, it will not send a heartbeat the master for a certain period, which is a condition of the respective master to place this straggler in the suspected set. Regarding the *accuracy* property, a timeout according to the initial estimation is believed to be sufficient for every task to deliver a heartbeat, informing the master about its liveness and progress score.

4.3.2 Performance

In order to detect a MapReduce straggler through the HR module, the initial timeout adjustment is crucial. As we have stated before, this algorithm should be capable to adjust a job specific timeout, according to the estimated job completion time, and a probable margin of error. This is particularly important for small jobs, whose estimated completion time is under 10 minutes. For longer lasting jobs, we have decided to leave the default timeout of the Hadoop MapReduce. By this statement, we have placed a higher boundary (that is, 10 minutes) of the timeout. Knowing that the majority of the production clusters run small jobs [68, 45, 19, 16, 46], this timeout is actually addressing the vast majority. We consider that there are also very tiny jobs whose estimated completion time is very small, maybe in matter of seconds. Since a timeout of these margins would result harmful for the infrastructure (resource utilization), by giving many wrong suspicions, the algorithm should deploy a minimal timeout in these cases. This minimal timeout should have a lower boundary, that guarantees no eventual processing overhead. For small infrastructures, the administrator can decide this value. For larger infrastructures, the best choice would be to apply an autonomic approach, similar to our contribution in Chapter 6.

We provide performance simulations, by comparing the Hadoop timeout with the HR-FD timeout. In Table 4.3, we have taken as a sample a workload with an estimated time completion of 5 minutes. This is an average value, which help us to see the evolution, since every iteration lasts approximately 1 minute [115]. In the first column, there are different iterations of the same workload. According to the iteration, the second column indicates the finish time of the workload with no failures. The third column represents the failure injection time, respective to each iteration. Right after the Hadoop finish time, there are 4 columns listing the HR-FD finish time, with different λ, which represents the margin of error. As we can notice, this static timeout, which is clearly better than the default timeout in Hadoop MapReduce, performs really well for most of the average production cluster jobs, whose completion time is

Iteration	Failure free completion time	Failure injection time	Hadoop completion time	$\lambda = 1.00$	$\lambda = 0.75$	$\lambda = 0.50$	$\lambda = 0.25$
1	5	1	10	6.00	5.75	5.50	5.25
2	4	2	11	7.00	6.75	6.50	6.25
3	3	3	12	8.00	7.75	7.50	7.25
4	2	4	13	9.00	8.75	8.50	8.25
5	1	5	14	10.00	9.75	9.50	9.25

Table 4.3: A performance comparison of Hadoop MapReduce timeout and HR-FD timeout for a 5 minutes workload.

neither very long and neither very small. As is shown in the Table, the smaller margin error is used, the estimations are more accurate, since the system is more stable.

The best performance of HR-FD corresponds to the first iterations. This is due to the fact that the timeout triggers a threshold very early, and is capable of maintaining a moderate completion time upon failures, comparable to the normal case. For example, in case of $\lambda = 0.75$, if a failure is enforced during the first iteration, from a normal estimated completion time of $t_{e1} = 5min$, the new completion time would be $t_r = 5.75min$, instead of the Hadoop completion time of $t_H = 10min$. In other words, HR-FD timeout exhibits only 15% of performance degradation, whereas Hadoop timeout exhibits 100% of performance degradation. Figure 4-2 shows the behavior of HR-FD compared to default Hadoop and a failure-free scenario along the iterations. As the job progress score advances through iterations, the HR-FD timeout benefits decrease when compared to the Hadoop timeout, but despite this, they are still clearly much more favorable.

4.4 Medium relax failure detector

The previous algorithm (Algorithm 4) belongs to the failure detector mechanisms deploying a static timeout service. That is, the timeout provided by HR-FD is static, although adjusted at the beginning. Even the algorithm outperforms the default Hadoop mechanism, our aim is to extend the basic HR-FD algorithm by giving to the timeout service a dynamic value.

The ideal choice would be to use an estimated progress score of the overall job, which is then divided in its phases and consequently, in individual tasks. A dynamic timeout can rely on the progress score, especially when is predictably task-dependent. This is an already built-in feature in MapReduce-based systems [11, 4, 62], and other contributions have given even more accurate results in this field [81, 82].

The main novelty in this module is outlined in the lines of Algorithm 5, where the algorithm calculates the progress score, by adding a margin of error value (λ), that should carry the timeout of the next iteration. This involves that the new progress

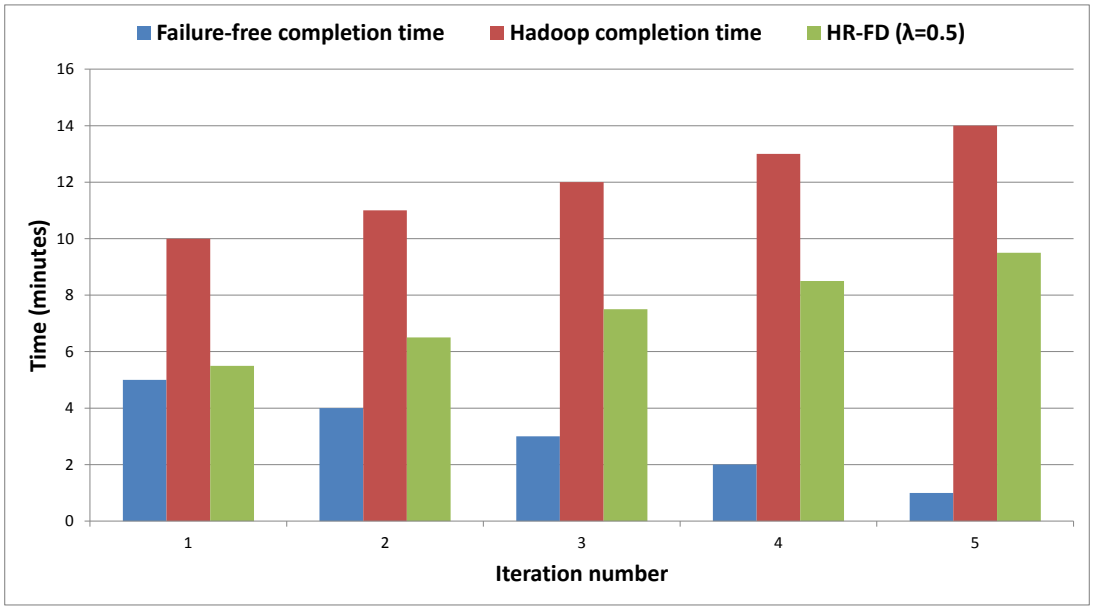


Figure 4-2: A performance comparison of Hadoop MapReduce timeout and HR-FD timeout for a 5 minutes workload.

score is the main parameter of the newly chosen and calculated timeout. Upon each execution of the timeout event, the module finally starts a new timeout with a new value.

Algorithm 5: Procedure to calculate the estimated progress score.

```
...
ProgressScore := ProgressScore + λ;
...
startTimer(ProgressScore);
...
```

The Algorithm 6 presented below, is called Medium relax failure detector (MR-FD) and implements an eventual failure detector algorithm on top of a partially synchronous system. MR-FD enforces stronger timing assumptions than in HR-FD, but it still implements an eventual failure detector, presumably in a partially synchronous system. Algorithm 6 has resemblance to the previous algorithm, except in some additional lines. Basically, every task that gets out from the normal set, joins the suspected set of tasks, and in a certain moment, when the cluster provides additional resources, it is executed.

Algorithm 6: The Medium relax failure detector

```
Implements: MRTimeout
Uses        : ProgressScore
upon event (Init) do
    normal := ⊓;
    suspected := ∅;
    speculated := ∅;
    startTimer(ProgressScore);
end
upon event (Timeout) do
    ProgressScore := ProgressScore + λ;
    forall the task ∈ ⊓ do
        if (task ∉ normal) ∧ (task ∉ suspected) then
            suspected := suspected ∪ {task};
            trigger(task, SUSPECT);
        else if (task ∈ normal) ∧ (task ∈ suspected) then
            suspected := suspected \ {task};
            trigger(task, RESTORE);
        end
    end
    forall the task ∈ suspected do
        if (task ∉ speculated) then
            if (availableResources) ∧ (availableResources ∉ worker_task ) then
                trigger(task, SPECULATE);
                speculated := speculated ∪ {task};
            end
        end
        if (suspected_worker ≥ 3) then
            trigger(suspected_worker, LOST);
        end
    end
    trigger(HeartbeatRequest, task, SEND);
    startTimer(ProgressScore);
end
upon event (HeartbeatReply, task, DELIVER) do
    normal := normal ∪ {task};
end
```

Iteration	Failure free completion time	Failure injection time	Hadoop completion time	$\lambda = 1.00$	$\lambda = 0.75$	$\lambda = 0.50$	$\lambda = 0.25$
1	5	1	10	6	5.75	5.5	5.25
2	4	2	11	5	4.75	4.5	4.25
3	3	3	12	4	3.75	3.5	3.25
4	2	4	13	3	2.75	2.5	2.25
5	1	5	14	2	1.75	1.5	1.25

Table 4.4: A performance comparison of Hadoop MapReduce timeout and MR-FD timeout for a 5 minutes workload.

4.4.1 Correctness

The failure detector properties are stronger than in the previous algorithm. This means that, both completeness and accuracy have their assumptions equal or stronger than HR-FD. Considering this, the completeness property triggers a threshold as long as the minimum progress score is left, in order to change the set of a suspected task. On the other hand, the accuracy property is stricter to guarantee the liveness property of the monitored task.

4.4.2 Performance

Unlike the Algorithm 4, whose initial adjustment of the timeout is crucial, the Algorithm 6 does not really depend on the initial adjustment, except in the case of those workloads whose estimated completion time is really small, since in this case, the completion time is equal to the minimum possible timeout adjustment. This minimum value is the same as in the Algorithm 4, in order to enforce boundaries for future latency and heartbeat overhead.

The dynamic timeout service is provided at the expense of a higher resource utilization. Whereas the computing resources did not represent any real starvation risk, the Algorithm 6 is using clearly a higher amount of resources, since more speculative executions are needed.

In Table 4.4, we provide performance simulations, maintaining the same methodology than HR-FD (a workload sample with an estimated completion time of 5 minutes), by comparing the Hadoop timeout with the MR-FD timeout. Unlike the HR-FD timeout, whose reaction outcome was clearly noted in the first iterations due to its static parameter, MR-FD behaves clearly much better than the default timeout setup of Hadoop MapReduce.

For instance, for $\lambda = 0.75$, if a failure is injected during the first iteration, from a normal estimated completion time of $t_{e1} = 5min$, the new completion time would be $t_r = 5.75min$, instead of the Hadoop completion time of $t_H = 10min$. In other words, MR-FD timeout exhibits only 15% of performance degradation, whereas Hadoop

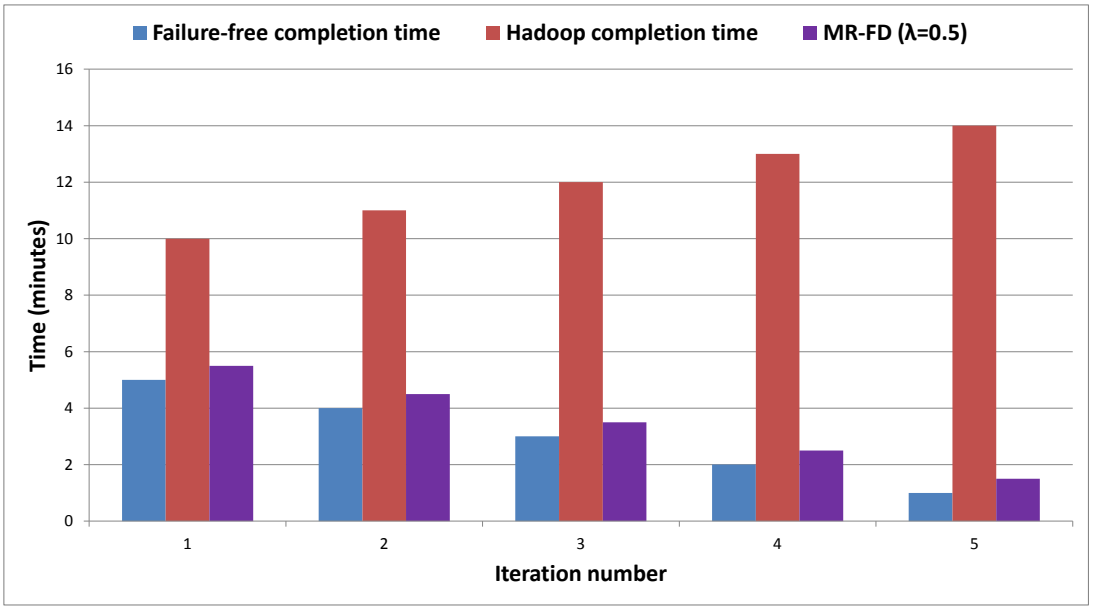


Figure 4-3: A performance comparison of Hadoop MapReduce timeout and MR-FD timeout for a 5 minutes workload.

timeout exhibits 100% of performance degradation. And actually, as the job progress score advances through iterations, the MR-FD timeout maintains its performance when compared to the Hadoop timeout, except by the small influence of the accuracy margin values, which make the difference for all the iterations.

As in the previous section, Figure 4-3 shows the behavior of MR-FD compared to default Hadoop and a failure-free scenario along the iterations. Unlike HR-FD, the MR-FD timeout benefits increase when compared to the Hadoop timeout as the iterations increase.

4.5 Low relax failure detector

As previously mentioned, the difference between Algorithm 4 and Algorithm 6 is the time reacting to failures. Whereas Algorithm 4 assume static timeout predictions for suspicious tasks, the Algorithm 6 reacts dynamically to the same suspicions, by providing clear advantage in job completion time. However, the Algorithm 6 does not completely provide strictly bounded timings assumptions. This algorithm does not fit with systems whose results are strictly deadline-bounded, and where it would be possible to afford a bigger amount of resources in order to complete their tasks as fast as possible. These systems could belong to the type of mission critical systems (such as military or air traffic control systems), or enterprise systems (such as auctioning systems), whose decision making is important and urgent.

Apart from the heartbeat mechanism that is used to deploy a timeout service for tasks, the same mechanism could be used to monitor the machines (node, worker)

metrics and adjust specific thresholds in order to target or enforce the completion time of deadline-bounded workloads (requests).

For deadline-bounded workloads we consider the timeout for the workers as an additional parameter. As long as this timeout notices uncommon behavior after a certain established period, it will request from the application to trigger speculative execution for the ongoing tasks on that particular worker. If there is no task, then it will stop deploying future tasks, until the worker establishes itself into the normal set. For example, in a worker, whose monitoring system monitors parameter p, if this parameter does not appear during a number of times provided by a *threshold* function, the failure detector mechanism will suspect all of its tasks, and declare the worker as lost, as shown in Algorithm 7.

Algorithm 7: Worker parameters monitored with separate timeout.

```
forall the worker ∈ ⊓ do
    if (measures(worker, p) ≤ threshold(p)) then
        suspected := suspected ∪ {task_worker};
        trigger(task, SUSPECT);
        lost_workers := lost_workers ∪ {suspected_worker};
        trigger(suspected_worker, LOST);
    end
end
```

The Algorithm 8 may even expand and maintain a history of the workers. By detecting a repeatable defect in any of them, it may decide to give priority to newer or more stable workers, as long as they respect a certain degree of data locality [110].

The approach is shown in Algorithm 8. This algorithm expands the Algorithm MR-FD, by considering the additional timeout of the workers. As soon as one of the timeouts shows an uncommon behavior, from either tasks or workers, the algorithm triggers speculations in different stable nodes.

4.5.1 Correctness

Let us consider the completeness property first. As long as a task or worker is behaving right, the algorithm does not act. However, if a task or worker do not deliver heartbeat signals for a certain period, these tasks or workers will be deleted from the normal set of tasks/workers, entering into the suspected set. The master process will suspect these tasks and workers until the job has finished. Therefore, this algorithm shows some differences in terms of the use of the sets with regards to the previous two failure detectors. Indeed, the change of the sets is for this algorithm more rigorous. If there is a timeout threshold, the suspected tasks will not be terminated. However, after a certain task enters into the suspected set, it will not be able to come back to the normal set again. Therefore, all these tasks of the suspected set have to be speculated in other workers.

Algorithm 8: The Low relax failure detector

```
Implements: LRTimeout
Uses      : ProgressScore
upon event (Init) do
    normal := ⊓;
    suspected := ∅;
    speculated := ∅;
    startTimer(ProgressScore, threshold(p));
end
upon event (Timeout) do
    ProgressScore := ProgressScore + λ;
    forall the task ∈ ⊓ do
        if (task ∉ normal) ∧ (task ∉ suspected) then
            suspected := suspected ∪ {task};
            trigger(task, SUSPECT);
        end
    end
    forall the task ∈ suspected do
        if (task ∉ speculated) then
            if (availableResources) ∧ (availableResources ∉ worker_task ) then
                trigger(task, SPECULATE);
                speculated := speculated ∪ {task};
            end
        end
        if (suspected_worker ≥ 3) then
            trigger(suspected_worker, LOST);
        end
    end
    trigger(HeartbeatRequest, task, SEND);
    startTimer(ProgressScore);
end
upon event (Timeout_worker) do
    forall the worker ∈ ⊓ do
        if (measures(worker, p) ≤ threshold(p)) then
            suspected := suspected ∪ {task_worker};
            trigger(task, SUSPECT);
            lost_workers := lost_workers ∪ {suspected_worker};
            trigger(suspected_worker, LOST);
        end
    end
    startTimer(threshold(p));
end
upon event (HeartbeatReply, task, DELIVER) do
    normal := normal ∪ {task};
end
```

Iteration	Failure free completion time	Failure injection time	Hadoop completion time	$\lambda = 0.25$	$\lambda = 0.15$	$\lambda = 0.10$	$\lambda = 0.05$
1	5	1	10	5.25	5.15	5.10	5.05
2	4	2	11	4.25	4.15	4.10	4.05
3	3	3	12	3.25	3.15	3.10	3.05
4	2	4	13	2.25	2.15	2.10	2.05
5	1	5	14	1.25	1.15	1.10	1.05

Table 4.5: A performance comparison of Hadoop MapReduce timeout and LR-FD timeout for a 5 minutes workload.

Regarding the accuracy property, the master will suspect a task (worker), only if a task (worker) is not able to transmit a message within a specified interval. Otherwise, if any task or worker behave properly, it is assumed that would be able to send delivery notifications to the master.

4.5.2 Performance

The above Algorithm 8 does not work with a single static timeout, but neither it depends only on a single dynamic timeout. Therefore, it requests the participation of an external monitoring system. While having two timeout observers, every time when a change happens that is detected from one or both existing entities, the user can detect probable relationships between correlations, and react in advance based on our predefined conditions, although independent to the resource cost.

On the contrary to the two previous Algorithm 4 and Algorithm 6, whose initial and the dynamic adjustment of the timeout was crucial, in this case, the Algorithm 8 goes beyond this. As previously mentioned, again, the first adjustment is only important for those workloads whose estimated completion time is really small, and the dynamic adjustment is only important for those workloads whose estimated completion time is endangered from common task problems. However, the suspicion probability takes bigger risks when noticing machine issues in those places where tasks are allocated. On the other hand, if the failure detector monitor depends on two stricter timeouts instead of one, the resource utilization factor increases, because the algorithm reacts sooner to suspicions and therefore, there would be more suspicions than in the other scenarios.

In Table 4.5, we provide performance simulations, maintaining the same methodology as in HR-FD and MR-FD (a workload sample with an estimated time completion of 5 minutes), by comparing the Hadoop timeout with the LR-FD timeout. Unlike the HR-FD timeout, the Algorithm 8 reacts very well for any production cluster scenario, and performs a slight improvement when comparing to the MR-FD timeout.

For example, in case of $\lambda = 0.15$, if a failure is enforced during the first iteration, from a normal estimated completion time of $t_{e1} = 5min$, the new completion time

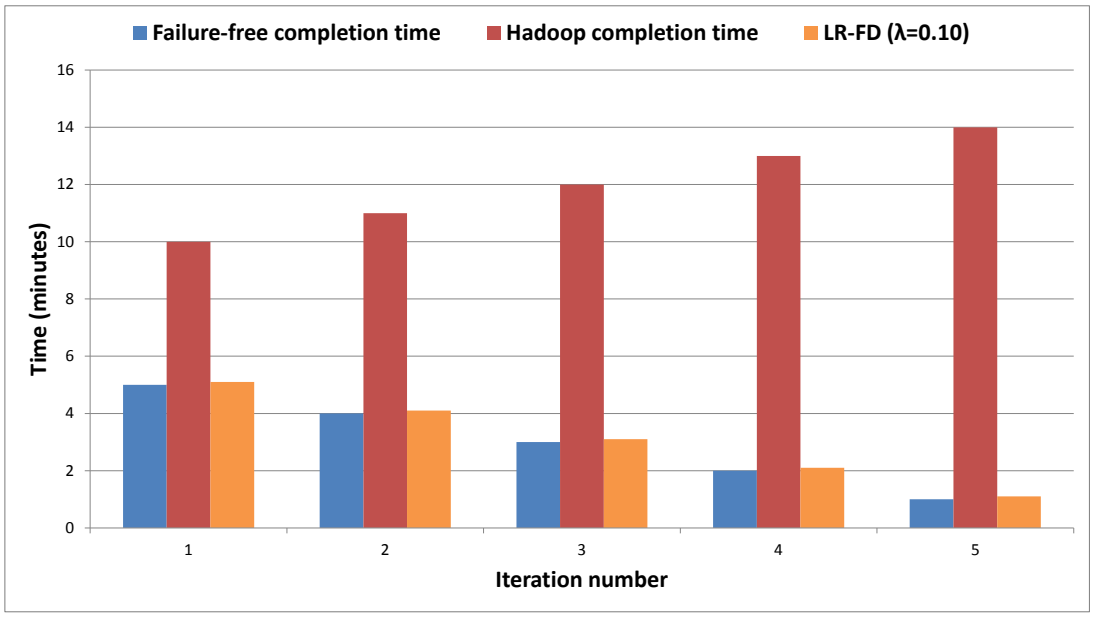


Figure 4-4: A performance comparison of Hadoop MapReduce timeout and LR-FD timeout for a 5 minutes workload.

would be $t_r = 5.15min$, instead of the Hadoop completion time of $t_H = 10min$. In other words, LR-FD timeout exhibits only 3% of performance degradation, whereas Hadoop timeout exhibits 100% of performance degradation.

As in the previous cases, Figure 4-4 shows the comparison between default Hadoop timeout and LR-FD timeout, representing also the failure-free completion time. This is clearly the approach with a higher performance, due to both timeouts (tasks and workers) and the lower margin of error exhibited by this algorithm.

Finally, Figure 4-5 shows the comparison between all the algorithms, HR-FD, MR-FD and LR-FD and the default Hadoop MapReduce timeout. As we can notice, MR-FD and LR-FD behaves much better than the other alternatives. This is due to the dynamic adjustment of the timeout, which is applied by both approaches. LR-FD behavior is slightly better than MR-FD, demonstrating that the external timeout, that is, the timeout associated to the workers, is not so relevant as the task timeout.

4.6 Summary

This chapter provides a formalization of a failure detection abstraction for MapReduce-based systems. As part of this formalization, we have introduced three different algorithms, namely: (i) High relax failure detector (HR-FD), based on a static timeout service; (ii) Medium relax failure detector (MR-FD), based on a dynamic timeout service; and (iii) Low relax failure detector (LR-FD), based on the intersection between the MR-FD timeout service and an external monitoring system timeout service, in order to achieve more efficient failure detections.

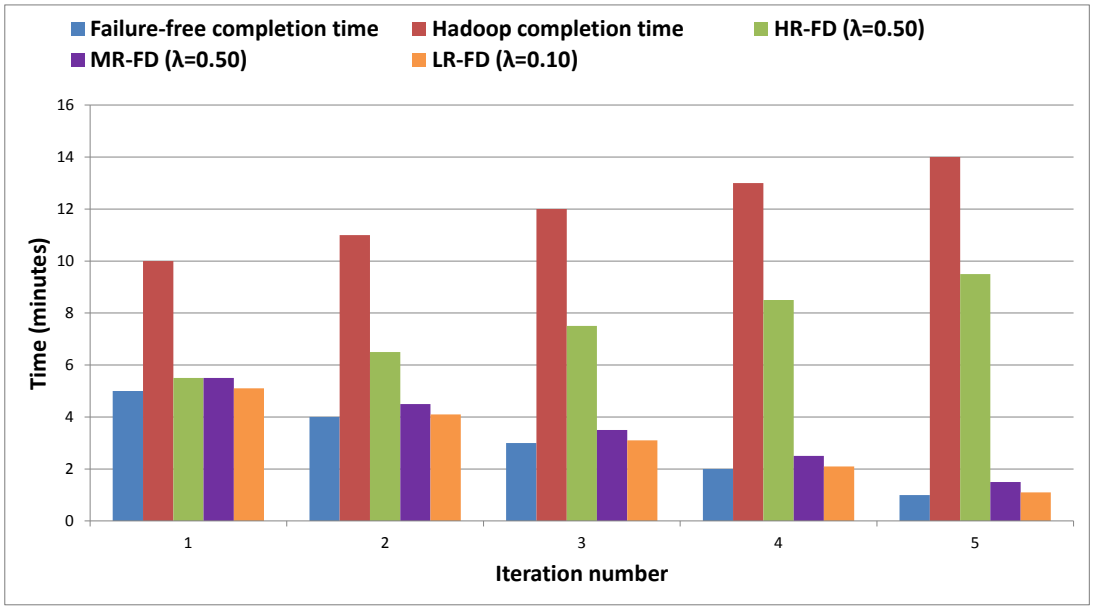


Figure 4-5: A performance comparison of Hadoop MapReduce timeout, HR-FD, MR-FD, and LR-FD timeout for a 5 minutes workload.

According to the performance measurements, we have demonstrated that these abstractions outperform the default timeout service of the Hadoop YARN.

Chapter 5

Diarchy: peer management for solving the MapReduce single points of failure

5.1 Introduction

Hadoop has been widely used in the last years as the open source implementation of the MapReduce framework proposed by Google [42]. Over these years, the classic version of Hadoop has faced many drawbacks in large-scale systems. Among them, scalability, reliability and availability are major issues not solved in the classic Hadoop. The YARN project has been developed with the aim of solving these problems and other additional ones, described in [101].

The key idea behind YARN is the division of the duties of the JobTracker of the classic Hadoop version into separate entities: (i) the Resource Manager, composed by the Scheduler and Application Manager; (ii) the Node Manager, which is responsible for monitoring nodes and informing to the Resource Manager, and (iii) the Application Master, whose duty is to negotiate the number of workers and monitoring their progress. Whereas the Scheduler is in charge of resource allocation, the Application Manager accepts job submissions, and initiates the first job container for the job master. This job master is the leader and is called Application Master.

This architectural change has as main goal to provide scalability. In addition, it removes the single point of failure presented by the JobTracker. However, this design decision has introduced new sources of problems. Namely, the resource scheduler, the application manager and the application master are in the YARN architecture single points of failure.

This chapter describes a new proposal, called Diarchy, which aims to increase the reliability of YARN, without decreasing its performance and scalability. Diarchy is based on the sharing and backup of responsibilities between two master peers. For the sake of simplicity, the chapter only addresses the solution for the application master. However, this methodology can be directly applied to the other two cases: resource scheduler and the application manager.

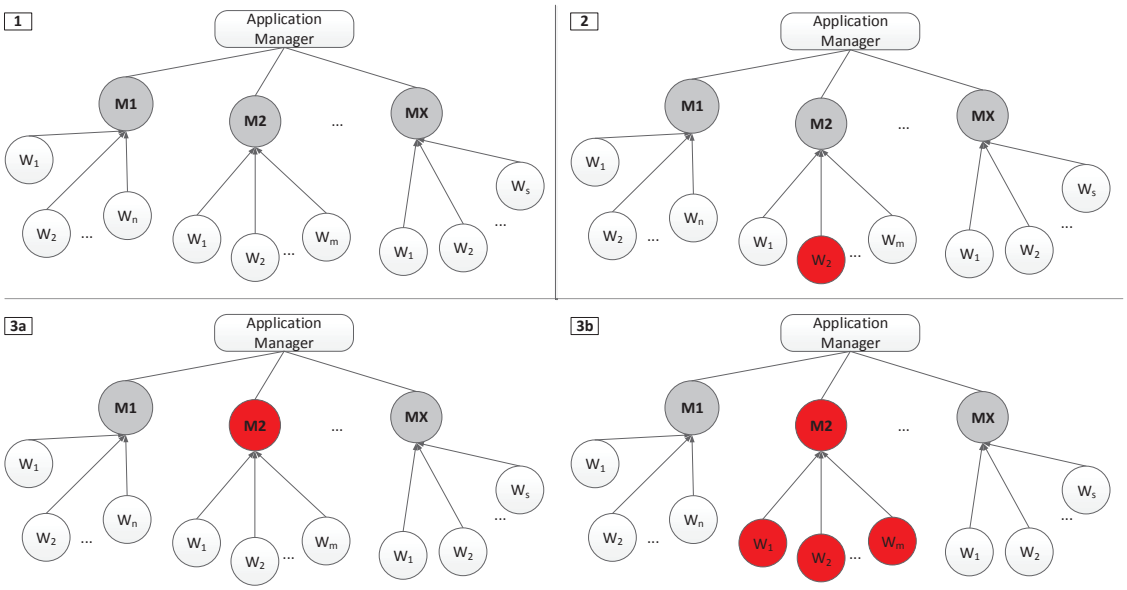


Figure 5-1: YARN architecture in different scenarios. Case **1.** Normal working state. Case **2.** Failures among workers. Case **3a** and **3b.** Failure among masters. YARN does not have a solution for Case 3a. The consequence of this is Case 3b (3a $\preccurlyeq$ 3b).

As a summary, this chapter has the following main contributions:

1. Definition of the reliability problem of YARN from a probabilistic point of view.
2. Design of a new model, called Diarchy, based on responsibilities delegation between peers.
3. Evaluation of Diarchy and comparison with the behaviour of YARN.

The remainder of this work is organized as follows. We describe in-depth the problem in Section 5.2. We introduce the Diarchy architecture in Section 5.3. We evaluate the Diarchy approach in Section 5.4. Finally, we summarize the contribution in Section 5.5.

5.2 Problem definition

In this section, we describe the motivation behind this work. We analyze, from a probabilistic point of view, the impact of failure of MapReduce masters in a large-scale environment scenario. A sketch of this analysis is illustrated in Figure 5-1.

According to [68], there are 5 failures on average for each MapReduce job in Industry clusters. Many approaches have been dealt with the failures of generic tasks, but as far as we know, none of them is focused on the failures of masters in the YARN architecture.

As we can see in Figure 5-1, the Case 1 shows a normal working state in YARN. In this case, we have M_X MapReduce masters and $n + m + \cdots + s$ workers, and all of them work perfectly. Case 2 shows a scenario in which one worker fails. This does not affect to other workers. Our work focuses on the Case 3, in which the failure in a master implies a failure in the workers managed by such a master.

Let us consider a cluster S that runs 1 million jobs ($J = 1000000$) per day. Let us suppose each job needs around 350 slave tasks ($N_T = 350$) to complete. One of these tasks is the master task T_M, which controls the other workers tasks T_W. As previously mentioned, on average, a cluster has a failure rate of 5 tasks per job ($N_F = 5$). Since the master is executed as any other tasks, its failure probability, $P(F_M)$, has the same value as the failure probability of other tasks in the cluster, that is, the workers failure probability, $P(F_W)$. Both are equivalent to the task failure probability, $P(F_T)$:

$$P(F_M) = P(F_W) = P(F_T) \tag{5.1}$$

According to Laplace rule, the failure probability for each task is:

$$P(F_T) = \frac{1}{N_T} \times N_F \tag{5.2}$$

Therefore, in our scenario, we get:

$$P(F_T) = \frac{1}{350} \times 5 = \frac{1}{70}$$

By transforming the binary distribution β into a normal distribution $\mathcal{N}$ where $Q = 1 - P(F_T)$, we can calculate the total number of failed tasks, F_T :

$$F_T = \beta\left(N_T, P(F_T)\right) \approx \mathcal{N}_{\mathcal{T}}\left(N_T P(F_T), \sqrt{N_T P(F_T) Q}\right) \tag{5.3}$$

In our case, according to the above formula we have:

$$\begin{aligned} F_T &= \beta\left(350.000.000, \frac{1}{70}\right) \\ &\approx \mathcal{N}\left(\frac{350.000.000}{70}, \sqrt{\frac{1}{70} \times \frac{69}{70} \times 350.000.000}\right) \\ &\approx \mathcal{N}\left(5.000.000, 2220\right) \end{aligned}$$

Consequently, the number of failed masters, F_M is:

$$\begin{aligned} F_M &= \beta\left(1.000.000, \frac{1}{70}\right) \\ &\approx \mathcal{N}\left(1.000.000 \times \frac{1}{70}, \sqrt{\frac{1}{70} \times \frac{69}{70} \times 1.000.000}\right) \\ &\approx \mathcal{N}\left(14286, 119\right) \end{aligned}$$

This is a very large number[1], taking into account that a potential failure of these masters will cause re-execution of the same number of jobs. Each of these failed masters, managing a considerable amount of tasks, will consequently involve the loss of the progress of these tasks:

$$F_{WM} = F_M \times N_T = 14286 \times 350 = 5.000.100$$

being F_{WM} the number of failed workers by the failure of a master.

This means that in the worst case assumption the number of failed tasks would be this number times 2, because of the addition of the probability of failure of worker tasks. This number would go up to 10 million per day. In other words, 2.86% of the tasks needs to be re-executed for a million job cluster.

The resource utilization numbers we get are even higher. If each re-executed job uses 10 minutes on average to complete, this will represent a huge delay, making it impossible a close to optimal resource utilization as expected in the official proposal of YARN [101].

5.3 Diarchy algorithm

Diarchy is an old form of government which is still used in some countries. This government system has two individuals, called *diarchs*, in charge of governing the country or the State. Diarchs share the responsibilities and report the progress state of their work to each other. In this way, the diarch 1 is informed of the duties of diarch 2 and viceversa. In a wider context, diarchy is nowadays used for defining a dual rule system for both organizations or governments.

Our approach, called Diarchy algorithm, is based on the Diarchy ruling as follows: in the applications master queue, there are two jobs, J_1 and J_2, waiting to be run. The application manager will assign one master to each job, $T_{M1} \in J_1$ and $T_{M2} \in J_2$, while informing both of them of their peers. If something happens to T_{M1}, the other peer T_{M2} will be in charge by the virtue of delegation. In the same way, if something happens to T_{M2}, the respective peer T_{M1} will take the responsibility for it. Along the time, both masters will behave as peers, and apart from the progress score of their workers, they will also receive the progress score of their peer workers. In order to achieve this, the application manager waits for two consecutive jobs to arrive in the queue and then submits them both for execution. This is feasible, since the load of a master node is significantly lower than the worker nodes one. Therefore, an additional progress score from its peer workers would not represent potential danger or misbehavior for any of the masters. Figure 5-2 shows graphically the Diarchy algorithm.

[1]The first value is the master failure probability rate, and the second value its standard deviation.

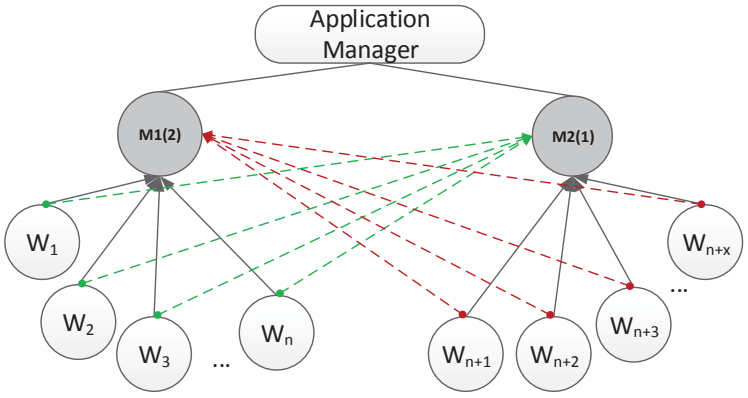


Figure 5-2: Diarchy architecture.

5.3.1 Diarchy performance

One of the major issues of failure detection mechanisms in MapReduce is the proper usage and configuration of an important parameter, the heartbeat. By default, missing a certain number of heartbeats is used for detecting the entities failures in MapReduce. Besides from serving as acknowledge message to the master, indicating that a worker is alive, heartbeats are also a channel for other kind of messages. As a part of the heartbeat, a worker will indicate whether it is ready to run a new task, and if it is, the master will allocate a task, which is used to communicate to the worker using the heartbeat return value [105]. In the case of the Diarchy algorithm, as each worker reports in parallel to both masters, this will cause higher bandwidth utilization. In order to avoid this, the heartbeat will only report to a master, alternating the master in every iteration. Namely, if in the current YARN version, this report value is sent every t seconds, the Diarchy algorithm will require each master to be informed every $2 \times t$ seconds, with the aim of optimizing the network utilization.

Now, the probability value of the failure rate by using the Diarchy algorithm is:

$$P(T_M) = \frac{N_F}{N_T} \times \frac{N_F}{N_T} \tag{5.4}$$

since both masters have to fail in the case of a master failure.

Therefore, in our study case we have:

$$P(T_M) = \frac{1}{70} \times \frac{1}{70} = \frac{1}{4900}$$

And thus, the number of failed masters is:

$$
\begin{aligned}
X &= \beta\left(N_M, P\left(T_M\right)\right) \\
&= \beta\left(1.000.000, \frac{1}{4900}\right) \\
&\approx \mathcal{N}\left(204, 14\right)
\end{aligned}
$$

This formula provides the average failure rate for Diarchy, taking into account that when both of the master peers fail, they are considered as one failure, as previously mentioned. Consequently, the real number of failed masters would be:

$$
F_M = \mathcal{N}\left(N_M, P(F_M)\right) \times 2 \tag{5.5}
$$

As we can notice, the Diarchy algorithm is $\frac{N_T}{N_F}$ more reliable than the default Hadoop YARN. This means that Diarchy puts a lower boundary in the worst case assumption, with the number of failed tasks not surpassing 5 millions per day. In other words, the new percentage margin is 1.43% of possible re-executions, instead of 2.86% as it was with the default YARN framework.

5.3.2 Going general: a possible m-peers approach

Theoretically, the Diarchy algorithm could work with more peers (let us say m peers) than those assigned in the default proposal, that is, 2. In this way, the fault tolerance provided by this hypothetical framework would be higher. However, in practice, this is not advisable, because of three reasons: (i) *network overhead*; (ii) *failure detection efficiency* and (iii) *abstraction complexity*. Below we explain in depth all these issues.

By using more than 2 peers in a Diarchy scheme, all the workers have to report their progress to the additional masters. This would increase the number of messages exchanged in the framework, since there has to be a message interaction between m masters and n workers. The default heartbeat value in YARN Hadoop ranges from 1 to 3 seconds, depending on different attributes such as cluster size, type of jobs, etc. If this interval stays unchanged, the **network load** is increased proportional to the number of masters. This is unfeasible, thus it would be necessary to increase the default heartbeat value.

By default, the MapReduce failure detection service only addresses crash and crash-recovery failures. If the heartbeat message of a task has not been sent after a given threshold (600 seconds, by default), the master will mark the task as failed and will perform a set of steps to solve the issue. Namely, it requests an additional container for the task and reschedules its execution, trying to avoid rescheduling the task on the same node where it has previously failed. An earlier speculative execution of failed tasks is highly prioritized, in order to detect if a tasks fails repeatedly due to a bug. If this is the case, then a master could invoke stopping the job earlier.

In failure detection, there are two metrics that provide the goodness of the mechanism [33]:

- *Completeness*, which requires that a heartbeat-based detector eventually suspects every task that actually crashes.

- *Accuracy*, which restricts the mistakes that a heartbeat-based detector can make.

By using an m-peer scheme, the failure detection mechanism can decrease the completeness property and therefore, to decrease the **failure detection efficiency**. This is due to the fact that as the heartbeat would be higher, some failures could not be detected.

Finally, an important feature of Diarchy is its simplicity. Keeping two peers maintains the abstraction simplicity. However, if the number of master peers is increased, the **abstraction complexity** will be increased, since it would be necessary to design a more complex protocol to be used in case of failure of one of the masters.

5.4 Experimental evaluation

The aim of this section is the evaluation of the Diarchy algorithm. The evaluation is based on the following parameters:

- Job size: in the set of experiments, we have analyzed the Diarchy algorithm for different job sizes, starting from a job that consists of 100 workers tasks, until the job reaches the size of 1000 tasks.

- Cluster size: we have tested Diarchy in different set of clusters, ranging from a small cluster with a capacity to serve 10 daily normal jobs, up to a large-scale cluster with potentially million job executions running per day.

- Failure rate: last but not least, the failure rate in the cluster is an important metric for the performance of Diarchy, since there are different infrastructures with a variable number of failure rates, ranging from 1 failure to 10 failures[2].

Subsequently, we have evaluated how these parameters are related by means of the following scenarios:

- Job size vs. failure rate: this test measures how Diarchy behaves regarding these two parameters, considering the same cluster.

- Job size vs. cluster size: it measures the behavior of Diarchy with regards to the job and cluster sizes, keeping the same failure rate.

- Cluster size vs. failure rate: it measures the behavior of Diarchy for the same type of jobs, according to different failure rates and cluster size.

[2]As previously mentioned, there are 5 failures on average for each MapReduce job in Industry clusters.

5.4.1 Experimental settings

The Diarchy algorithm has been evaluated on a round-based simulator. This simulator enables the evaluation of all the important parameters of Hadoop, including the default functioning of Hadoop YARN. On top of YARN, we have implemented the Diarchy algorithm.

Along the experiments, if not stated otherwise, the default configuration of the simulating environment is $J = 10000$ jobs in cluster S, each of the jobs composed by $N_T = 350$ tasks (one for master, the rest for worker nodes), and with $N_F = 5$ task failure rate. For the sake of simplicity and efficiency, a single value was analyzed, the master failure rate.

5.4.2 Single parameter tests

Job size. In this first set of experiments, we compare Diarchy and the default YARN Hadoop in a cluster with different job workloads. In practice, users submit jobs which consist on different workloads, depending on the user application. Consequently, the MapReduce framework allocates an estimated number of tasks for every workload.

In the three following figures (Figure 5-3, 5-4, and 5-5 we have run experiments with job workloads whose number of tasks ranges from 100 up to 1000. In Figure 5-3 we assume that when a job increases its number of tasks, the master failure probability gets lower. The reason for this is that independently from the number of tasks, we keep constant the number of failures. At first, we notice that YARN has a poor behavior when the number of tasks is 100. While the master failure reaches the value of 500 in YARN, the Diarchy execution value remains by far lower, not exceeding 50. As expected, the failure rate in bigger workloads demonstrates relatively smaller values for both executions. However, Diarchy again outperforms YARN, by showing low boundary values that are near to zero.

Figure 5-4 shows the results of the experiments introducing a different setup, that is, the failure rate is proportional to the number of tasks by a factor $5 : 350$. On average, the workloads used in Hadoop jobs are composed by 350 tasks [68, 45]. This means that the smallest workload in the experiment, which consists of only 100 tasks, has an approximate failure rate of 1.43, the workload with 200 tasks would have a failure rate of 2.86 and so on, until the largest workload, which consists of 1000 tasks, having an average failure rate of 14.29. We observe that while YARN failure rate ranges from 110 to 160 master failures rates for 10000 jobs, the highest value in Diarchy reaches 4.8, which is considerably lower compared to YARN.

In Figure 5-5 we can see the results of the experiments by using a different proportion between failure rate and task number, that is, $5 : 100$. In this case, we are considering an environment with a higher number of failures on average. For this reason, the number of master failures is higher, with an average value of 500. Diarchy also increases this value, however, it is clearly lower than its counterpart, with an average value of 50. These results suggest that a MapReduce framework enriched with Diarchy may provide around $10x$ more reliable masters compared to the default YARN Hadoop.

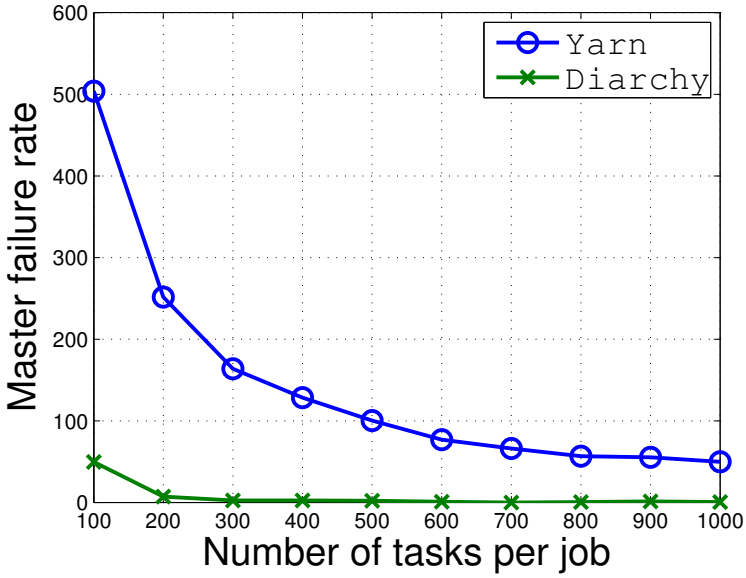


Figure 5-3: Keeping constant $N_F = 5$

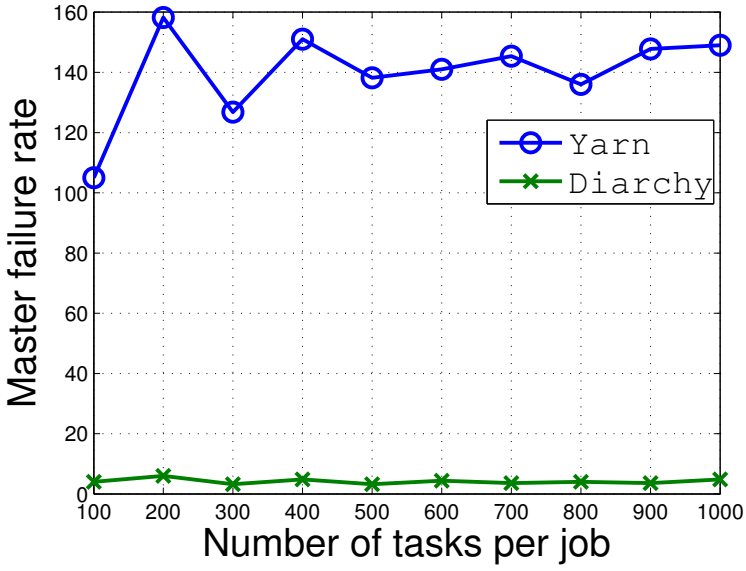


Figure 5-4: $N_F = 5$ per every 350 tasks on average

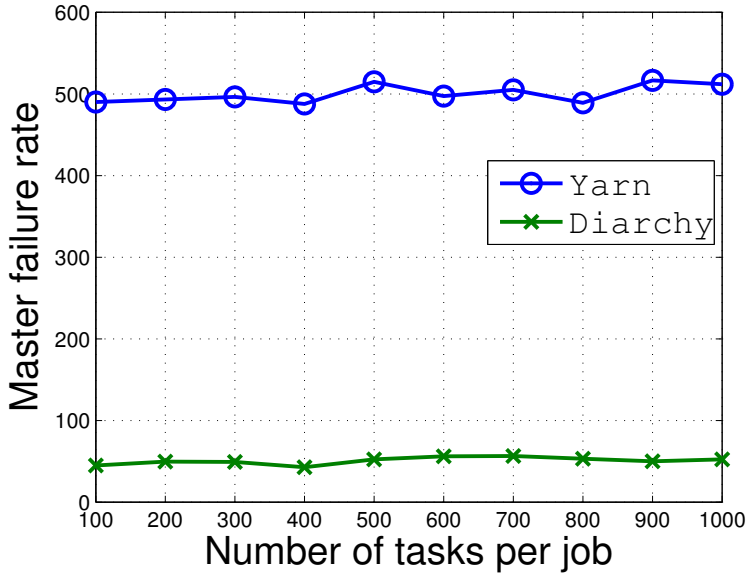


Figure 5-5: $N_F = 5$ per every 100 tasks on average

In addition, we have executed a probability test for a million job cluster. Table 5.1 shows the average value and standard deviation for both YARN and Diarchy. This Table demonstrates the scalability of Diarchy, which outperforms clearly the behavior of YARN, also in this scenario.

Cluster size. This part of the evaluation investigates how Diarchy behaves when we vary the number of jobs in a cluster, evaluating clusters with only 10 daily jobs up to clusters with one million daily jobs. We have made all the experiments keeping the number of tasks constant for a job (we consider that a job consists of 350 tasks).

In this scenario, the number of master failures grows linearly. As shown in Figure 5-6, both YARN and Diarchy follow closely this trend. However, while the gradient of the growth line of YARN is around $1,43\%$, the gradient of the growth line of Diarchy is much lower, around $0,06\%$. This means that, for instance, in the case of 10000 jobs per cluster, YARN has 142 master failures on average, while Diarchy has an average value of 6 master failures. In addition, it is also important to notice that it is difficult that two peer masters fail at the same time.

Table 5.2 shows the results of applying the probability tests with the same number of jobs. The results of the probabilistic test are similar to the results achieved by running the simulation.

Failure rate. This last part of the evaluation describes how Diarchy behaves when we vary the number of failures. At some point, a high failure rate can provoke the degradation of the system, jeopardizing the overall functioning of the framework. In other words, if the number of failures is very high, the probability that failures affect the same worker is also high. In Hadoop, if a failure is repeated more than 4 times, it will imply the job re-execution. As in the previous scenario, we have performed the experiments keeping the number of tasks constant for a job (350 tasks). We have also

Number of tasks	YARN		Diarchy	
	Average	Stdev	Average	Stdev
100	50000	217.945	2500	49.937
200	25000	156.125	625	24.992
300	16666.667	128.019	277.778	16.664
400	12500	111.102	156.25	12.499
500	10000	99.499	100	9.999
600	8333.333	90.906	69.444	8.333
700	7142.857	84.213	51.020	7.143
800	6250	78.810	39.063	6.250
900	5555.556	74.328	30.864	5.555
1000	5000	70.534	25	5.000

Table 5.1: Probabilistic number of master failures, according to the number of tasks per job for a million job cluster.

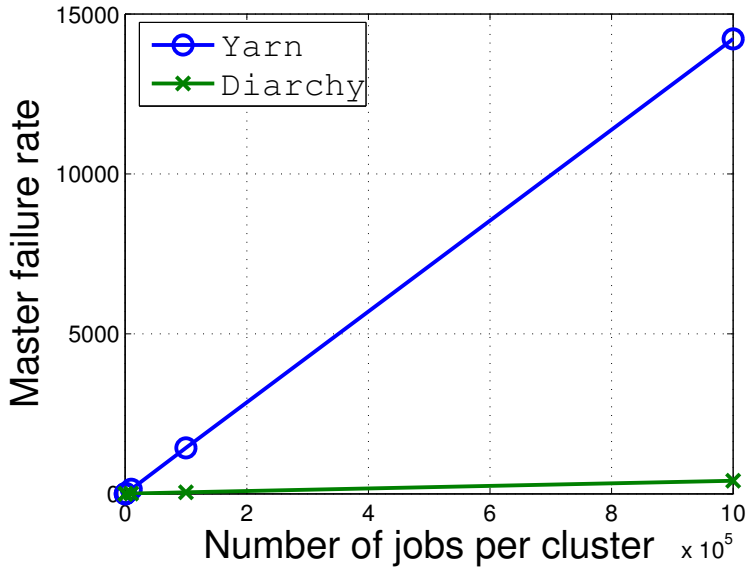


Figure 5-6: Experimental failure rate, according to the number of jobs per cluster.

Number of jobs	YARN		Diarchy	
	Average	Stdev	Average	Stdev
10	0.149	0.375	0.002	0.045
100	1.429	1.187	0.020	0.143
1000	14.286	3.753	0.204	0.452
10000	142.857	11.867	2.041	1.428
100000	1428.571	37.526	20.408	4.517
1000000	14285.714	118.666	204.082	14.284

Table 5.2: Probabilistic failure rate, according to the number of jobs per cluster.

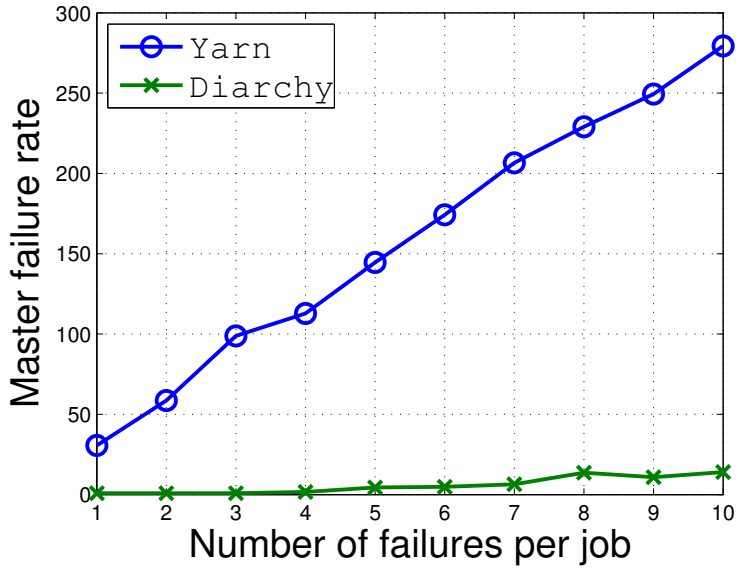


Figure 5-7: Experimental failure rate, according to the average number of failures per job.

kept constant the number of jobs to 10000. As we can see in Figure 5-7, the number of master failures grows also linearly. Here, the proportion between the gradient of the growth line in Diarchy and the gradient of the growth line in YARN is only 5%. This means that, for instance, if we have an average number of failures of 5 per job, the total number of master failures in the case of 10000 jobs is around 145 for YARN, while Diarchy has an average value of around 4 master failures.

Table 5.3 shows the results of the application of the probability tests to a million job simulated cluster. Again, the YARN values increase linearly, in the same way the failure ratio increases in Diarchy, but with a higher slope in the case of YARN.

Number of failures	YARN		Diarchy	
	Average	Stdev	Average	Stdev
1	2857.143	53.376	8.163	2.857
2	5714.286	75.377	32.653	5.714
3	8571.429	92.184	73.469	8.571
4	11428.571	106.292	130.612	11.428
5	14285.714	118.666	204.082	14.284
6	17142.857	129.804	293.878	17.140
7	20000	140	400	19.996
8	22857.143	149.448	522.449	22.851
9	25714.286	158.282	661.224	25.706
10	28571.429	166.599	816.327	28.560

Table 5.3: Probabilistic failure rate, according to the average number of failures per job.

5.4.3 Double parameter tests

This subsection describes how the three parameters (job size, cluster size and failure rate) analyzed previously, are related between them.

Job size vs. failure rate. In large scale data environments, even though there are often short jobs instead of large ones [45], and $N_F = 5$ seems to be a good average value, it is reasonable to assume that not all the environments have the same behavior and that Diarchy can be used in different scenarios. For instance, Diarchy could be used for a supercomputing scenario [43, 66] or in systems that require high reliability [39]. For this reason, it is important to know how Diarchy behaves when different pairs (job size, failure rate) are used, compared to YARN.

As shown in Figure 5-8, the best scenario for Diarchy is achieved when jobs are small, and the failure rate per job is high. Diarchy is able to keep a moderate master failure rate. The highest difference between YARN and Diarchy is achieved when the average failure rate is close to the maximum number of failures per job (10), and the number of tasks per jobs is close to the minimum configured size limit of 100 tasks. In this case, Diarchy outperforms YARN, providing 5x better reliability performance. Moreover, Diarchy behaves better than YARN for any pairs (job size, failure rate).

Cluster size vs. failure rate. This test evaluates the behavior of Diarchy and YARN when we have different clusters that are specialized to run similar jobs, but with different failure rate between them. Figure 5-9 shows this comparison, concluding again that Diarchy provides better reliability results that its counterpart for any combinations (cluster size, failure rate).

The best scenario for Diarchy is achieved when the cluster has the maximum number of jobs (1000000) and the highest failure rate (10). In this case, the reliability performance improvement of Diarchy goes up to 17x.

Cluster size vs. job size. This experiment evaluates as Diarchy and YARN behave in a scenario in which we can have different clusters and different jobs, keeping

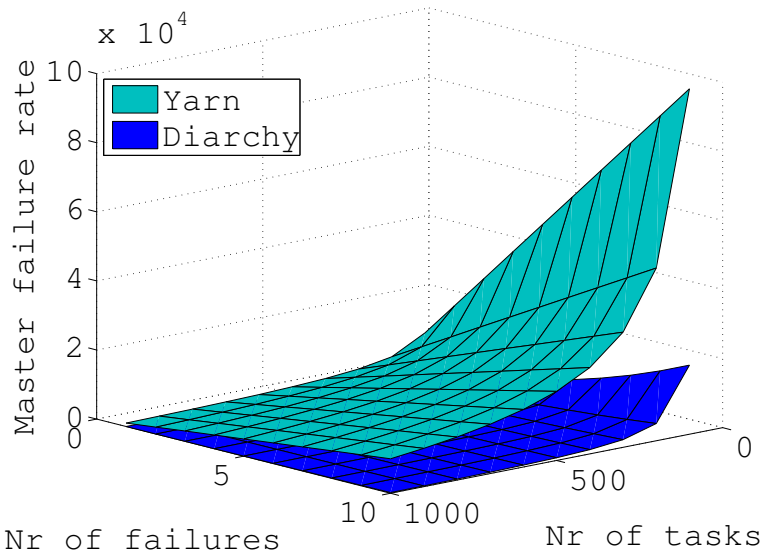


Figure 5-8: YARN vs. Diarchy failure rate compared in a static cluster.

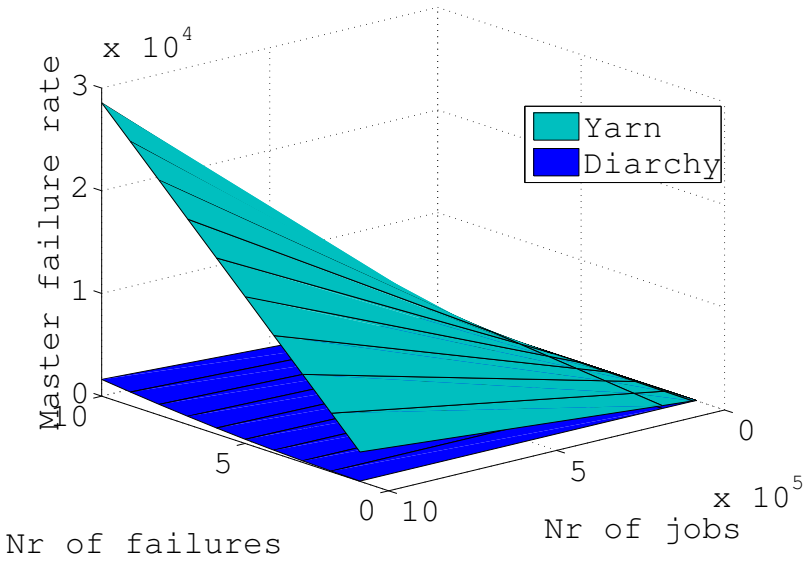


Figure 5-9: YARN vs. Diarchy failure rate compared in static jobs.

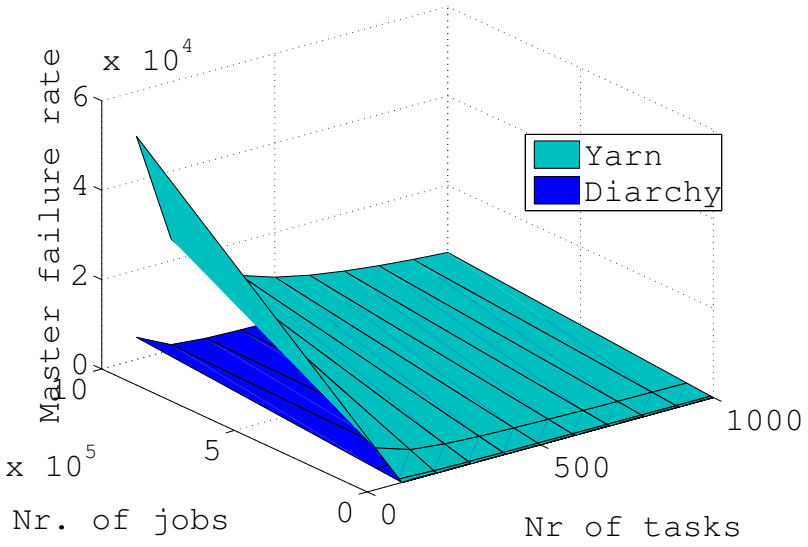


Figure 5-10: YARN vs. Diarchy failure rate compared in static failures.

constant the failure rate ($N_F = 5$). Figure 5-10 shows the results of this evaluation, with the same conclusions than in previous tests, that is, Diarchy outperforms YARN for any configurations (cluster size, job size).

The maximum difference between Diarchy and YARN is given when the cluster size is close to the maximum number of jobs (1000000) and the job size is close to the minimum number of tasks (100). In this case, Diarchy gets a reliability performance improvement of 9x.

5.5 Summary

Hadoop YARN, the so-called next generation MapReduce framework, has tried to solve some of the drawbacks of its predecessor, classic Hadoop. Specifically, YARN has improved the scalability, by means of the separation of each master role into different daemons. However, the reliability of the application master has not been solved yet.

This chapter describes Diarchy, a novel framework that tries to enhance the reliability of YARN, by means of the sharing of responsibilities between two master peers.

We have evaluated Diarchy in a set of experiments, considering three important parameters, i.e., cluster size, job size, and an average failure rate. The intersection of each of the parameters has also been used for the evaluation. As a consequence, we have found that Diarchy outperforms the default YARN framework in all the experimental scenarios, by reaching an improved performance reliability up to 17x at most.

As far as we know, this is the first study to measure the impact of the master

problem in YARN, and the first solution to this problem by means of the definition of MapReduce masters as peers. Diarchy is expected to have wider application in the rest of the YARN daemons, and other master/slave frameworks in general.

Chapter 6

AdaptCont: feedback-based resource allocation system for MapReduce

6.1 Introduction

Cloud Infrastructures usually make use of virtualization techniques. The concept of Virtual Machine (VM) has been present in cloud solutions since the beginning. Many of the cloud-based approaches provide virtual machines to their users to fulfill their processing needs by means of an isolated environment.

Some of the most important cloud frameworks are MapReduce systems [42], which are oriented to batch processing. MapReduce does not need all the power offered by VMs, which are able to emulate a complete hardware and software infrastructure through a full operating system and its add-ons. Instead of this, MapReduce only requires the isolation of a limited set of computing resources for individual processes (map, reduce and other daemons). This capability is provided by *containers*, which, unlike VMs, are oriented to support a single application or process.

A container is an encapsulation of a subset of computing resources, placed on a single node of a cluster. A VM has by far much more overhead than a container, because the former is designed to emulate a virtual hardware through a full operating system and its proper additional add-ons, whereas the latter is designed to run a single application or process. As a result of this, a considerable amount of cloud solutions, not only MapReduce-based clouds, are using currently containers as resource allocation facility. Indeed, many experts are seeing containers as a natural replacement for VMs in order to allocate resources efficiently, although they are far from providing all the features needed for virtualizing operating systems or kernels.

Although containers are a good choice for MapReduce (vs VMs), we state that the containers-based resource allocation given by the state-of-the-art MapReduce approaches, such as Hadoop YARN [101], is not efficient. For each user request in a YARN framework, the container configuration is static in terms of computing resources, no matter the particular requirements of both the request and the application. In order to deal with the different types of requests, YARN containers are usually oversized, decreasing the performance of the system. Moreover, occasionally

containers do not have sufficient resources for addressing the request requirements. This could lead to unreliable situations, jeopardizing the correct working of the applications. For the sake of simplicity, we only consider the main computing resources, the main memory (RAM) and the Computing Processing Unit (CPU).

We present a novel approach for optimizing the resource allocation at the container level in MapReduce systems. This approach, called *AdaptCont*, is based on feedback systems [23], due to its dynamism and adaptation capabilities. AdaptCont is in charge of choosing the amount of resources needed by a specific container, depending on several parameters, such as the real-time request input, the number of requests, the number of users and the dynamic constraints of the system infrastructure, such as the set of resources available. We define two different selection approaches, Dynamic AdaptCont and Pool AdaptCont. Whereas Dynamic AdaptCont calculates the exact amount of resources per each container, Pool AdaptCont chooses a predefined container from a pool of available configurations.

In order to validate our approach, we use AdaptCont for a particular case study on a particular MapReduce system, the Hadoop YARN. We have chosen the Application Master of Hadoop YARN instead of the YARN workers, because of the importance of this daemon and because it involves the most complex use of containers. The application master container is required in every application. Additionally, the master orchestrates its proper job, but its reliability can jeopardize the work of the job workers. On the other side, a particular worker usually does not have impact on the reliability of the overall job, although it may contribute to the delay of the completion time. The experiments show that our approach brings substantial benefits compared to the default mechanism of YARN, in terms of use of RAM and CPU. Our evaluation shows improvements in the use of these resources, that ranges from 15% to 75%.

As a summary, this chapter has the following main contributions:

1. Definition of a general-purpose framework called AdaptCont, for the resource allocation at the container level in MapReduce systems.

2. Instantiation of AdaptCont for a particular case study on Hadoop YARN, that is, the application master container.

3. Evaluation of AdaptCont and comparison with the default behavior of Hadoop YARN.

The rest of the chapter is organized as follows. In Section 6.2 we introduce AdaptCont as a general framework based on feedback systems for allocating container resources. We introduce a case study of the framework in Section 6.3. We evaluate AdaptCont in Section 6.4. Finally, we summarize the main contributions in Section 6.5.

6.2 AdaptCont model and design

According to [23], feedback systems refer to two or more dynamical systems, which are interconnected in such way that each system affects the behavior of others. Feedback

systems may be open or closed. Assuming a feedback system F, composed by two systems A and B, F is closed if their components form a cycle, being the output of system A the input of system B, and the output of system B the input of system A. On the contrary, F is open when the interconnection between systems B and A is broken.

Feedback systems are based on a basic principle: correcting actions should always be performed on the difference between the desired and the actual performance. Feedback allows us to (i) provide robustness to the systems, (ii) modify the dynamics of a system by means of these correcting actions and (iii) provide a higher level of automation. When a feedback system is not properly designed, a well known drawback is the possibility of instability.

An example of a dynamic system that can benefit from the feedback theory nowadays is a production cloud [20]. In this scenario, users, applications and infrastructure are clearly interconnected and the behavior of one of these systems influence each other. Our approach, AdaptCont, is a Feedback system, whose main goal is to optimize the resource allocation at the container level in clouds and specifically in MapReduce-based systems.

Before designing the Feedback System, it is necessary to define the features of a cloud. Namely:

- A cloud has a limited set of nodes $n_1, n_2, \ldots n_m$.
- Each node n_i has a limited set of containers $c_i1, c_i2, \ldots c_il$.
- The system can receive a limited set of job requests $j_1, j_2, \ldots j_r$.
- Every job request has its workload input. These jobs are part of applications.
- The same workload can be used as an input for different applications.
- Applications could divide a large workload in small input partitions called *splits*, each split being a workload of a particular container.
- Depending on the cluster size and scheduler limitations, simultaneous containers could run in a single or multiple sequential groups called *waves*.
- By default, all the containers should finish before the application submits the final output to the user.
- Applications may introduce different job completion time, even though under the same user, input and allocated resources.

In a dynamic cloud, these parameters may change in real-time. Detecting these changes is strongly dependent on the monitoring system, which should be particularly focused on the infrastructure. In order to perform this monitoring process, we have developed GMonE [79], a general-purpose cloud monitoring tool that provides a better performance compared to those present in state-of-the-art systems such as Amazon EC2 [3] and OpenNebula [80].

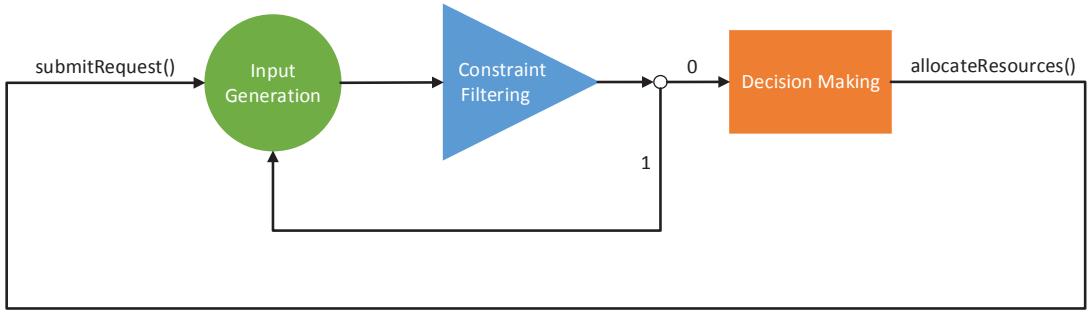


Figure 6-1: A generalized framework for self-adaptive containers, based on the feedback theory.

At a generic level, we can follow a feedback-based approach based on three stages: Input Generation, Constraint Filtering and Decision Making. The general pattern is shown in Figure 6-1. This approach is closed. Before running the Decision Making module, a dynamic system may need several runs of the Input Generation module, applying the dynamic constraints that can be arisen. This number of runs of the Input Generation module is proportional to the modifications (constraints) identified from the system.

6.2.1 Input Generation

The Input Generation module of AdaptCont collects or generates the required parameters for making decisions about efficient resource allocation. These parameters are:

- The input workload size.
- The input split size enforced by the application.
- The total number of available containers per each user.
- The wave size in which these containers may be run.
- The constraints introduced by users.

Some of these parameters are collected directly from the application. For instance, the input workload size comes in every job request. Other parameters are more complex to be generated. For instance, the number of waves w depends on the number of input splits n_s and the number of available containers per user n_c, being calculated as $w = n_s/n_c$.

6.2.2 Constraint Filtering

This stage is needed because clouds have a limited number of costly resources. Constraints may be imposed by the infrastructure, application and/or users.

Infrastructure constraints are those constraints related to the limitation of the cloud provider, since not always the number of resources is enough for fulfilling the resource requests of all the applications and users.

Some constraints are enforced by applications. For instance, some applications require a certain type of sequential containers. This is the case of MapReduce systems, where by default, containers of the first phase (map) need to finish before the containers of the second phase (reduce) start [102, 60].

Finally, other constraints are defined by users. For instance, some users have a limited capability for buying resources.

6.2.3 Decision Making

Based on the parameters coming from the previous modules, the Decision Making module outputs the final resource allocation. In particular, this module decides the minimum recommended container memory c_{RAM} and CPU power c_{CPU} per every container. This decision depends on the particular problem addressed by these containers.

Once this module has decided these values for a specific application of a user, the rest of the process is automatic, since all the containers of an application are equal. This process has to be called for different applications or different users.

6.2.4 Predefined Containers

A possible improvement of AdaptCont is enabling the use of predefined containers with different configurations (e.g. small, medium, large). This means that a cloud has a pool of static containers that can be used for different user request. In this way, it will not be necessary to trigger a new container, but a predefined one ready to be used. This reduces the overhead of the resource allocation process during the job submission. This feature should be part of the Decision Making module.

How can the framework define this pool of containers? First, it should be able to identify the typical user requests in the system. These requests may be evaluated from (i) previous (stored) monitoring values, or (ii) from other monitoring variables measured at the same time, according to [93].

What happens if the container does not have the exact configuration we need? In this case, the Decision Making module establishes a threshold. If the difference between the required and existing configurations is below this threshold, the system uses the already existing container. Otherwise, the system triggers a new container.

6.3 Case study: YARN application master

We have chosen as a case of study the analysis of a relevant type of a container in a specific kind of cloud systems, that is, MapReduce-based clouds. Namely, the chosen container is the application master in the next-generation MapReduce system called YARN [101].

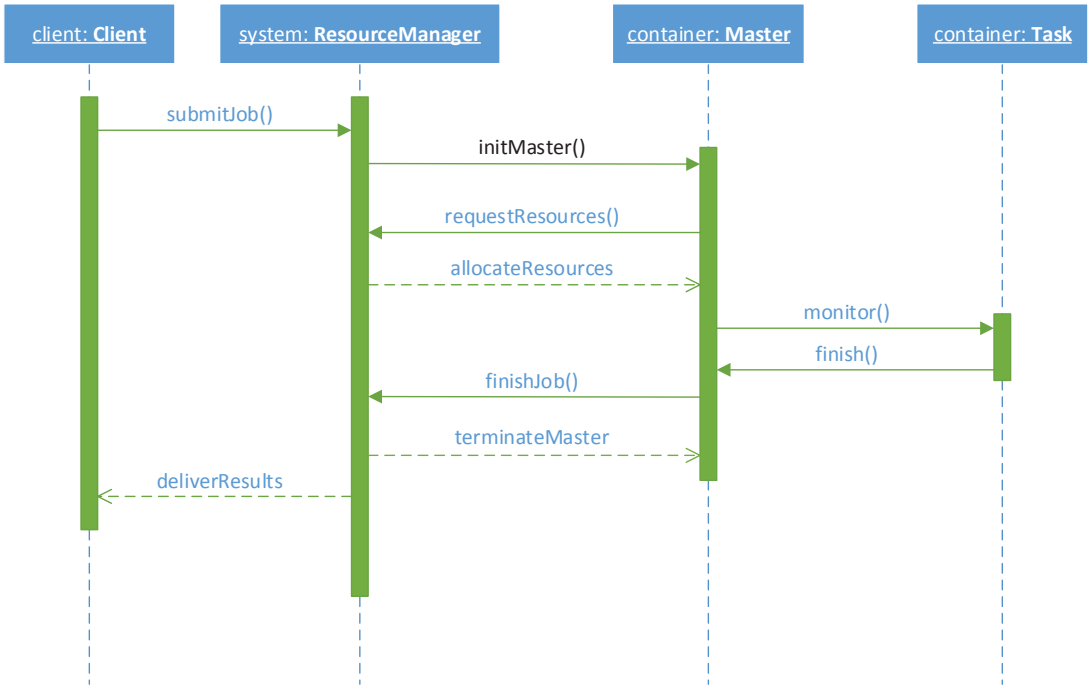


Figure 6-2: Job flow messages in Hadoop YARN: A sequence diagram.

6.3.1 Background

YARN constitutes the new version of Apache Hadoop. This new implementation was built with the aim of solving some of the problems shown by the old Hadoop version. Basically, YARN is a resource management platform, that unlike the former Hadoop release, provides greater scalability, higher efficiency and enables different frameworks to efficiently share a cluster. YARN offers, among others, MapReduce capabilities.

The basic idea behind YARN is the separation between the two main operations of the classic Hadoop master, resource management and job scheduling/monitoring, into separate entities or daemons. The resource manager consists of two main components: the *scheduler* and the *application manager*. While the scheduler's duty is resource allocation, the application manager accepts job submissions, and initiates the first job container for the application master. After this, the job is managed by the application master, which starts negotiating resources with the resource manager and collaborates with the node managers to run and monitor its tasks. Finally, it informs the resource manager that has completed, and releases its container. The resource manager delivers the results to the client. A simple sequence of these steps is given in Figure 6-2.

For each job submission, the application master configuration is static and does not change for different scenarios. According to the state-of-the-art literature [68, 45,

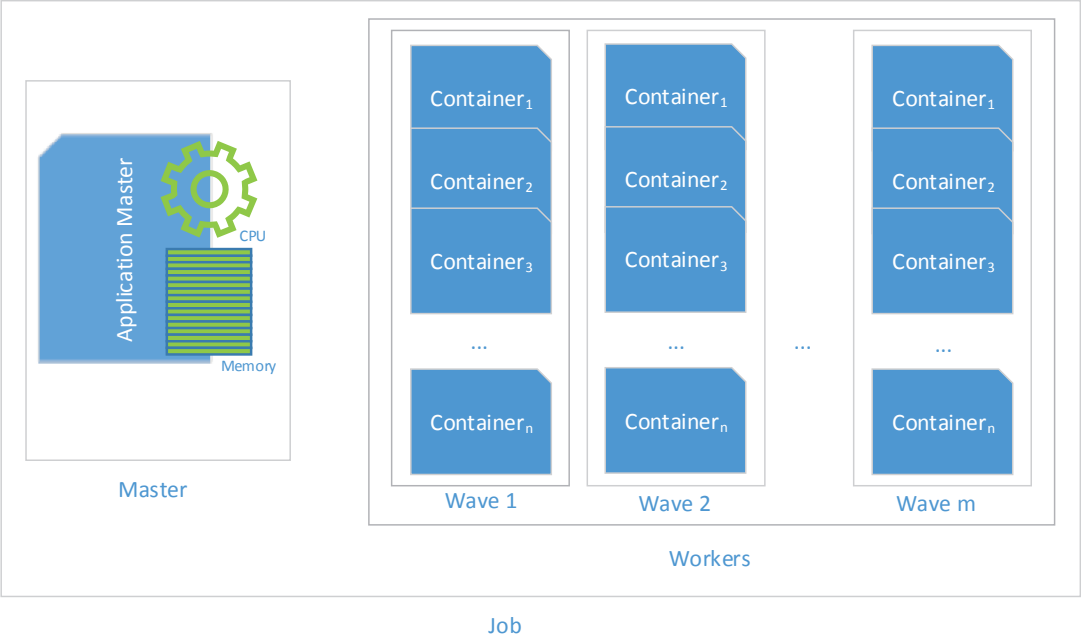


Figure 6-3: Workers containers monitored in waves by the application master container.

19, 16, 46], most large-scale MapReduce clusters run small jobs. As we will show in Section 6.4, even the smallest resource configuration of the application master exceeds the requirements of these workloads. This implies a waste of resources, which could be alleviated if the configuration is adapted to the workload size and the infrastructure resources. Moreover, some big workloads could fail if the container size is not enough for managing them. At large-scale level, this would have a higher impact. Therefore, our goal is to choose an appropriate container for the application master.

6.3.2 AdaptCont applied to YARN

In order to optimize containers for the application master we will follow the same pattern of the general framework, that is, AdaptCont.

The input generation module divides the workload input size into splits. The YARN scheduler provides containers to users, according to the number of available containers of the infrastructure each instant of time. As we mentioned above, the input generation module calculates the number of waves from the number of input splits and the number of available containers per user. Figure 6-3 shows how the application master manages these waves.

Many constraints can be raised from the scheduler. An example of this is the phase priority. It is well known that the map phase input is by default bigger or equal to the reduce phase input [105]. This is one of the reasons why the number of mappers is higher than the number of reducers. Due to this, as a reasonable constraint, the

constraint filtering module prioritizes the number of mappers with regards to the number of reducers.

Decision making module considers mainly two parameters, total workload and wave sizes. Contrary to what it may seem at first sight, the type of applications does not affect the resource allocation decision of our use case. Some applications could have more memory, CPU or I/O requirements, influencing the number and types of needed containers. However, this would only determine the size of the worker containers, and in this case study, our scope is focused only on the master containers, which contribute largely to the reliability of the application executions.

Decision making module uses two parameters: Ω and Ψ. The first parameter represents the minimum recommended memory size for an application master container that manages one unit wave, w_{unit}. Our goal is to calculate c_{RAM} from the value of Ω, being c_{RAM} the recommended memory size for the application master. In the same way, we aim to calculate c_{CPU} as the recommended CPU power for the application master, from Ψ, which is the minimum recommended CPU power for an application master that manages w_{unit}.

To calculate the memory, if the actual wave w is bigger than what could be handled by Ω, that is, bigger than w_{unit}, then we declare a variable λ that measures this wave magnitude: $\lambda = w/w_{unit}$. Now, it is easy to find the c_{RAM}:

$$c_{RAM} = \lambda * \Omega + Stdev, Stdev \in [0;\ \Omega/2] \tag{6.1}$$

Regarding the CPU power, the formula for c_{CPU} is:

$$c_{CPU} = \lambda * \Psi + Stdev, Stdev \in [0;\ \Psi/2] \tag{6.2}$$

Figure 6-4 represents the AdaptCont modules, which are executed in the context of different YARN daemons. Whereas the input generation and the decision making modules are part of the application manager, the constraint filtering module is part of the scheduler. The combination of both daemons forms the resource manager. The resource manager has a complete knowledge about each user through the application manager and the available resources through the scheduler daemon. When the application manager receives a user request, the resource manager is informed about the workload input. The scheduler informs the application manager of every important modification regarding the monitored cluster. According to this, the application manager reacts upon the user request, by optimizing the container for its application master.

6.4 Experimental evaluation

We have performed a set of experiments to validate our approach and compare it with YARN. These experiments have been made by means of a round-based simulator. In order to make this evaluation, we have followed the methodology of Section 6.4.1. Results of the evaluation are described in Section 6.4.2. Finally, the discussion about these results is shown in Section 6.4.3.

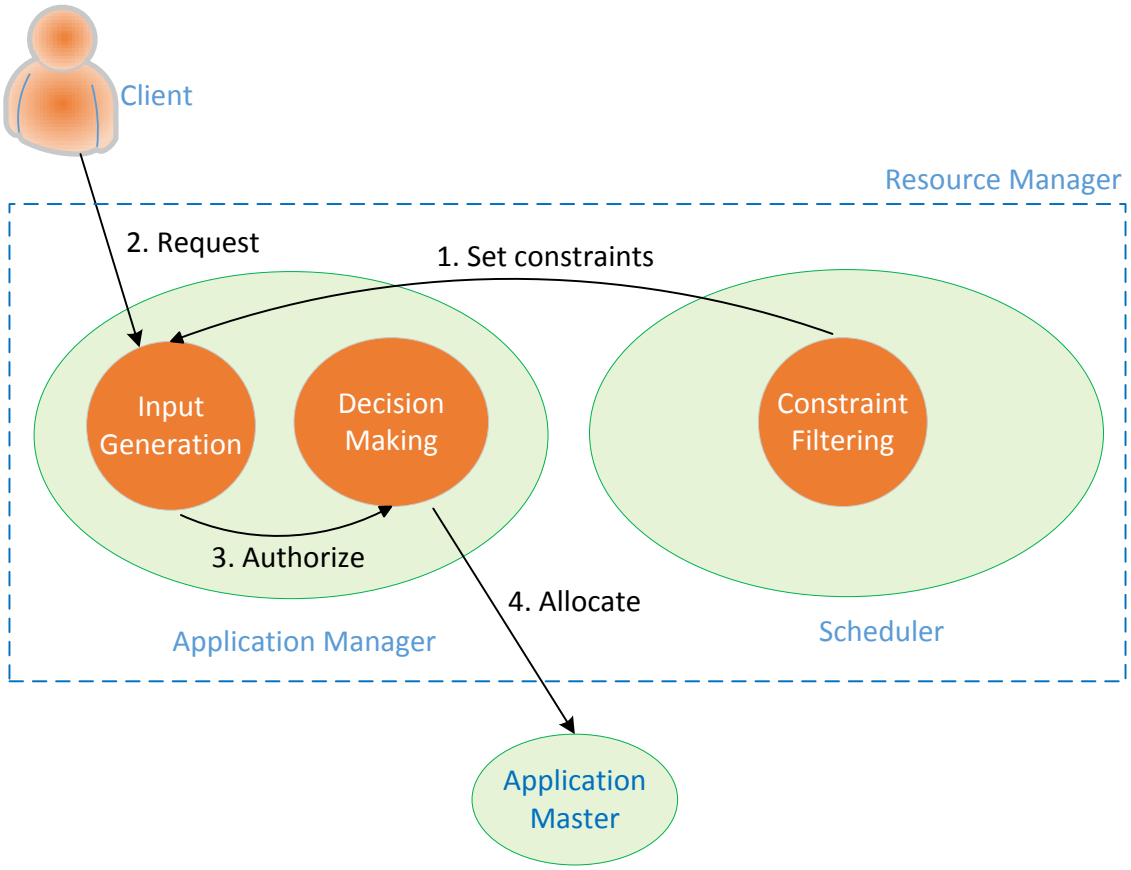


Figure 6-4: AdaptCont model applied to the Hadoop YARN application master.

Scheduler	Master		
	YARN	Dynamic	Pool
FIFO	FIFO-YARN	FIFO-Dynamic	FIFO-Pool
Fair	Fair-YARN	Fair-Dynamic	Fair-Pool
Capacity	Capacity-YARN	Capacity-Dynamic	Capacity-Pool

Table 6.1: Methodology description, taking into account different schedulers and masters. **FIFO**: FIFO scheduler. **Fair**: Fair scheduler. **Capacity**: Capacity scheduler. **YARN**: YARN master. **Dynamic**: Dynamic master. **Pool**: Predefined containers-based master.

6.4.1 Methodology

To evaluate AdaptCont, we have considered three different schedulers and three different application master configurations, as is shown in Table 6.1. Below we give details for all of them.

Scheduler. We have taken into account three important schedulers, already implemented in YARN:

- FIFO scheduler. This was the first scheduling algorithm that was implemented for MapReduce. It works on the principle that the master has a queue of jobs, and it simply pulls the oldest job first.
- Fair scheduler. It assigns the same amount of resources (containers) to all the workloads, so that on average every job gets an equal share of containers during its lifetime.
- Capacity scheduler. It gives different amount of resources (containers) to different workloads. The bigger the workload is the more resources are allocated to it.

Master. To compare YARN with AdaptCont, we use the following application master configurations:

- YARN application master (YARN). This is the default implementation of the application master in YARN.
- Dynamic master (Dynamic AdaptCont). This master container is adjusted in accordance with AdaptCont. Namely, it calculates the memory and CPU, according to the decision making module and only after this, it initiates the master.
- Predefined containers-based master (Pool AdaptCont). As defined in Section 6.2.4, the resource manager has a pool of master containers, which can be allocated depending on the workload size. This is an optional optimization of AdaptCont.

Workload. According to the job arrival time, we consider two additional sets of experiments:

- *Set-All.* In this scenario, all the jobs are already in the queue of the scheduler. We are going to combine this scenario with all the values of the Table 6.1, since it is important to evaluate the approach under pressure, that is, when the load reaches high values.

- *Set-Random.* This is a more realistic scenario, where jobs arrive at random times. Again, this scenario is evaluated in combination with all the values of the Table 6.1, in order to simulate the behavior of a common MapReduce cluster.

An important parameter to take into account is the workload size. We introduce two additional scenarios:

- *Workload-Mixed.* In this case, the workload size will be variable, ranging from 500 MB to 105 GB, taking (1) 500 MB, (2) 3.5 GB, (3) 7 GB, (4) 15 GB, (5) 30 GB, (6) 45 GB, (7) 60 GB, (8) 75 GB, (9) 90 GB, and (10) 105 GB as workload size inputs. We have used these boundaries, because of the average workload sizes of important production clusters. For instance, around 90% of workload inputs in Facebook [19] are below 100 GB.

- *Workload-Same.* In this case, every input (10 workloads) is the same: 10 GB. We have used this value, since, on average, the input workloads at Yahoo and Microsoft [19] are under 14 GB.

Therefore, we evaluate AdaptCont with the values of the Table 6.1 and the 4 combinations from previous scenarios: *Set All - Workload Mix*, *Set All - Workload Same*, *Set Random - Workload Mix*, and *Set Random - Workload Same*.

Constraints. In MapReduce, the application master has to manage both map and reduce workers. The map phase input is always bigger or equal to the reduce phase input [105]. This is one of the reasons why the number of mappers is bigger than the number of reducers. On the other hand, both phases are run sequentially. Thus, we can assume as constraint that the master container resources depend on the number of mappers and not on the number of reducers.

In order to simulate a realistic scenario, we have introduced in our experiments a partition failure that will impact around 10% of the cluster size. We assume that this failure appears in the fifth iteration. This constraint forces AdaptCont to react in real-time and adapt itself to a new execution environment, having to make decisions about future resource allocations.

Setup. In our experiments, 250 containers are used for worker tasks (mappers and reducers). This number of containers is sufficient to evaluate the approach, considering 25 containers per workload. We consider that every map and reduce container is the same, and can execute a particular portion (split) of the workload. Each task runs on a container that has 1024 MB RAM and 1 virtual core. According to [4, 61, 110], a physical CPU core is capable of giving optimal performance of the container, if

simultaneously processes 2 containers at most. Therefore, we take 1 CPU core as equivalent to 2 virtual cores.

Our goal is to evaluate the resource utilization of the application masters, in terms of CPU and RAM. To get this, we consider an isolated set of resources oriented only to application masters. In this way, it will be easier to measure the impact of AdaptCont in saving resources.

6.4.2 Results

In this section, we compare the CPU and memory efficiency of YARN vs Dynamic AdaptCont and Pool AdaptCont. Before that, we analyze the wave behavior of the 10 workloads.

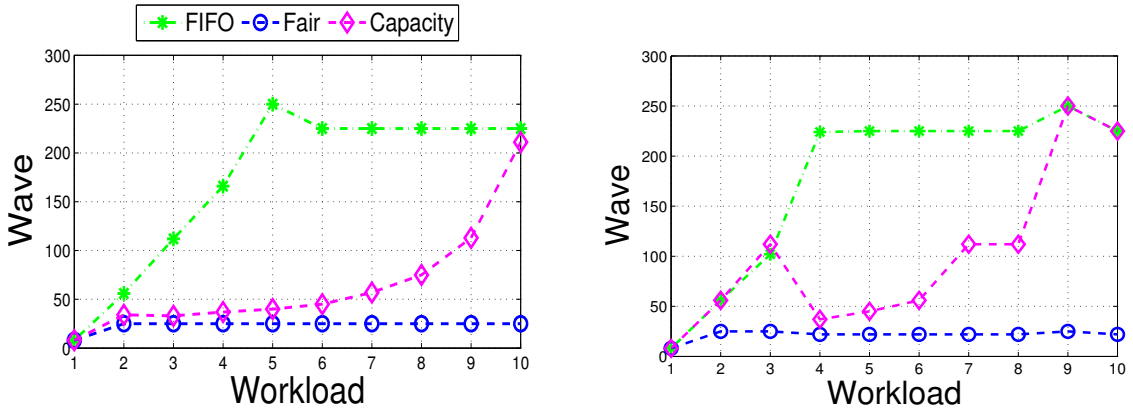
Wave behavior. Figure 6-5 represents the resource allocation (maximum number of container or wave size) for the combination we have mentioned before: *Set All - Workload Mix*, *Set All - Workload Same*, *Set Random - Workload Mix*, and *Set Random - Workload Same*.

Figure 6-5a shows different workload sizes with the same arrival time (already in the scheduler queue). The experiments demonstrate that a maximum wave is dependent on the workload size and the scheduler. Regarding the FIFO scheduler, since the queue order is formed by the smallest workload first, for these small workloads, the maximum wave is represented by the needed containers. For instance, the first workload needs only 8 containers. This number of containers is calculated dividing the workload size by the split size (64 MB). These 8 containers are provided by the infrastructure, and this is the case of the second workload (56 containers), and the third workload (112 containers). For the fourth workload, the infrastructure is not capable to provide the needed containers, which only has 74 containers in the first iteration, that is, $250 - (8 + 56 + 112)$. The fourth workload needs 240 containers in total. Thus, the remaining containers ($240 - 74 = 166$) will be provided in the next iteration.

In the second iteration, since the first three workloads have finished, the scheduler will provide 166 containers to the fourth workload and the rest ($250 - 166 = 84$) to the fifth workload. This process is repeated until all the workloads are given the necessary containers and every job has terminated. As we can notice, the maximum wave for the latest workloads reaches higher amount of allocated containers, since the workload is bigger, and in most of the cases the scheduler is busy with a unique job. Although initially the infrastructure has 250 containers, from the fifth iteration there is a slight decrease (225), due to the partition failure (10% of the resources). This only affects the workloads not having finished before this iteration (in this case, fifth).

The main drawback of the FIFO scheduler is that it may delay the completion time of the smallest jobs, especially if they arrive late to the queue. In general, this scheduler is not fair in the resource allocation and depends exclusively on the arrival time.

Regarding the fair scheduler, this scheduler allocates the same number of containers to all the workloads and consequently to all the users, that is, $250/10 = 25$. The



(a) Set All - Workload Mix

(b) Set Random - Workload Mix

(c) Set All - Workload Same

(d) Set Random - Workload Same

Figure 6-5: Wave behavior: Wave size according to the scheduler and the workload type.

partition failure forces the fair scheduler to decrease the number of containers to 22 (225/10) from the fifth iteration.

With regards to the capacity scheduler, this scheduler takes advantage of available resources once some jobs have finished. At the beginning, it behaves like the fair scheduler. However, when some small jobs have terminated, the available resources can be reallocated to the rest of workloads. This is the reason why the biggest workloads in the queue get a higher number of containers. As in the previous case, the partition failure also implies a slight decrease in the number of containers from the fifth iteration.

Figure 6-5b represents the same mixed workloads but when they arrive randomly to the scheduler queue. In our case, the random selection injects the workloads in this order: (9), (3), (2), (1), (7), (10), (6), (8), (4), (5). The first workload (9) is injected in the first iteration, the second workload (3) in the second iteration and so on. Clearly, the main differences are noted in the FIFO scheduler, because the arrival time of the workloads is different and now one of the biggest workloads (9) appears in first place.

The other subplots of Figure 6-5 show the experimental results of the same workloads with an input of 10 GB, This input requires a static number of containers. In this case, 160 containers.

In Figure 6-5c, all the jobs have arrived to the queue. In this scenario, the FIFO allocation oscillates between the maximum wave of 160 containers and the smallest wave of 90 containers $(250 - 160)$. This oscillation is caused by the allocation of resources to the previous workload, which does not leave enough resources for the next one and then the cycle is repeated again.

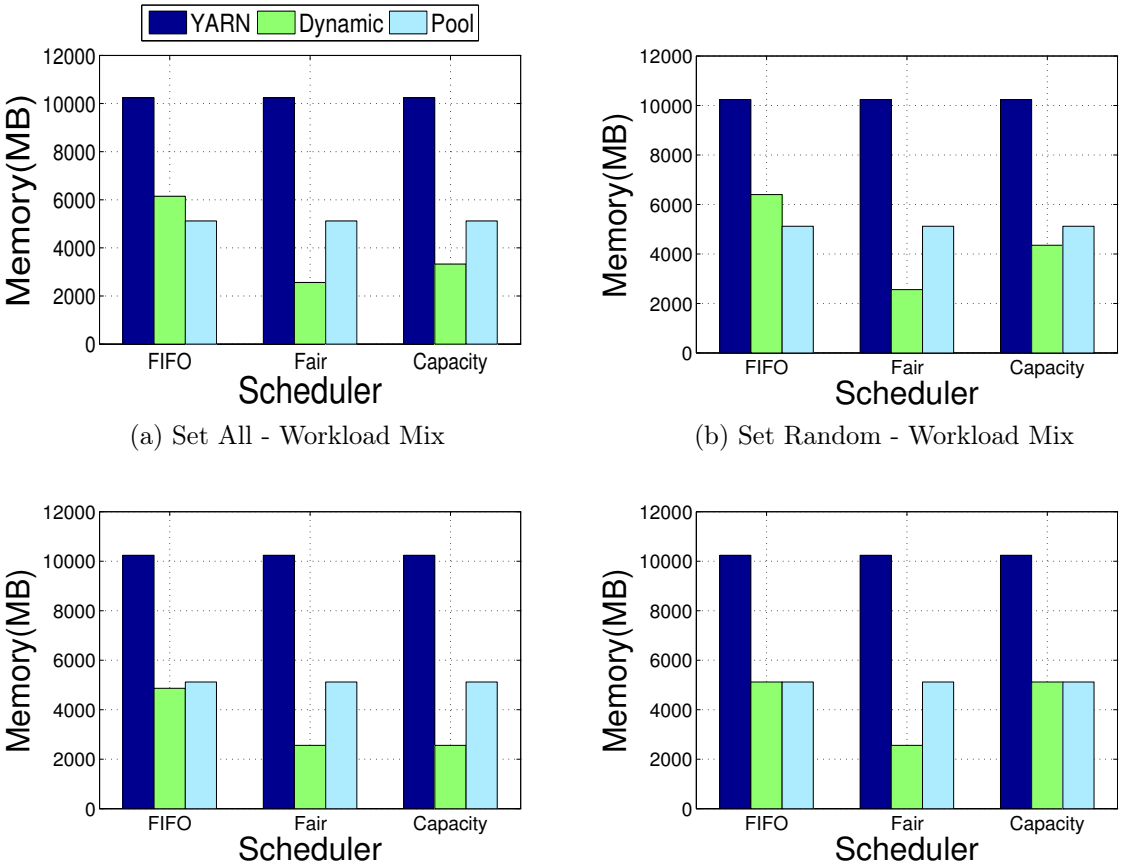
In this case, the fair and capacity schedulers have the same behavior, since all the workloads are equal.

Figure 6-5d shows the number of containers for the same workload with random arrival. The difference of this scenario versus the scenario shown in Figure 6-5c is twofold:

1. The arrival of these jobs is consecutive. In every iteration, a job arrives. Due to this, the FIFO scheduler is forced to wait after each round for a new workload, even though at every round there are available resources $(250 - 160 = 90)$, not allocated to any job. Thus, the FIFO scheduler always allocates 160 containers in every iteration.

2. Whereas in the previous scenario, the fair and capacity schedulers behaves the same, in this case, the capacity scheduler acts similar to the FIFO scheduler. This is because the capacity scheduler adapts its decisions to the number of available resources, which is enough in every moment for addressing the requirements of the jobs (160 containers). Thus, the capacity scheduler achieves a better completion time, compared to the fair scheduler.

According to this analysis, we can conclude that the wave behavior and size are decisive in the application master configuration.



(a) Set All - Workload Mix

(b) Set Random - Workload Mix

(c) Set All - Workload Same

(d) Set Random - Workload Same

Figure 6-6: Memory usage and master type versus scheduler.

Memory usage. Figure 6-6 shows for the 4 scenarios the total memory used by the three approaches: YARN, Dynamic AdaptCont and Pool AdaptCont.

In the case of YARN, we have deployed the default configuration, choosing the minimum memory allocation for the application master (1024 MB).

The Dynamic AdaptCont-based application master memory is dependent on the waves size. If the wave size is under 100, the Decision Making module allocates a minimum recommended memory of 256 MB. For each increase of 100 in the wave size, the memory is doubled. The reasons behind this are:

1. A normal Hadoop task does not need more than 200 MB [19], and this is even clearer in the case of the application master.

2. As most of the jobs are small [19, 16, 46], consequently the maximum number of mappers is also small and therefore, the application master requires less memory.

3. The minimum recommended memory by Hortonworks [61] is 256 MB.

The Pool AdaptCont-based application master works in a different way, constituting an alternative between the YARN master and the Dynamic master. This application master has three default configurations: *small*, *medium* and *big*. The small master has 512 MB of memory, for all small jobs that need a maximum of 250 containers. The medium master has 1024 MB, as it is the default minimum YARN setting. In order to deal with big waves, the big configuration has 2048 MB.

As we can see in Figure 6-6, YARN is outperformed by both AdaptCont approaches. YARN always consumes 10GB, not depending on the different use cases. For instance, in Figure 6-6a, Dynamic AdaptCont has a memory usage of 6144 MB versus 10 GB in YARN, achieving 40% memory improvement. In this case, Pool AdaptCont only uses 5120 MB, i.e. 50% improvement compared to YARN. This difference between Dynamic AdaptCont and Pool AdaptCont for the FIFO scheduler is due to the way of providing memory in both approaches. If the workload needs 250 containers, Dynamic AdaptCont provides $256\lceil(250/100)\rceil$ MB, that is, $256 * 3 = 768$ MB. In the same scenario, Pool AdaptCont provides 512 MB, corresponding to the small size configuration.

In general, Dynamic AdaptCont is the best approach in terms of memory usage, except in the case of the FIFO scheduler, where the performance is close and slightly worse than the performance of Pool AdaptCont. In the case of Fair and Capacity schedulers, Dynamic AdaptCont is the best alternative, achieving on average 75% and 67.5% improvement compared to YARN, versus 50% improvement provided by Pool AdaptCont.

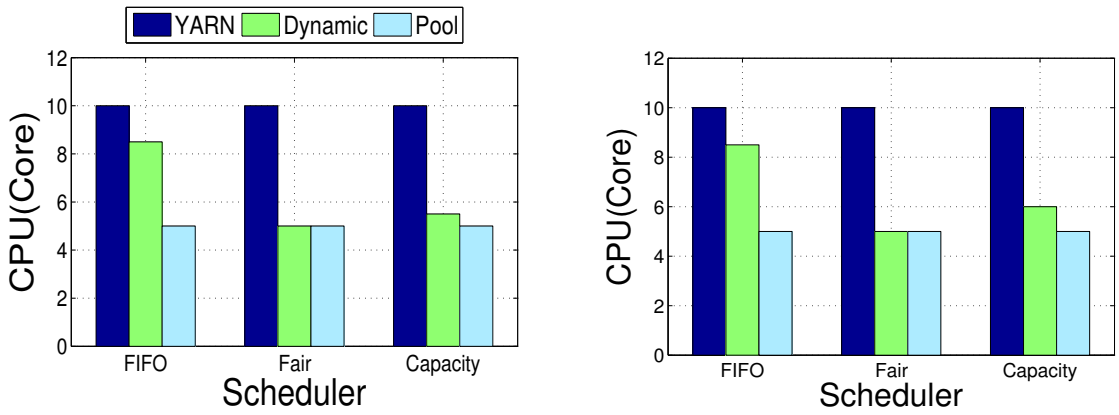
CPU usage. The CPU usage is another relevant parameter to take into account. In order to measure it, we have correlated memory and CPU, considering that we need higher CPU power to process a larger amount of data, stored in memory.

In YARN, you can assign a value ranging from 1 up to 32 of virtual cores for the application master. This is also the possible interval allocation for every other container. According to [4], 32 is the maximum value. In our experiments, we use the minimum value for the YARN master (1 virtual core for its container) per 1024 MB.

For the Dynamic AdaptCont, the Decision Making module increases the number of virtual cores after two successive increments of 256 MB of memory. This decision is based on the above-mentioned methodology, which states that a physical CPU core is capable of giving optimal performance of the container, if simultaneously processes 2 containers at most [4, 61, 110]. To be conservative, we address the smallest container, that is, a container of 256 MB. For instance, if the memory usage is 768 MB, the chosen number of virtual cores is 2.

The same strategy is valid for the Pool AdaptCont, assuming 1 virtual core for small containers, 2 virtual cores for medium containers and 3 virtual cores for large containers.

Due to this policy, the CPU does not change so abruptly as the memory for Dynamic and Pool AdaptCont. Thus, as is shown in Figure 6-7, both approaches behave similar, except in the case of FIFO with Workload Mix. This was previously justified in the memory usage evaluation. As the CPU is proportional to the memory usage, the behavior of Dynamic AdaptCont with FIFO for Workload Mix is again



(a) Set All - Workload Mix

(b) Set Random - Workload Mix

(c) Set All - Workload Same

(d) Set Random - Workload Same

Figure 6-7: CPU usage and master type versus scheduler.

repeated in the case of CPU.

In most of the cases, the improvement of both Dynamic and Pool AdaptCont against YARN reaches 50%.

6.4.3 Discussion

In this section, we discuss what combination of approaches and schedulers can be beneficial in common scenarios.

As a result of the experiments, we can conclude that YARN used by default is not appropriate for optimizing the use of MapReduce-based clouds, due to the waste of resources.

In the presence of heavy and known in advanced workloads (this is the usual case of scientific workloads), according to our results, the best recommended strategy is to use Dynamic AdaptCont combined with FIFO scheduler.

However, if we have limited resources per user, a better choice could be Dynamic AdaptCont combined with Fair scheduler. This scheduler allocates a small set of resources to every workload, improving the overall performance.

In a scenario where we have a mixture of large and small workloads, the choice should be Dynamic AdaptCont combined with Capacity scheduler. This is due to the adaptability of this scheduler with regards to the input workload and available resources.

Finally, as shown in the experiments, if our focus is on CPU and not in memory, we can decide to use Pool AdaptCont (combined with any schedulers) instead of the Dynamic approach.

6.5 Summary

This chapter proposes AdaptCont, a novel optimization framework for resource allocation at the container level, based on feedback systems. As part of the framework, two selection methodologies have been introduced, Dynamic AdaptCont and Pool AdaptCont. Whereas Dynamic AdaptCont calculates the exact amount of resources per each container, Pool AdaptCont has the ability to choose a predefined container from a pool of available configurations. The evaluation results demonstrate that AdaptCont outperforms the default resource allocation mechanism of YARN in terms of RAM and CPU usage, the gains ranging from 40% to 75% for memory usage and from 15% to 50% for CPU utilization. Our experiments have allowed us to use an extensive methodology, including 3 different schedulers, 10 different workloads, random arrival times and the introduction of partition failures.

To the best of our knowledge, this is the first contribution that acknowledges the impact of resource utilization problem in MapReduce approaches at container level, and additionally proposes a solution to this problem by means of an alternative framework.

Part IV

Conclusions

Chapter 7

Conclusions and future work

This chapter summarizes the main contributions, discusses the future directions and introduces the main publications resulting from this thesis.

7.1 Conclusions

Data-intensive frameworks, such as Hadoop MapReduce, have as main goal the processing of an enormous amount of data in a short time, by transmitting the computation where the data resides. In failure-free scenarios, these frameworks usually achieve good results. However, this is not a realistic scenario. In addition, these frameworks exhibit some fault tolerance and dependability techniques as built-in features.

In general, dependability improvements are known to imply additional resource costs. This is reasonable and providers offering these infrastructures are aware of this. However, not all the approaches provide the same tradeoff between fault tolerant capabilities (or more generally, reliability capabilities) and cost. In this thesis, we try to address the coexistence between reliability and resource efficiency in MapReduce-based systems, looking for methodologies that introduce the minimal cost, guaranteeing an appropriate level of reliability.

In particular, in the context of this thesis we have the following three contributions, which answers directly to the three research questions formulated in the introduction:

Formalization of the failure detector abstraction in MapReduce Apart from the crash-stop failures, omission failures represent an important drawback in data-intensive processing frameworks. In these frameworks, omission failures are caused by slow tasks, known as stragglers, which could heavily jeopardize the workload performance. As we can deduce from the state-of-the-art, to address the omission failures in MapReduce-based systems, most of the current contributions have preferred to explore and extend the speculative execution mechanism. Other alternatives have based their contributions in doubling the computing resources for most of the tasks. Nevertheless, none of these approaches has researched a fundamental aspect related to the detection and further solving of the the omission failures (stragglers), that is, the timeout service adjustment.

In this thesis, particularly in the Chapter 4, we have studied the failure omission drawbacks in MapReduce systems, formalizing their failure detector abstraction by means of three different algorithms for defining the timeout. The first abstraction, called High relax failure detector (HR-FD), acts as a static alternative to the default timeout, but is able to estimate the completion time for the user workload. Its static adjustment is particularly efficient for small workload requests, which are a majority of them. The second abstraction, called Medium relax failure detector (MR-FD), dynamically modify the timeout, according to the progress score of each workload. Finally, taking into account that some of the user requests are strictly deadline-bounded, we have introduced the third abstraction, called Low relax failure detector (LR-FD), which is able to intersect the MapReduce dynamic timeout with an external monitoring system, in order to enforce more accurate failure detections.

Whereas HR-FD shows performance improvements for most of the user request (in particular, small workloads), MR-FD and LR-FD enhances significantly the current timeout selection, for any kind of scenario, independent of the workload type and failure injection time.

Diarchy: peer management for solving the MapReduce single points of failure Single point of failure is a common drawback in distributed computing systems. Since the earliest MapReduce-based systems, these frameworks have been strongly dependent on a single daemon, that is, the JobTracker. Hadoop YARN has changed the architecture of Hadoop (splitting the functionalities of the JobTracker between the Resource Manager and the Application Master) in order to provide scalability and remove the single point of failure presented by the JobTracker. However, the Resource Manager and the Application Master now become single points of failure in the YARN architecture.

In this thesis, particularly in the Chapter 5, we have formalized an alternative failure handling model for any MapReduce single point of failure. Regarding this, we have proposed a novel framework, called Diarchy, different from classical standby and checkpointing methodologies, that tries to enhance the MapReduce reliability, by means of the sharing of responsibilities between two master peers. In addition, we have instantiated a case study with respect to Diarchy, by addressing the application master failures within Hadoop YARN.

According to the experimental evaluations, Diarchy shows better performance efficiency when compared to Hadoop YARN, in every experimental scenario, regardless of the cluster size, workload type, or average failure rate.

AdaptCont: feedback-based resource allocation system for MapReduce Many production clouds are starting to make use of containers as resource allocation facility. Due to their characteristics, they are especially useful for running single tasks of MapReduce-based systems. However, because they are in its early stage of research, the container-based resource allocation of the state-of-the-art approaches is not particularly efficient. We have performed a fine-grain analysis of the common resource allocation of MapReduce-based systems, that is, at container level. As a result,

we notice that its resource utilization is suboptimal. This could cause a considerable performance degradation at large-scale, and even jeopardize reliability.

In this thesis, particularly in the Chapter 6, we have introduced a novel approach (AdaptCont) for resource allocation at container level, based on a closed-loop feedback-based system. In addition, we have instantiated AdaptCont for a particular case study, the application master container of Hadoop YARN.

The evaluation results indicate that AdaptCont brings substantial benefits in terms of resource utilization compared to the default resource allocation mechanism of Hadoop YARN in different setups, regardless of cluster size, scheduler policies and workload type.

7.2 Future work

Linked to the three main contributions, there are some research work lines which constitute the future directions of this thesis. In this section we describe these lines.

Instantiation of the failure detector abstraction This thesis has defined an abstract failure detector to be used in data-intensive processing frameworks, and particularly, in MapReduce-based systems. We will instantiate this abstraction for Hadoop, in order to enhance the behavior of this framework in terms of failure detection and its relation with the default timeout.

This is particularly important in the case of production clouds. Indeed, many SLAs are dependent on the timing assumptions of the data-intensive computing systems. Any decision making process made in this scenario should be based on the knowledge of the accuracy boundaries of these systems. This constitutes an important challenge in order to achieve that the data-intensive computing systems constitute a fierce competitor in some fields where relational database systems have been ruling for many decades.

Extension of the Diarchy approach to other MapReduce daemons, other infrastructures and other frameworks The Diarchy algorithm has been designed to work on every MapReduce daemon. Experimentally we have tested the Diarchy approach for the application master daemon, as use case. Nevertheless, it is expected to have wider application in the rest of the MapReduce daemons. Among others, it would be interesting to extend this approach to increase the scheduler or application manager reliability, to study how they perform under high concurrency and with unstable infrastructures, such as volunteering computing systems. In the near future, we are also planning to research and evaluate the behavior of Diarchy in different environments, including multiple cloud infrastructures, with a particular emphasis on how Diarchy performs when heterogeneous environments, such as federated clouds, are introduced as partial resources of the MapReduce framework. Finally, it would be desirable also to use Diarchy in other distributed computing frameworks with the aim of providing reliability, such as those based on master-slave model. Due to the

simplicity and performance of Diarchy, this could be implemented with minor cost, especially in those infrastructures with highly powerful networks.

Extension of the AdaptCont approach to worker tasks The AdaptCont optimization framework has been designed to be used in every MapReduce-based task. Experimentally we have evaluated AdaptCont with the application master containers, which are indeed one of the most important and complex containers within Hadoop YARN. However, the use cases where AdaptCont could introduce performance benefits are many. The main goal would be to adapt this framework for different container requests of MapReduce worker tasks. A particular challenging work would also be the deployment of AdaptCont on real distributed infrastructures.

Adaptation of AdaptCont to VMs The container-based resource allocation is performing really well. However, in many environments, VMs still represent the de facto standard of virtualization. At this moment, the container advancements have not reached the point where they could easily emulate a complete virtualized environment. Due to this, we envision the container co-living with VMs. And therefore, at long term, we expect to explore AdaptCont for VMs, in particular for allocating raw VMs to different user requests. We believe that fine-tuning a VM can be optimized, driven by requirements coming from an intersection between performance, reliability and energy-efficiency.

7.3 Publications

Part of the work detailed in this thesis has led to the following peer reviewed publications:

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *Feedback-based resource allocation in MapReduce-based systems.* Submitted for publication as Article in Scientific Programming, 2016.

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *Failure detector abstractions for MapReduce-based systems.* Submitted for publication as Article in Information Sciences, 2016.

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *Dynamic Containers for Optimizing MapReduce-based Systems.* Submitted for publication as Proceedings paper in International Conference On Computational Science (ICCS) 2016.

- Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. *Fault Tolerance in MapReduce: A Survey.* Book chapter in Resource Management for Big Data Platforms and Applications. Studies in Big Data Springer Book series. To appear in Springer, Summer 2016.

- Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. *On the Dynamic Shifting of the MapReduce Timeout.* Book chapter in Managing and Processing Big Data in Cloud Computing, Advances in Data Mining and Database Management (ADMDM) Book series. IGI Global, Pages 1-22, January 2016.

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *Diarchy: An Optimized Management Approach for MapReduce Masters.* Proceedings paper in International Conference On Computational Science (ICCS) 2015, Computational Science at the Gates of Nature. Procedia Computer Science, Volume 51, Pages 9-18, June 2015.

- Jesús Montes, Alberto Sánchez, Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *GMonE: A complete approach to cloud monitoring.* Article in Future Generation Computer Systems (FGCS), Volume 29, Issue 8, Pages 2026-2040, October 2013.

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *Enhanced failure detection mechanism in MapReduce.* Doctoral Workshop paper in International Conference on High Performance Computing and Simulation (HPCS), 2012, Pages 690-692, July 2012.

- Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. *High performance, Secure and Fault tolerant large scale storage system.* Doctoral Workshop paper in EuroSys 2011 Conference. April 2011.

Bibliography

[1] How-to: Set Up a Hadoop Cluster with Network Encryption. http://blog.cloudera.com/blog/2013/03/how-to-set-up-a-hadoop-cluster-with-network-encryption/, 2013.

[2] Introduction to Hadoop Security. http://www.cloudera.com/content/cloudera/en/home.html, 2013.

[3] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/, 2015.

[4] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2015.

[5] Apache Zookeeper. http://zookeeper.apache.org/, 2015.

[6] Facebook, Inc. https://www.facebook.com/, 2015.

[7] Hadoop Releases. http://hadoop.apache.org/releases.html, 2015.

[8] Kerberos: The Network Authentication Protocol. http://web.mit.edu/kerberos/, 2015.

[9] Microsoft, Inc. http://www.microsoft.com/, 2015.

[10] PoweredBy Hadoop. http://wiki.apache.org/hadoop/PoweredBy, 2015.

[11] The Apache Hadoop Project. http://hadoop.apache.org/, 2015.

[12] Yahoo! Inc. http://www.yahoo.com/, 2015.

[13] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-reduce Environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[14] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 287–300, New York, NY, USA, 2011. ACM.

[15] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[16] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[17] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 289–302, Berkeley, CA, USA, 2014. USENIX Association.

[18] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[19] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California Berkeley, 2009.

[21] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[22] Arun C Murthy. The Next Generation of Apache Hadoop MapReduce. `http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen/`, February 2011.

[23] Karl Johan Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, USA, 2008.

[24] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[25] Shivnath Babu. Towards Automatic Optimization of MapReduce Programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 137–142, New York, NY, USA, 2010. ACM.

[26] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993.

[27] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the Data Deluge. *Science*, 323(5919):1297–1298, 2009.

[28] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[29] Thomas C Bressoud and Michael A Kozuch. Cluster fault-tolerance: An experimental evaluation of checkpointing and MapReduce through simulation. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. Ieee, 2009.

[30] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[31] 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. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[32] Abhishek Chandra, Rohini Prinja, Sourabh Jain, and ZhiLi Zhang. Co-designing the failure analysis and monitoring of large-scale systems. *SIGMETRICS Perform. Eval. Rev.*, 36:10–15, August 2008.

[33] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[34] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[35] Qi Chen, Cheng Liu, and Zhen Xiao. Improving mapreduce performance using smart speculative execution strategy. *Computers, IEEE Transactions on*, 63(4):954–967, April 2014.

[36] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Chandra Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[37] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[38] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[39] M. Correia, P. Costa, M. Pasin, A. Bessani, F. Ramos, and P. Verissimo. On the feasibility of byzantine fault-tolerant mapreduce in clouds-of-clouds. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 448–453, Oct 2012.

[40] Pedro Costa, Marcelo Pasin, Alysson Bessani, and Miguel Correia. Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In *Proceedings of the 3rd IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 17–24, Washington, DC, USA, 2010. IEEE Computer Society.

[41] Jeff Dean. Building Software Systems at Google and Lessons Learned, 2010. Stanford EE Computer Systems Colloquium. Available at `http://www.stanford.edu/class/ee380/Abstracts/101110-slides.pdf`.

[42] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, OSDI'04. USENIX Association, 2004.

[43] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the Montage example. In *Supercomputing'08*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[44] Florin Dinu and T. S. Eugene Ng. Hadoop's Overload Tolerant Design Exacerbates Failure Detection and Recovery. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, NetDB'11, pages 1–7, New York, NY, USA, 2011. ACM.

[45] Florin Dinu and T.S. Eugene Ng. Understanding the effects and implications of compute node related failures in Hadoop. In *HPDC '12: Proceedings of the*

21st international symposium on High-Performance Parallel and Distributed Computing, pages 187–198, New York, NY, USA, 2012. ACM.

[46] Khaled Elmeleegy. Piranha: Optimizing Short Jobs in Hadoop. *Proc. VLDB Endow.*, 6(11):985–996, August 2013.

[47] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[48] Inc. Facebook. Under the Hood: Scheduling MapReduce jobs more efficiently with Corona. `http://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920`, November 2012.

[49] Gilles Fedak, Haiwu He, and Franck Cappello. BitDew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. *J. Network and Computer Applications*, 32(5):961–975, 2009.

[50] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An Optimization Framework for Map-reduce Queries. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 26–37, New York, NY, USA, 2012. ACM.

[51] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43:9:1–9:40, February 2011.

[52] Eli Gafni and Leslie Lamport. Disk paxos. In Maurice Herlihy, editor, *Distributed Computing*, volume 1914 of *Lecture Notes in Computer Science*, pages 330–344. Springer Berlin Heidelberg, 2000.

[53] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[54] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[55] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[56] Rachid Guerraoui and André Schiper. Software-Based Replication for Fault Tolerance. *Computer*, 30(4):68–74, April 1997.

[57] Rachid Guerraoui and AndrÃľ Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies âĂŤ Ada-Europe '96*, volume 1088 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 1996.

[58] Yanfei Guo, Jia Rao, Changjun Jiang, and Xiaobo Zhou. FlexSlot: Moving Hadoop into the Cloud with Flexible Slot Management. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 959–969, Piscataway, NJ, USA, 2014. IEEE Press.

[59] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[60] Tzu-Chi Huang, Kuo-Chih Chu, Wei-Tsong Lee, and Yu-Sheng Ho. Adaptive Combiner for MapReduce on Cloud Computing. *Cluster Computing*, 17(4):1231–1252, December 2014.

[61] Hortonworks Inc. *Hortonworks Data Platform: Installing HDP Manually*, 2013.

[62] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[63] 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*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[64] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.*, 4(6):385–396, March 2011.

[65] Hui Jin and Xian-He Sun. Performance Comparison Under Failures of MPI and MapReduce: An Analytical Approach. *Future Gener. Comput. Syst.*, 29(7):1808–1815, September 2013.

[66] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Benjamin P. Berman, Bruce Berriman, and Phil Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *Supercomputing'10*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[67] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *Proceedings of the 12th conference*

on Hot topics in operating systems, HotOS'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[68] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 181–192, New York, NY, USA, 2010. ACM.

[69] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[70] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[71] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 165–176, New York, NY, USA, 2014. ACM.

[72] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[73] Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. Technical report, University of Maryland, College Park, April 2010.

[74] Huan Liu. Cutting MapReduce Cost with Spot Market. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[75] Huan Liu and D. Orban. Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 464–474, 2011.

[76] Jun Liu, Nishkam Ravi, Srimat Chakradhar, and Mahmut Kandemir. Panacea: Towards Holistic Optimization of MapReduce Applications. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 33–43, New York, NY, USA, 2012. ACM.

[77] Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. On the Dynamic Shifting of the MapReduce Timeout. In Rajkumar Kannan, Raihan Ur Rasool, Hai Jin, and S.R. Balasundaram., editors, *Managing and Processing Big Data in Cloud Computing*, pages 1–22. IGI Global, Hershey, Pennsylvania (USA), January 2016.

[78] Gregory Mone. Beyond Hadoop. *Commun. ACM*, 56(1):22–24, January 2013.

[79] Jesús Montes, Alberto Sánchez, Bunjamin Memishi, María S. Pérez, and Gabriel Antoniu. GMonE: A Complete Approach to Cloud Monitoring. *Future Gener. Comput. Syst.*, 29(8):2026–2040, October 2013.

[80] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24, New York, NY, USA, 2009. ACM.

[81] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 681–684, 2010.

[82] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.

[83] Ekpe Okorafor and Mensah Kwabena Patrick. Availability of Jobtracker machine in Hadoop/MapReduce Zookeeper coordinated clusters. *Advanced Computing: An International Journal*, 3(3):19–30, May 2012.

[84] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[85] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bryan Langston. Cura: A Cost-Optimized Model for MapReduce in a Cloud. In *IPDPS*, pages 1275–1286. IEEE Computer Society, 2013.

[86] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[87] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, April 1980.

[88] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[89] James S. Plank, Matthew Allen, and Rich Wolski. The Effect of Timeout Prediction and Selection on Wide Area Collective Operations. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*

(NCA'01), NCA '01, pages 320–329, Washington, DC, USA, 2001. IEEE Computer Society.

[90] RedHat. *A guide for developers using the JBoss Enterprise SOA Platform*, 2008. Programmers Guide.

[91] Paolo Romano. Fault Tolerant and Highly Available Systems. Notes of the course: Capacity Planning, 2009.

[92] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: security and privacy for MapReduce. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[93] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):1–42, 2010.

[94] SCALUS. *Marie Curie Initial Training Network (MCITN) "SCALing by means of Ubiquitous Storage (SCALUS)"*, 2015.

[95] Bikash Sharma, Ramya Prabhakar, Seung-Hwan Lim, Mahmut T. Kandemir, and Chita R. Das. MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society.

[96] Jason Shih. Hadoop Security Overview - From security infrastructure deployment to high-level services. Hadoop & BigData Technology Conference. Available online at `hbtc2012.hadooper.cn/subject/keynotep8shihongliang.pdf`, 2012.

[97] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53:64–71, January 2010.

[98] Bing Tang, Mircea Moca, Stephane Chevalier, Haiwu He, and Gilles Fedak. Towards MapReduce for Desktop Grid Computing. In *Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '10, pages 193–200, Washington, DC, USA, 2010. IEEE Computer Society.

[99] Shanjiang Tang, Bu-Sung Lee, and Bingsheng He. DynamicMR: A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters. *Cloud Computing, IEEE Transactions on*, 2(3):333–347, July 2014.

[100] Zhuo Tang, Junqing Zhou, Kenli Li, and Ruixuan Li. A MapReduce Task Scheduling Algorithm for Deadline Constraints. *Cluster Computing*, 16(4):651–662, December 2013.

[101] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[102] Abhishek Verma, Brian Cho, Nicolas Zea, Indranil Gupta, and Roy H. Campbell. Breaking the MapReduce Stage Barrier. *Cluster Computing*, 16(1):191–206, March 2013.

[103] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. Hadoop high availability through metadata replication. In *Proceedings of the first international workshop on Cloud data management*, CloudDB '09, pages 37–44, New York, NY, USA, 2009. ACM.

[104] Daniel Warneke and Odej Kao. Nephele: Efficient parallel data processing in the cloud. In *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[105] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated)*. O'Reilly, 2012.

[106] Zhifeng Xiao and Yang Xiao. Achieving Accountable MapReduce in Cloud Computing. *Future Gener. Comput. Syst.*, 30:1–13, January 2014.

[107] Huanle Xu and Wing Cheong Lau. Speculative execution for a single job in a mapreduce-like system. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 586–593, June 2014.

[108] Huanle Xu and Wing Cheong Lau. Optimization for speculative execution in a MapReduce-like cluster. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pages 1071–1079, 2015.

[109] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[110] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[111] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[112] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[113] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[114] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[115] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[116] Hao Zhu and Chen Haopeng. Adaptive failure detection via heartbeat under Hadoop. In *Proceedings of the 2011 IEEE Asia-Pacific Services Computing Conference*, ApSCC'11, pages 231–238, New York, NY, USA, 2011. IEEE.