UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

PEDRO DE BOTELHO MARCOS

Maresia - An Approach to Deal with the Single Points of Failure of the MapReduce Model

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Porto Alegre, January 2013

CIP – CATALOGING-IN-PUBLICATION

de Botelho Marcos, Pedro

Maresia - An Approach to Deal with the Single Points of Failure of the MapReduce Model / Pedro de Botelho Marcos. – Porto Alegre: PPGC da UFRGS, 2013.

78 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2013. Advisor: Cláudio Fernando Resin Geyer.

1. Distributed systems. 2. Mapreduce. 3. Fault tolerance. 4. P2P. I. Resin Geyer, Cláudio Fernando. II. Título.

For those who somehow encouraged me to get here.

ACKNOWLEDGEMENTS

First of all, I would like to thank my parents and family for all the support during the last two years.

I will always be grateful for the opportunity that professor Claudio Geyer gave me to study here and obtain my master's degree. His friendship and support were also important to achieve my goals. Plus, I would like to thank professor Fernando Pedone for his contributions on this work.

Additionally, I would like to thank my housemates Rodrigo Oliveira and Renan Maffei, and my laboratory colleagues, in special Wagner Kolberg, by their friendship. Finally, I would like to thank RNP, the Green-Grid Project and CAPES for the scholarships that funded me.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
DELMA	Dynamically ELastic MApreduce
DFS	Distributed File System
DHT	Distributed Hash Table
DoS	Denial-of-Service
GFS	Google File System
GPGPU	General Purpose Graphics Processing Unit
GPPD	Parallel and Distributed Processing Group
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
LC	Local Checkpointing
LHC	Large Hadron Collider
MOON	MapReduce On Opportunistic eNvironments
MRSG	MapReduce Over SimGrid
MVM	Majority Voting Method
NASDAQ	National Association of Securities Dealers Automated Quotations
P2P	Peer-to-Peer
PaaS	Platform as a Service
QMC	Query Meta-data Checkpointing
RC	Remote Checkpointing
S3	Simple Storage Service
SLA	Service Level Agreement
SPOF	Single Point of Failure
SQS	Simple Queue Service
VIAF	Verification-based Integrity Assurance Framework

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

During the last years, the amount of data generated by applications grew considerably. To become relevant, however, this data should be processed. With this goal, new programming models for parallel and distributed processing were proposed. An example is the MapReduce model, which was proposed by Google. This model, nevertheless, has Single Points of Failure (SPOF), which can compromise the execution of a job. Thus, this work presents a new architecture, inspired by Chord, to avoid the SPOFs on MapReduce. The evaluation was performed through an analytical model and an experimental setup. The results show the feasibility of using the proposed architecture to execute MapReduce jobs.

Keywords: Distributed systems, mapreduce, fault tolerance, P2P.

Maresia - Uma Abordagem para Lidar com os Pontos de Falha Única do Modelo MapReduce

RESUMO

Durante os últimos anos, a quantidade de dados gerada pelas aplicações cresceu consideravelmente. No entanto, para tornarem-se relevantes estes dados precisam ser processados. Para atender este objetivo, novos modelos de programação para processamento paralelo e distribuído foram propostos. Um exemplo é o modelo MapReduce, o qual foi proposto pela Google. Este modelo, no entanto, possui pontos de falha única (SPOF), os quais podem comprometer a sua execução. Assim, este trabalho apresenta uma nova arquitetura, inspirada pelo Chord, para lidar com os SPOFs do modelo. A avaliação da proposta foi realizada através de modelagem analítica e de testes experimentais. Os resultados mostram a viabilidade de usar a arquitetura proposta para executar o MapReduce.

Palavras-chave: sistemas distribuídos, mapreduce, tolerância a falhas, P2P.

1 INTRODUCTION

1.1 Motivation

During the last years, the amount of data generated by applications grew considerably. One example of this is the Facebook, which hosts approximately 700 terabytes of data on tables and one petabyte of photos (THUSOO et al., 2009). Other examples are NASDAQ (National Association of Securities Dealers Automated Quotations), which produces one terabyte of data per day, and the Large Hadron Collider (LHC), which generates about 15 petabytes of data by year (WHITE, 2009).

To become relevant, however, this data should be processed. With this goal, distributed environments such as grids, clusters and cloud computing, are used. To properly take advantage of all the resources available, software such as Condor and Globus were developed. Another approach is to use Message Passing Interface (MPI) to distribute the processing. More recently, new programming models for parallel and distributed processing were proposed. These models intend to deal with the main aspects of distributed applications, such task scheduling, data distribution, communication and fault tolerance. As examples of these programming models Microsoft Dryad (ISARD et al., 2007) and Google's MapReduce model (DEAN; GHEMAWAT, 2004) should be highlighted.

Although both have the same goal, MapReduce and Dryad have a fundamental difference: the workflow. While MapReduce is inspired by the *map* and *reduce* primitives from functional languages, like LISP and Haskell, Dryad uses a Directed Acyclic Graph (DAG), which allows the programmer to have more freedom to develop his applications. However, even with more constraints, MapReduce is more popular than Dryad. Nowadays, MapReduce has been used on universities and big companies, such as Google, Facebook, Yahoo, Twitter, IBM and others (APACHE SOFTWARE FOUNDATION, 2011). Due to this fact and the research focus of GPPD (Parallel and Distributed Processing Group), this work will be about the MapReduce model.

MapReduce model follows a master/worker architecture and was developed to execute on big data centers of commodity machines. Under these circumstances, the probability that a fault occurs grows (DEAN; GHEMAWAT, 2004). The average number of faults on Google's data centers is between 2% and 4% per year (DEAN, 2009). In order to deal with this situation, MapReduce has fault tolerance mechanisms. These mechanisms, nevertheless, were developed only to tolerate crash faults on the workers nodes and to provide data availability. If a crash fault happens on the master node, all progress done by the job until it is lost. To guarantee data availability MapReduce uses a Distributed File System (DFS), which also uses a master/worker architecture and, generally, stores the data on the same nodes that execute the MapReduce job. A crash fault on the name node of the DFS, however, can also cause the loose of all computation done because without

it, it is impossible to locate the data to be processed. Thus, the master of the job and the name node of the DFS are considered Single Points of Failure (SPOF) of MapReduce (DEAN; GHEMAWAT, 2004).

The presence of SPOFs can be a serious problem depending on the time need to process a job. For example, if the job is running on a cloud computing environment, where the user pays for the virtual machines, a fault on a SPOF causes the loss of time and money. Also, if the same situation occurs in data centers, where the user needs to do a request to use the machines, like on Grid'5000, the user will need to reschedule his reservation to execute the job.

To avoid these problems, different solutions were proposed. The basic ones use the primary-backup replication to store the information contained on the master node into backup nodes (WANG et al., 2009). The other ones were developed to take advantage of specific services from cloud computing environments to deal with crash faults (LIU; ORBAN, 2011). On the first group of works, the backup nodes will be used only when a crash fault occurs. In this scenario, there will be computational power that will not be properly harnessed. In the second group, considering the need for specific services from cloud computing operating systems, there are constraints about the environment to use the proposal. Also, part of the existing solutions were not able to deal with both SPOFs. Thus, the need for a new method to avoid the SPOFs was detected.

1.2 Contributions

The main objective of this work is to present a new proposal, without SPOFs, to guarantee the execution of the MapReduce model on the presence of crash faults. Thus, an architecture, called Maresia, was proposed. It follows a decentralized approach and is inspired by the Chord model (STOICA et al., 2001). Among the main contributions of Maresia are:

- A new architecture, without SPOFs, to execute the MapReduce model;
- No need to add new machines on the system, for example, backup nodes, to tolerate faults;
- A platform independent solution.

To evaluate the new architecture, a prototype was developed from scratch and compared against the original MapReduce architecture. Also, an analytical model was made to compare the two approaches. The results obtained on the evaluation show the benefits of using the Maresia architecture to execute the MapReduce model.

1.3 GPPD's research on MapReduce

MapReduce has been focus of research on GPPD (Parallel and Distributed Processing Group) since 2010. The first work was MRSG (MapReduce over SimGrid) simulator, which was developed to evaluate the behavior of the MapReduce model under different scenarios. More recently, a set of algorithms was presented to adequate the core of MapReduce in order to improve its performance on heterogeneous environments. Currently, the main focuses of research are mechanisms to improve the performance of MapReduce on volatile environments, fault tolerance techniques and performance evaluation.

1.4 Cooperation

This work was developed in cooperation with Professor Fernando Pedone, from Università della Svizzera Italiana (USI), located at Lugano, Switzerland. The cooperation started on October 2011 with a visit of Professor Fernando to UFRGS.

1.5 Organization

The remainder of this work is organized as follows: Chapter 2 presents an overview about the MapReduce model and its implementations. On Chapter 3 the related work is shown and discussed. On Chapter 4 the proposed architecture and its implementation are described. Chapter 5 shows the evaluation of the proposed architecture and a comparison with the master/worker architecture. Finally, on Chapter 6 the conclusions and the future work are presented.

2 MAPREDUCE

This Chapter presents an overview about the MapReduce model and its implementations. At its end, a discussion regarding the existing fault tolerance mechanisms is done. Created by Google, MapReduce is a programming model to process large amounts of data on a parallel and distributed way. It was developed because Google has a significant number of applications which have a preprocessing step before the effective computation. On the preprocessing step, the data were split into tuples (key, value). Thus, inspired by the map and reduce primitives from functional languages, Google proposed the MapReduce model, which the main goal is allow the programmer to abstract the need to deal with the main aspects of a distributed system, such as task scheduling, data distribution, communication and fault tolerance (DEAN; GHEMAWAT, 2004). Therewith, the programmer only needs to provide the map and the reduce functions.

2.1 Overview

The MapReduce model follows a master/worker architecture. On one side there is a master node, which is responsible to schedule the tasks and to detect faults. On the other side, there is a set of workers, which role is to process the map and reduce tasks. Also, a DFS is used to store the input and the output data.

The DFS also uses a master/worker architecture, where the master node (also know as Name Node) stores a table with information about the data and guarantees its availability, for example, using replication. The worker nodes (also know as Data Nodes) are responsible to effectively store the data. In order to improve the performance (reducing the data transfer) of MapReduce, the same workers used on the MapReduce job are the Data Nodes of the DFS. The DFS used by Google's MapReduce is Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003).

On both MapReduce and DFS, the master/worker (Name node - Data node) communication occurs only to schedule tasks. The data transfer process at the beginning of map phase, or the fetching of the intermediate data between the map and the reduce phase happens directly from one node to other. Thus, the master node is not a communication bottleneck.

As mentioned before, a MapReduce application has two main steps: a map phase, where the input data are mapped into tuples, and a reduce phase, which consists on joining the tuples with same key and performing a computation over them. The job submission is done through the master node. Figure 2.1 presents the MapReduce workflow.

The first step of MapReduce's workflow is the split and distribution of the input data over the DFS. Each piece of data is called chunk, and all of them have the same size. Then, the worker nodes start to process the map function. The map function will produce

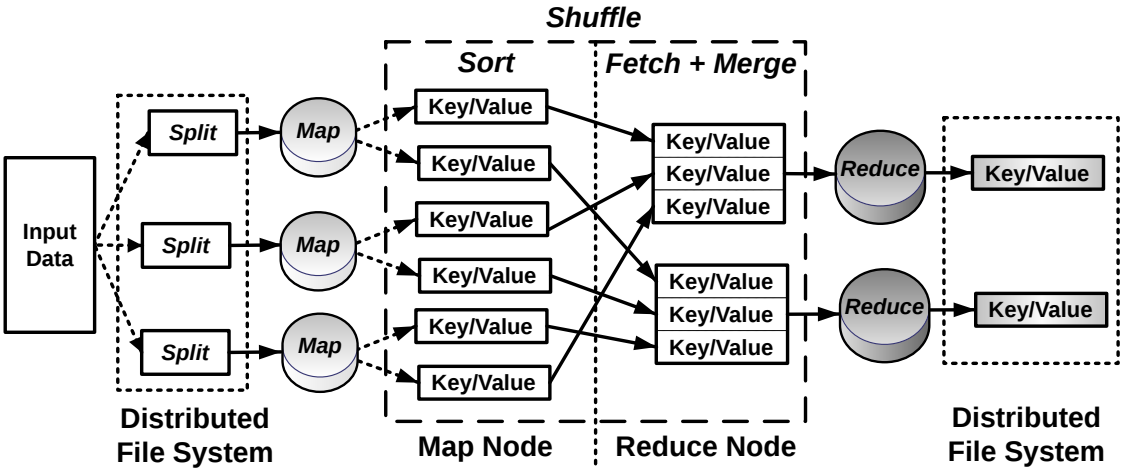


Figure 2.1: MapReduce workflow

the intermediate data, tuples (key, value). These will be consumed on the reduce phase. It is important to highlight that the intermediate data are stored locally only, not on the DFS. Algorithm 1 shows an example of map function to the word count application.

Algorithm 1 Map function to the word count application

```
map (line_number, text):
word_list[] = split (text)
for each word in word_list: do
    emit (word, 1)
end for
```

The function receives as input a pair, containing the line number and its content. Then, the content of the line is separated into words. After, for each word of the line, a tuple (key, value) is emitted, where the key is the word and the value is 1. The intermediate data will be transfered between the workers and then computed by the reduce function. This step is called shuffle and will be explained in details later. The Algorithm 2 presents the reduce function to the word count application.

Algorithm 2 Reduce function to the word count application

```
reduce (word, values[]):
word_count = 0
for each v in values: do
    word_count+ = v
end for
emit (word, word_count)
```

The reduce function will process all the values which have the same key. At the end of execution, the function will produce a tuple (key, value). On the word count example, the key is the word and the value is the number of occurrences of the word on the input data. Two considerations should be done about the reduce phase. The first one is that its computation must only start after the end of the map phase. The second is about how the reduce tasks are composed. The user defines a number of reduce tasks and each tuple

is mapped to a task through a hash function. This means, for example, that on the word count application, a reduce task may contain more than one key (word).

As mentioned before, there is a intermediate step, called shuffle, between the map and the reduce phases. The main goal of the shuffle phase is the exchange of the intermediate data among the workers. However, during this step, there are some important internal process. Before sending the data to the reducers (workers that will execute the reduce function), the mappers (workers that computed the map function) perform a sort step over the produced data. The goal of this step is to group, on each map worker, the data to be sent for each reduce worker. After, the reduce workers are able to pull the data from the mappers. Finally, after receiving the intermediate data and before starting the computation of the reduce function, the reducers execute a merge operation over the pulled data (WHITE, 2009).

Also, another step, called combine, can be executed during the shuffle phase. Its goal is to optimize the sending of messages. When a MapReduce job uses a combine function, the map workers will group all intermediate data with the same key and send it into only one message containing a tuple (key, value). On the word count application, for example, the key will be the word and the value will be the sum of all occurrences of the word on the chunks processed by the worker. This optimization is important because it saves network resources, going on the direction of one initial requirement of Google's MapReduce, which assumes that the network resources are scarce (DEAN; GHEMAWAT, 2004).

Another MapReduce optimization are the backup tasks, which aims to avoid that slow workers, also called stragglers, decrease the job performance. When the master detects, through heartbeat messages, a task which progress is below the others (can be caused, for example, by hardware problems), it launches a second instance of the task on an available worker. When one of the copies finishes, the other one is aborted. The use of backup tasks results in an improvement of the job performance. As an example, can be mentioned a sort algorithm, which execution time decreased about 44% with the use of backup tasks (DEAN; GHEMAWAT, 2008). This gain, however, is significant only because Google developed MapReduce to execute on homogeneous clusters. On this scenario, there is an assumption that all workers will need, approximately, the same time to compute the same amount of data. On heterogeneous environments, such as desktop grids, the use of backup tasks may not achieve satisfactory results (ZAHARIA et al., 2008).

A consideration about MapReduce is that due to the model simplicity, not all applications can be expressed on a single MapReduce job. Thus, a common practice on the data processing with MapReduce is chaining jobs. On this case, the output of one job is used as input to another one.

2.2 Where is it used?

Due to its programming and usage facilities, the MapReduce model has been used on universities and companies as research tool in many areas, having as target applications without dependency between the tasks. As examples can be mentioned: machine learning (CHU et al., 2006), image processing (WILEY et al., 2011) and bioinformatics (MATSUNAGA; TSUGAWA; FORTES, 2008). Big companies, such as Google, Yahoo, Facebook, Amazon and Ebay, also run MapReduce on its clusters to process large amounts of data (APACHE SOFTWARE FOUNDATION, 2011).

Yahoo has more than 40.000 nodes running MapReduce. Its biggest cluster has 4500

nodes, each one with 4 terabytes of storage. Its applications include research on advertisement systems and web search. Also, MapReduce is used on the spam filter training process of its mail service, which manages about 450 million inboxes and more than 5 billion messages per day. On Facebook, MapReduce is used for log processing, machine learning and generation of reports and statistics. One of its biggest clusters has 1100 nodes, totalizing 8800 processing cores and 12 petabytes of storage.

Another example is Google, which has more than 10 thousand applications with MapReduce. Among them are the Page Rank algorithm, large scale machine learning, report generation for the most frequent searches, computation of large scale graphs and others (DEAN; GHEMAWAT, 2010).

2.3 Implementations

Initially, MapReduce was developed by Google to run on homogeneous clusters of commodity machines. This implementation, nevertheless, was created only for internal use. Thus, from the specification of the model, the Apache Software Foundation developed an open-source implementation, called Hadoop (WHITE, 2009).

Due to the big repercussion of the its usage on homogeneous clusters, MapReduce gained different implementations, such as Phoenix, which focus are the multicore processors, MARS for GPGPUs (General Purpose Graphics Processing Unit), Cloud MapReduce and Azure MapReduce which target are the cloud computing environments, and BitDew, which was developed to run on grids. This section will present the main characteristics of these implementations, highlighting their fault tolerance mechanisms, which are the focus of this work.

2.3.1 Hadoop

Developed by Apache Software Foundation and supported by big companies, such as Facebook and Yahoo, Hadoop is the most popular implementation of the MapReduce model. Strongly inspired by the Google's MapReduce specification (DEAN; GHEMAWAT, 2004), Hadoop also follows a master/worker architecture and provides an API (Application Programming Interface) to allow the programing of the map and the reduce functions. Hadoop was developed with Java while Google's MapReduce uses C++. Apart from its use on clusters, Hadoop is offered as a service on cloud computing environments, for example Amazon Elastic Computing.

As well as Google's MapReduce, Hadoop equally uses a DFS, called HDFS (Hadoop Distributed File System), which has practically the same functionalities from Google File System (SHVACHKO et al., 2010). Hadoop general operation is the same as Google's MapReduce, which was explained on Section 2.1. Similar to Google's version, Hadoop tolerates only crash faults on the workers nodes or in the tasks. Also, Hadoop has a mechanism to skip bad records from the input data. These fault tolerance mechanisms will be explained below.

To detect crash faults, Hadoop uses heartbeat messages. These messages are periodically sent from the worker nodes to the master, and contain information about the tasks progress. The job defines a timeout (the default value is 10 minutes) and, if the master does not receive messages from a specific worker node within this interval, the worker will be considered as faulty. On this situation, the task scheduler will reschedule all the tasks that were assigned to the faulty worker, including those that were already complete, to other workers. If a crash fault occurs only on the task, it will be rescheduled to the same

worker. However, the worker will need to re-execute the entire task because Hadoop (and Google'e MapReduce) has no check-pointing mechanism to restore the execution of a task. Regarding to the bad records, if an error occurs when an input record is read, this record can be skipped to avoid a fault on the task.

2.3.2 Phoenix

The repercussion about the MapReduce usage on homogeneous clusters attracted the attention of the scientific community. Thus, new implementations of MapReduce were proposed with the goal do take advantage of other platforms to execute the model. One of these is Phoenix, which target are the multicore architectures (RANGER et al., 2007). Figure 2.2 shows the workflow of Phoenix.

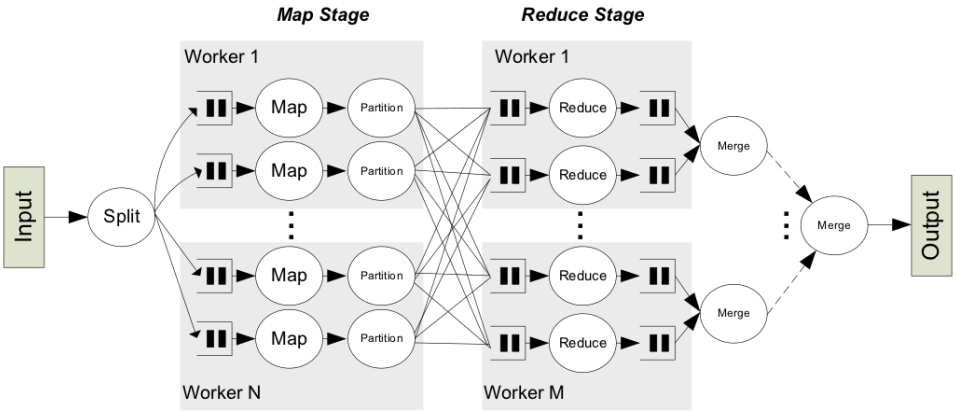


Figure 2.2: Phoenix workflow

Like Google's MapReduce and Hadoop, the input data is also divided into chunks. On Phoenix, however, the number of chunks is defined by the number of existing cores on the system. Each core executes the map function over its chunk. After this step, the intermediate data are mapped into buffers, which will be processed by the reduce function. After, the buffers are processed and the output is generated.

Different from the cluster version of MapReduce, Phoenix has no mechanism for exchange messages between the scheduler and the cores. Thus, Phoenix needs another way to detect crash faults. To do that a timeout mechanism is used. If the scheduler detects that most cores have already finished the processing of their chunks and one core does not, the chunk of the straggler (same term used to characterize the slow nodes on Google's MapReduce) core is rescheduled to a free core. When one instance (original or rescheduled) finishes, the other is terminated. This mechanism is similar to the backup tasks of the Google's MapReduce, nevertheless, instead of using the progress to reschedule a task, a timeout is used. If a crash fault occurs on the main process, all progress done is lost. Also, there is no mechanism to verify the results, however, the authors suggest the use of memory error detection techniques do deal with this situation (RANGER et al., 2007).

2.3.3 MARS

Another implementation of MapReduce is MARS (FANG et al., 2011), which was developed to execute MapReduce on GPGPUs. Its workflow is presented on Figure 2.3.

Beyond the map and reduce phase, MARS has two extra steps of processing. These are called MapCount and ReduceCount, and are executed before the map and the reduce

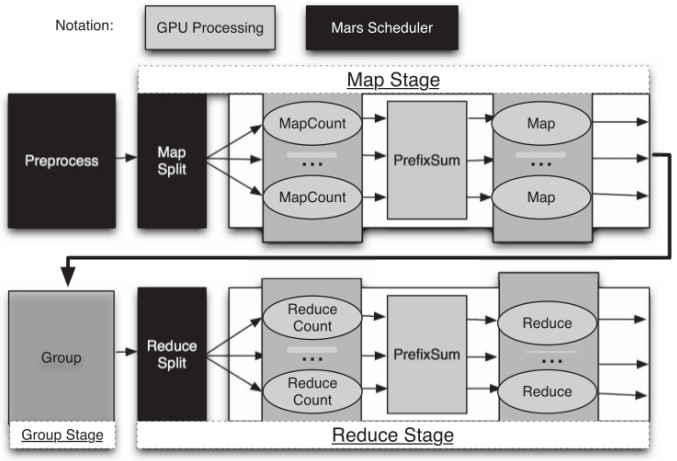


Figure 2.3: MARS workflow

phase, respectively. The goal of these steps is to estimate the size of the buffers needed for each phase. This is necessary due to the fact that the GPGPUs used to evaluate MARS do not have dynamic allocation of memory.

Two more versions of MARS were proposed. One is an hybrid approach to execute MapReduce on GPGPUs and CPUs simultaneously. On this proposal, there is a component responsible to schedule the tasks between the processors. The other version aims to execute MARS on distributed GPGPUs and CPUs. To run MARS on a distributed environment, the Hadoop Streaming tool is used. Streaming is a tool (included on Hadoop package) which allows the developers to code the map and the reduce functions on other languages (instead of Java). The only constraint is that the input data should be read from the *stdin* (Standard Input) and written on the *stdout* (Standard Output). On this approach of MARS, Hadoop is responsible to split the input data, schedule the tasks and deal with faults. Also, HDFS is used as DFS. The only version of MARS that has fault tolerance mechanisms is the distributed (combined with Hadoop), which tolerates the same types of faults of Hadoop. The other versions have none fault tolerance mechanisms.

2.3.4 BitDew

Proposed to execute on desktop grid environments, BitDew is a middleware to manage data on large scale environments (TANG et al., 2010). Over BitDew, an extension was proposed to allow the execution of MapReduce jobs on these scenarios. Figure 2.4 presents an overview of BitDew architecture.

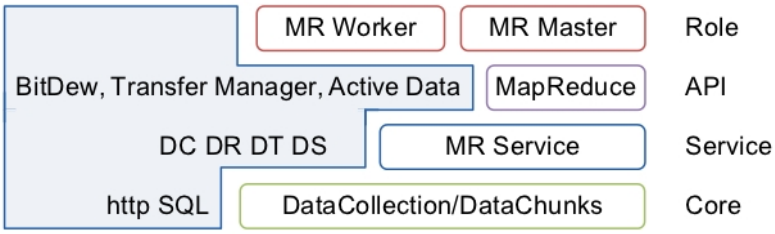


Figure 2.4: BitDew architecture

To allow the execution of MapReduce over BitDew new components were developed. These are:

- MR *Worker*: It is the role of the volatile nodes of the *desktop grid*. They are responsible for processing the map and reduce functions. Also, they store the data;
- MR *Master*: It is the role assumed by the non-volatile nodes. Examples are the master node of MapReduce and the service node, which will be explained below;
- MapReduce: It is the API to develop the map and reduce functions. It is similar to Hadoop's API;
- MR *Service*: It is the node responsible for monitoring the tasks progress and it is controlled by the master.

One of the main characteristics of desktop grid environments is the volatility. To deal with this, BitDew has fault tolerance mechanisms similar to the original MapReduce to re-execute faulty tasks. It assumes (as well as original MapReduce), however, that the MR master and the MR service nodes are non-volatile and will not crash.

2.3.5 Cloud MapReduce

With the increasing availability of cloud computing services, new implementations of the MapReduce model have emerged focused on these environments. One of them is Cloud MapReduce (LIU; ORBAN, 2011). It was developed having as base the services of the operating system of Amazon's cloud computing environment. The goal of this proposal is to take advantage of the services offered by the cloud provider to simplify the implementation and increase the MapReduce performance. A characteristic of Cloud MapReduce is its decentralized approach. Figure 2.5 presents the proposed architecture.

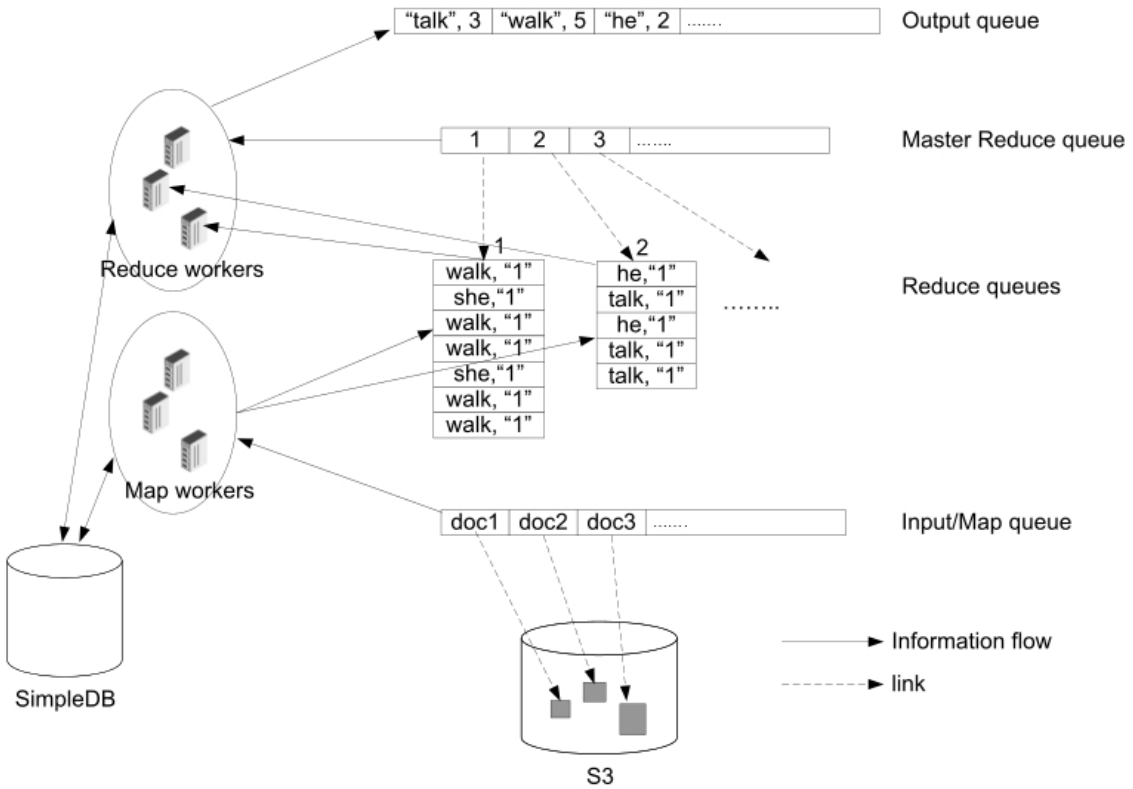


Figure 2.5: Cloud MapReduce architecture

On this architecture, Amazon S3 (Simple Storage Service) is responsible to store the input and the output data. Other services used by Cloud MapReduce are SQS (Simple Queue Service), which stores the intermediate data, and the SimpleDB, a message service, responsible for coordinate the execution of tasks.

To execute a MapReduce job the user should first store the input data on Amazon S3. Next, the first step of initialization is the creation of the queues. Also, SimpleDB should be started with messages containing the tasks descriptions. Each SimpleDB message has a pointer to a part of the input data. Then, each worker gets a message and executes the map function over the associated data. To avoid that more than one worker process the same data, the system leverages the message visibility mechanism, which "hides" a message when a worker gets it. After a timeout, the message reappears. If the worker did not finished the processing, it takes the message again. If the data were successfully processed, the message is removed from SimpleDB. The visibility mechanism is also used to tolerate crash faults on the virtual machines. If a virtual machine crashes during the execution of a task, when the timeout of the visibility mechanism expires, the message will reappear on SimpleDB. Thus, other worker can take the task to process. After finishing the mapping, the reduce phase is started. Its execution is similar to the map step. The workers will get messages from SimpleDB containing the description of the reduce tasks and will process it. When this phase is done, the output is stored.

A positive aspect of Cloud MapReduce is the possibility to insert virtual machines during the execution of the job. This can be important to replace faulty instances. Another good point is the absence of SPOFs. Nevertheless, to achieve this, Cloud MapReduce uses specific services from the Amazon's operating system, which makes the implementation platform-dependent.

2.3.6 Azure MapReduce

Another implementation focused on cloud computing environments is Azure MapReduce (GUNARATHNE et al., 2010). Like Cloud MapReduce, this approach also uses specific services from a cloud computing operating system, on this case, the Microsoft Azure.

One of the few differences between the implementations lies on the fact that Amazon offers Infrastructure as a Service (IaaS), while Microsoft offers Platform as a Service (PaaS). Thus, on Cloud MapReduce the user should allocate the virtual machines, load the software image and program its MapReduce application, while on Azure MapReduce, the user only needs to develop the MapReduce application. The other differences are on the services used. To store the data, Azure MapReduce uses Azure Storage, instead of Amazon S3. The queue service is called Azure Tables and the message service is named Message Queues. On general lines, however, the workflow is equal to the workflow of Cloud MapReduce. Figure 2.6 shows the Azure MapReduce architecture.

Like Cloud MapReduce, Azure MapReduce also allows the addition of new virtual machines during the job execution. Another similar aspect is the use of specific services from the cloud computing operating system to tolerate crash faults, which also turns Azure MapReduce a platform-dependent implementation.

2.4 Fault Tolerance

With the popularization of MapReduce model new implementations were proposed. All of them have the goal to achieve a good performance on different environments. With

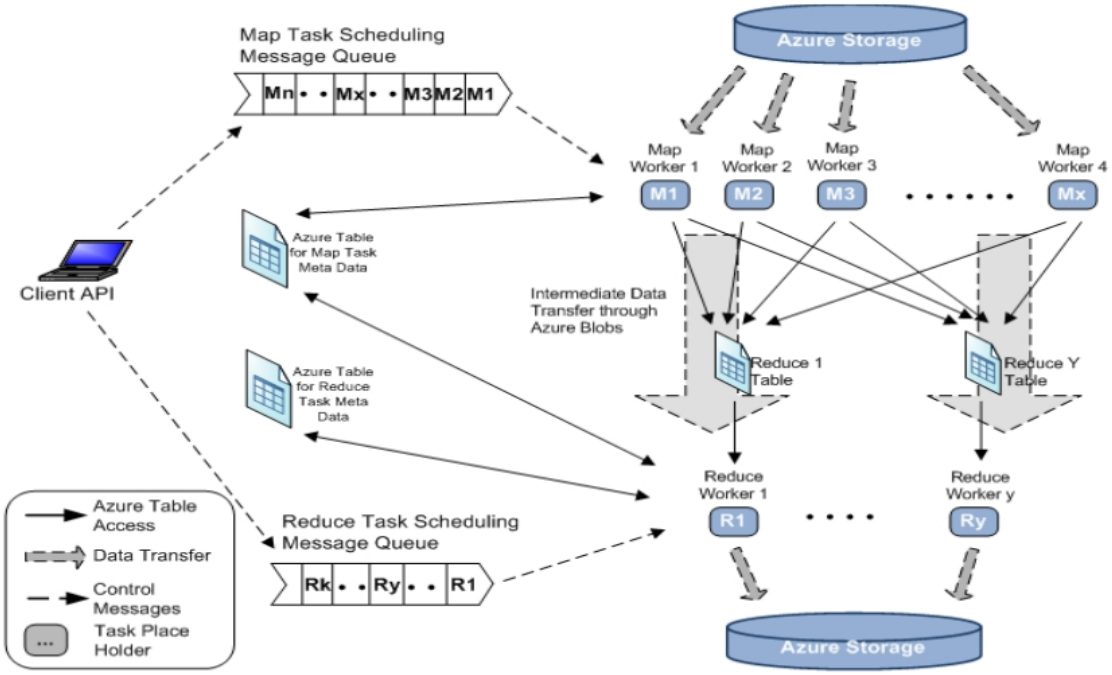


Figure 2.6: Azure MapReduce architecture

the exception of MARS, all proposals have fault tolerance mechanisms. However, none of them deals with all types of faults. Also, sometimes the adopted solution is not the most appropriate (i.e. the need to re-execute all tasks of a faulty worker), or it is dependent of a specific service from a given platform, which makes impossible to use the solution on other environment.

Hadoop, as well as Google's MapReduce, deals only with task/worker crash faults. This makes the system vulnerable to byzantine faults and to faults on one of the SPOFs. The same considerations can be done about BitDew. Another point is that, even with the fault tolerance mechanisms existing on these solutions, in case of a worker fault, will be necessary to re-execute tasks that were already complete.

Phoenix also offers mechanisms to detect tasks faults and suggests the use of memory error detection algorithms do avoid incorrect answers. However, a fault on its scheduler compromises the job execution. As mentioned before, MARS has no fault tolerance mechanisms on its own implementation. The distributed version of MARS, which uses Hadoop Streaming, tolerate faults. This is possible only due to the Hadoop's fault tolerance schemes. Finally, the cloud computing implementations do not have SPOFs. They are, however, sensitive to byzantine faults. Also, the use of specific services from cloud computing operating systems to tolerate crash faults make these implementations platform-dependents.

The next Chapter presents the state of art about fault tolerance on MapReduce. All areas of fault tolerance will be discussed, with a major attention to the works related to the SPOFs, which are the focus of this dissertation.

3 RELATED WORK

This Chapter presents the most relevant research about fault tolerance mechanisms on MapReduce model. The proposals are divided into three groups. First, those that are related to crash faults on the worker nodes. The byzantine faults are the focus of the second group. Finally, the last set of works presents the existing approaches to deal with the SPOFs. This group will receive more attention because the SPOFs are the focus of this dissertation.

3.1 Workers Faults

3.1.1 Intermediate data persistence

One of the main focuses of study on MapReduce is the persistence of the intermediate data. If the intermediate data are persisted, when a worker fails, the system does not need to re-execute the tasks that were already finished by the faulty worker. Persisting data, however, has a cost, and to be useful the cost to store and retrieve the data should be lesser than the cost to re-execute the tasks.

As mentioned on Section 2.1, the output of the map function is stored only on the local disk of the worker. Thus, the intuitive solution to persist the intermediate data is to store it on the DFS used with MapReduce. Nevertheless, studies shown that HDFS, the DFS used by Hadoop, has a high cost to store the intermediate data (KO et al., 2010).

A proposal to deal with this problem is the use of BlobSeer as MapReduce DFS (NICOLAE et al., 2011). The authors justify its use, instead of HDFS, because BlobSeer has a lower cost to persist the intermediate data. Also, the evaluation shows that BlobSeer achieves higher rates of read and write operations than the ones obtained by HDFS. Three experiments were done to evaluate the performance of BlobSeer on a MapReduce job: persisting the intermediate data on the new DFS, on HDFS, and only storing it on local disk (original method). The application used is the grep, which has as characteristic the small amount of intermediate data. The results show that BlobSeer has a lower cost than HDFS and the local persistence. The performance, however, is not significantly different. For a 9GB input, the time difference is less than ten seconds. Also, if the input size increases, the time difference decreases. Another point is that the authors only evaluated the time to store the intermediate data, while the recovery process is mentioned as a future work.

Another proposal with the goal to persist and to recover the intermediate data is the Intermediate Storage System (ISS) (KO et al., 2010). The authors suggest the use of asynchronous replication to reduce the persistence cost and guarantee data availability. The increase on the performance is followed by the reduction on data consistence (caused

by the asynchronous replication). On MapReduce, nevertheless, consistence is not a big problem because the data are only written once. The proposed system was evaluated on a chaining job scenario, where the output of a reduce function is used as input of the map function of other job. According to the authors, if a crash fault occurs, this scenario can suffer from the cascade re-execution process. An example is presented on Figure 3.1.

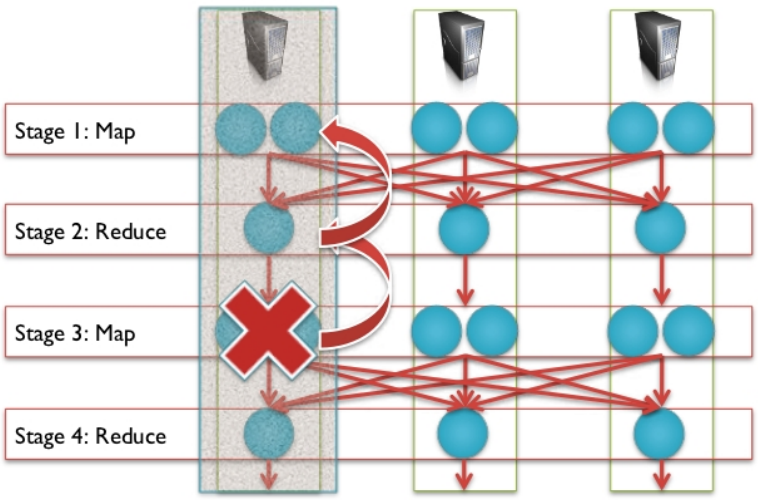


Figure 3.1: Cascade faults example

On this case, if there is no intermediate data persistence, a crash fault on a worker may cause the re-execution of all tasks that were executed by it. To solve this problem, ISS uses three techniques. The first one is the asynchronous replication, which increases the performance. To reduce the network usage (which is a scarce resource (DEAN; GHEMAWAT, 2008)), the replication is done, mainly, on the rack level. The last technique is focused on what intermediate data need to be replicated. The main problem are the intermediate data that will be consumed locally by a worker. Thus, if a crash fault occurs, this data will be lost. To solve this problem, the authors propose the replication of the local intermediate data on another worker. The experiments showed that ISS can almost hide the presence of a faulty worker. Which means that the time to execute Hadoop with ISS, on a scenario with none or one crash fault is similar. If compared to the execution time of Hadoop using HDFS, the time of ISS version is, approximately, 10% faster. Another consideration is that the network is a bottleneck to the replication process.

A third solution to provide the availability of the intermediate data is a set of algorithms called RAFT (Recovery Algorithms for Fast-Tracking) (QUIANE-RUIZ et al., 2011). Three mechanisms were proposed with the goal to persist the intermediate data, with low cost and fast recovery. Figure 3.2 presents the workflow of MapReduce with RAFT algorithms.

The first algorithm is the Local Checkpointing (LC), which consists only on performing a local checkpoint of the intermediate data during the task processing. Thus, if a crash fault occurs on a task, the worker only needs to execute the task from the point it made the last checkpoint. However, if the worker crashes, the checkpoint is lost. To deal with this problem, a Remote Checkpoint (RC) algorithm was proposed. It is similar to LC, the difference is that the intermediate data is persisted on a remote place, instead of the local disk. This algorithm, nevertheless, has a problem because the large amount of intermediate data that can be generated on the map phase. This can, significantly, increase

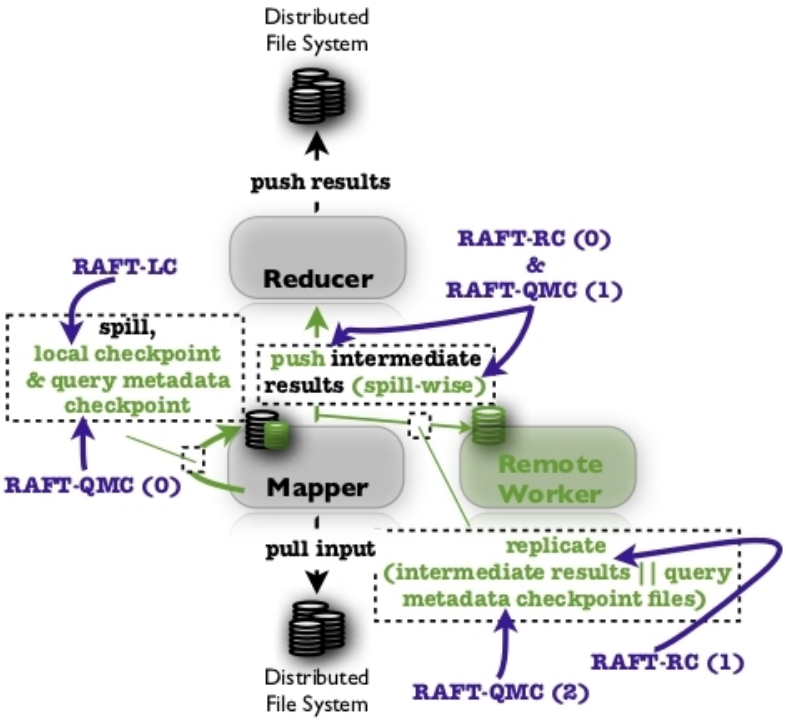


Figure 3.2: Workflow of MapReduce with RAFT algorithms

the execution time of the job. The last algorithm, called Query Meta-data Checkpointing (QMC), aims to improve the RC algorithm. To reduce the network resources needed to store the intermediate data on a remote worker, the QMC replicates only a file containing meta-data about the intermediate data. The meta-data is used to recovery the intermediate data in case of a crash fault on a worker. The evaluation shows an improvement of the performance of Hadoop in scenarios with crash faults. This gain is between 7% and 25%, depending of the moment and the number of crash faults. The overhead to persist the intermediate data is about of 5%.

3.1.2 Techniques to avoid the loss of computational power

Another problem caused by a crash fault on a worker is the loss of computational power. Suppose that the user allocated 10 machines to process 10GB of data. If a worker crashes, the amount of data (and probably the job time) each worker needs to process will increase. The original version of MapReduce does not allow the insertion of new machines during the job execution. This can be a serious problem, specially if the job is running on a cloud computing environment, where the user pays to use the resources. It can also be a problem for the service provider, which sometimes needs to guarantee a SLA (Service Level Agreement).

A proposal to admit the insertion of workers during the job execution is DELMA (Dynamically ELastic MApreduce) (FADIKA; GOVINDARAJU, 2011). Developed over Hadoop, its main goal is to add new machines without the necessity to stop the job and restart it from the beginning. The overhead caused by this process, however, should be smaller than the improvement on the execution time. The workflow of DELMA can be described as follows: the user does a request to add workers on the system. The master node pauses the job execution and verifies the status of each worker. The new nodes are

registered on the master node and the data that are not already processed are redistributed among the workers. After this step, the job returns its normal execution. The evaluation of DELMA shows that the time needed to add workers is independent of the number of new nodes. The reconfiguration time, however, is directly related to the input data size. This is caused by the fact that the data redistribution is responsible for the major time of the reconfiguration process. Other experiments shown that the performance increasing is related to the moment that the new nodes are included on the system. Insertions that happened at the beginning of the job resulted on a higher performance improvement.

Another approach to this problem is MapReduce RSA (KADIRVEL; FORTES, 2011), which presents a policy to allocate new workers. The goal of this policy is to guarantee that the SLAs will be respected, even on the presence of faulty workers. To define the amount of workers that will be inserted, the policy considerate: the number of crash faults, the moment they happened, the job progress and the SLA definitions. Experiments on a 16-node cluster indicates that a crash fault on one worker can cause an increase of 16% on the execution time. Also, it shows that the addition of 2 workers after a crash fault, can do the time increasing be almost null. On the case of 2 crash faults, the time increases about 84%, while the insertion of 2 workers reduces this cost to 9.1%, and, with 4 new workers, there is an improvement of 5.2% on the performance. Other considerations about the experiments show that the addition of workers is better when the original number of nodes is small. This happens because when a crash fault occurs on a scenario with a large number of nodes, the loss of computational power is small.

A third solution suggests the allocation of extras nodes at the beginning of the job (ZHENG, 2010). Different from the previous works, this proposal does not allow the insertion of new nodes during the execution time. The author suggests the allocation of a group of extra nodes at the beginning of the job. The number of extra workers can be estimated, for example, by the average percentage of crash faults during a job. The experimental scenario was performed over the Amazon Cloud Computing service. The results show that, even on a situation with a significant number of crash faults, the average improvement on the execution time was about only 3%.

3.1.3 Other improvements

Besides the two main groups of works about fault tolerance on MapReduce workers, there are some other aspects related to workers faults that are focus of research. One of them is the necessity to avoid that some data be computed more than once when a crash fault occurs on a real time MapReduce implementation. Thus, a new version of MapReduce, called StreamMapReduce, was proposed (MARTIN et al., 2011). The motivation to do this new version is that some MapReduce applications can require real time answers. One example is the monitoring of SLAs. If the detection that a SLA is not being respected is done quickly, the system manager can act to fix the problem. On StreamMapReduce, the reduce function can start without waiting for the end of the map phase. Thus, partial results can be produced and analyzed. This modification, however, creates the problem of some data be processed twice if a crash fault occurs. To solve this problem, the reducers store information about the map tasks progress and the received data. Thus, the duplicated data can be discarded. The experiments showed that on scenarios with less than 10 crash faults, the performance is almost the same obtained on scenarios without crash faults. Nevertheless, if the number of crash faults increases, the performance can be reduced about 75%.

Other improvements were made to execute MapReduce on volatile environments, such

as opportunistic environment. With this goal, MOON (MapReduce On Opportunistic eNvironments) was proposed (LIN et al., 2010). The main difference of this approach is the combination of dedicated nodes with volatile nodes. The dedicated nodes are used to store the input data, provide data availability and execute tasks, while the volatile nodes only execute tasks. The evaluation showed that the strategies used by MOON can achieve a 50% performance improvement on environments with a higher degree of volatility. However, on environments with up to 10% of volatility, the performance of MOON is similar to the obtained by Hadoop.

Finally, an adaptive model was proposed to define the timeout used to detect faults on Hadoop (ZHU; CHEN, 2011). According to the authors, the default timeout (10 minutes) is reasonable only for applications with a execution time higher than 10 minutes. For short applications, however, if the timeout was not properly adjusted, a fault can incur on a 1000% overhead on the execution time. To deal with this situation, a model to properly set the timeout of the heartbeat messages was presented. The results showed a reduction of 800% on the execution time if the timeout is set using the adaptive model.

3.2 Byzantine Faults

Byzantine faults are a problem, mainly, on environments where the workers do not belong to the same domain. Examples of that are the desktop grids and the opportunistic environments. The original MapReduce assumes that all workers are trustworthy. On these environments, however, it is impossible to assure that all workers are not malicious. Therefore, to deal with this problem, solutions were proposed to execute MapReduce on these scenarios.

The main aspect of the byzantine fault tolerance mechanisms is the verification of the results produced by the workers. One proposal to do this is SecureMR (WEI et al., 2009), which aims to provide: workers that are able to verify the input data integrity; reducers with the capacity to check the authenticity and correctness of the map output; mechanisms to check the job output. Thus, SecureMR tolerates: data tampering, incorrect answers, tampered messages, DoS (Denial of Service) and collusive behavior.

The basic workflow of SecureMR can be described as follows: the first step is the scheduling of the map tasks, which are replicated on different workers. After the processing, the worker sends a hash of the result to the master. If all workers that processed the same data sent the same value, the task is considered correct. If not, an alert is emitted. According to the authors, the goal of the work is only to detect suspect behavior, thus, no action is taken to fix the alert. After the verification, the reduce tasks are scheduled. To avoid that the mappers sent the incorrect data to the reducers, the master also sends the hash of each data to the reducers nodes. Thus, the reducer can verify the correctness of the received data. If some inconsistence is detected, an alert is raised. Else, the data is processed and the final result is produced. The authors do not presented a way to verify the results of the reduce phase, but they mentioned the possibility to use a mechanism similar to the one used to check the map output.

Another approach to tolerate byzantine faults on Hadoop is VIAF (Verification-Based Integrity Assurance Framework) (WANG; WEI, 2011). To guarantee the correctness of results, VIAF uses quiz methods and task replication. Also, to help on the verification process, a small number of trusted nodes, called *verifiers*, is used. These nodes are responsible for the detection of collusive or non-collusive attacks during the map phase. The reducers are not verified because, according to the authors, the number of reduce

tasks is small, which means that the trusted nodes can execute these tasks.

Each task is scheduled on two workers and a hash of its results is sent to the master node. If the hash is different, the task will be rescheduled to another pair of workers. Else, the answers will be stored on a cache. After, the verifier will check the results. To reduce the verification costs, this step will be done on a probabilistic way. If the hash computed by the verifier is different, a collusive attack was detected. On this case, the workers will be inserted on a blacklist and all tasks scheduled to these nodes will be re-executed. If the result of the verifier is the same, each worker will receive a credit. After achieving a certain number of credits, the results produced by these workers will be sent to the reducers. VIAF causes a 22% processing overhead and a 10% increase on the execution time.

A modification on BitDew's MapReduce to allow the result verification was presented (MOCA; SILAGHI; FEDAK, 2011). The authors presented two versions of their mechanism. The first one only replicates the map tasks, while the second one replicates both, the map and the reduce tasks. Differently from SecureMR and VIAF, this proposal uses a distributed result verification. The map tasks are verified by the reducers while the results of the reduce phase can be verified by any worker. The verification of the result is done using the Majority Voting Method (MVM). In this method, there are n replicas of each task, if $n/2 + 1$ return the same result, it will be considered as correct. The problem of MVM is that it is not secure against collusive attacks. To reduce the possibility of a collusive attack, the number of replicas of each task should be increased. Nevertheless, a larger number of replicas incurs on a higher overhead.

3.3 Single Points of Failure

As mentioned before, MapReduce has two SPOFs, the master node of the job and the Name Node of the DFS. If a fault occurs on one of these points, all processing already done is lost. To avoid this situation, several techniques were proposed. These include the use of backup nodes, modifications on the DFS architecture, decentralized architectures and others.

A common approach is the use of backup nodes to store the information contained on the SPOFs. Two implementation of this model were proposed (WANG et al., 2009). The first, called *active/standby*, uses only one backup node, while the second, called *primary/slaves*, has n replicas of the master node. The main difference between them is that the second version uses an election algorithm to choose the new master. It also tolerates more crash faults. Both proposals have three phases. On the initialization step the backup nodes register themselves into the master node and the synchronization between the master and the backup nodes is done. The second stage consists on the replication of meta-information about the actions performed by the master node. The meta-information is used to reduce the usage of the network resources. The last phase is the system recovery. The fault detection is done through the absence of messages between the backup nodes and the master. After, an election algorithm is used and the recovery process is started, which consists only on the new master changing its IP address to the one used by the previous master. With the active/standby model, which uses only one backup node, after the fault detection, the system needed about seven seconds to recovery. There is no tests using the *primary/slaves model*, which, probably, has a higher recovery time because the need to execute an election algorithm.

Another solution using backup nodes has focus on dynamic distributed environments

(MAROZZO; TALIA; TRUNFIO, 2011). The main characteristic of these environments is the churn of nodes[1]. Thus, a crash fault on the master node has a higher probability to occur. To deal with this situation, the architecture presented on Figure 3.3 was proposed.

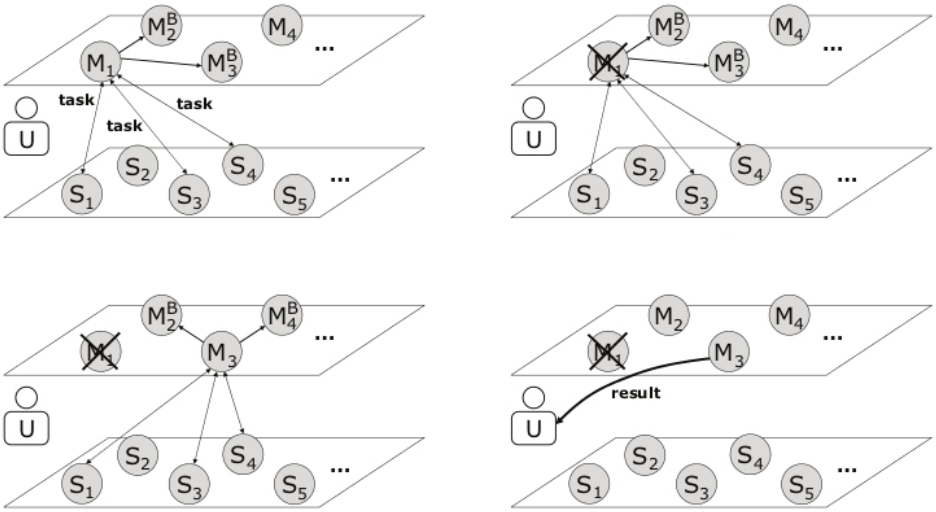


Figure 3.3: Proposed architecture

The first step is the job submission. At this moment, the master node will select other nodes to act as its backup nodes and will allocate a set of workers to process the tasks. If a crash fault occurs on the master node, it will be detected through the absence of heartbeat messages. On this case, another node will start to act as a master node. Also, one more backup node will be allocated. At the end of the execution, the master node returns the result for the user. The proposed architecture was evaluated using a simulator. The results shown that, independently of the number of crash faults, all jobs will finish their executions. The difference between this proposal and the previous one is that this allows the insertion of a new backup node if a crash fault occurs. None of them, however, treats crash faults on the Name Node of the DFS.

An approach to tolerate faults on the Name Node of the DFS is the use of UpRight library (CLEMENT et al., 2009). It is a general-purpose library to create fault tolerant applications. Its main characteristic is to allow fault tolerance without a significant modification on the source-code of the target application. To evaluate the library, the authors modified the source-code of HDFS, the DFS used by Hadoop. To make it fault tolerant, the authors needed to insert about of 1800 lines of code. According to they, the number of necessary lines is not significant if compared to the effort to turn HDFS fault tolerant without the library. The modified version of HDFS achieved 80% of the performance of the original HDFS on the write operation. For the read operation, the performance is almost the same.

Another alternative to avoid the SPOF on the Name Node is the substitution of the DFS. Brisk was proposed with this idea (DATASTAX, 2011). Its goal is to integrate Hadoop with Cassandra File System, a P2P DFS. Figure 3.4 presents an overview of Brisk.

On this architecture, all nodes are able to retrieve data without the necessity of a Name Node. This is possible because the information about the location of each file is spread

[1]When the nodes join and leave the system frequently

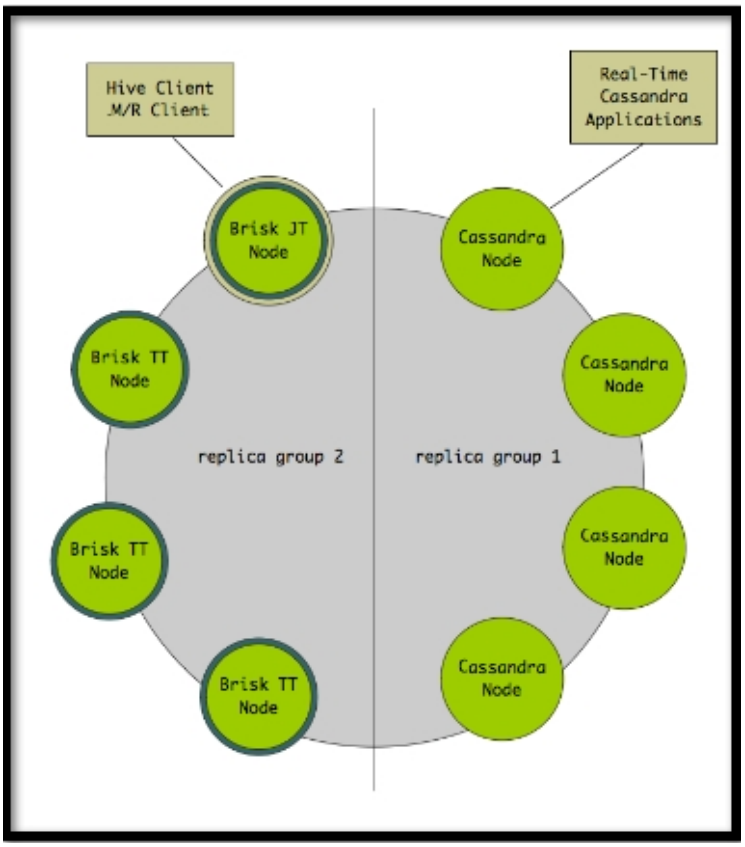


Figure 3.4: Brisk architecture

over the nodes. Like the solution using UpRight library, Brisk only deals with Name Node SPOF.

3.4 Discussion

Due to the increasing demand for availability, the use of fault tolerance techniques is becoming increasingly necessary. This characteristic is followed on MapReduce. Developed to process large amounts of data on a parallel and distributed way, MapReduce can suffer from performance decrease on the presence of crash faults. Thus, fault tolerance techniques are being used with three distinct focuses: crash faults on the workers nodes, byzantine faults and mechanisms to avoid the SPOFs.

Different solutions were presented to deal with crash faults on workers and byzantine faults. On the first group there are proposals to modify the DFS to store the intermediate data and to use new algorithms to avoid the need to re-execute a task. Also, there is a set of approaches focused on adding workers, during the execution time, to replace the loss of computational power caused by a crash fault. On the second group there are solutions which goal is to verify the produced results. To do this, some of them use a centralized verification, while the others apply a distributed verification.

The third group presented on this Chapter has proposals to deal with the SPOFs. As mentioned before, there are two SPOFs on MapReduce, the master node of the job and the NameNode of the DFS. To avoid the SPOF on the master node, two approaches were presented, both using backup nodes. Nevertheless, the computational power of these nodes will only be effectively used if a crash fault occurs on the master node. To deal

Table 3.1: Approaches to avoid the SPOFs of MapReduce

Criteria/Solution	**Backup Nodes**	**Brisk**	**UpRight**	**Cloud MR**	**Azure MR**
All SPOFs	No	No	No	Yes	Yes
Plat-indep.	Yes	Yes	Yes	No	No
Same # nodes	No	Yes	Yes	Yes	Yes

with the SPOF on the DFS, there is a proposal using a new DFS, which has as main characteristic its P2P architecture. Another solution is the modification of HDFS using the UpRight library. None of the previous works, nevertheless, presents a solution without any SPOFs..

Beyond the ideas above, there are the decentralized architectures presented on Section 2.3. Cloud MapReduce and Azure MapReduce were designed to take advantage of services offered on cloud computing operating systems to improve the MapReduce performance. These solutions also eliminate both SPOFs. To do this, however, they take advantage of specific services from the cloud computing operating systems. Thus, these proposals will only work on a specific environment. Table 3.1 presents a comparison between the existing approaches to avoid the SPOFs of MapReduce.

This work aims to present a decentralized architecture to avoid the SPOFs on MapReduce. The proposed solution deals with both SPOFs, it is platform-independent and all computational power available is harnessed. The next Chapter presents this architecture, which is called Maresia, in details.

4 PROPOSED MODEL

This Chapter presents the Maresia architecture, which main goal is to avoid the SPOFs on MapReduce. Also, the proposed architecture should properly utilize all the computational power available and be platform independent. In order to deal with the SPOFs, Maresia uses an approach inspired by Chord (STOICA et al., 2001). A P2P network is used to decentralize the information contained on the SPOFs and distribute it among the peers. Also, Chord was chosen because it provides a structured network with relative low cost to find an information. To a better understanding of the proposal, an overview about Chord is presented below. After, the Maresia architecture and its implementation are described.

4.1 Chord

Proposed to facilitate the search of information on nodes that are connected through a P2P network, Chord uses a virtual unidirectional ring to organize its peers. Each peer has an unique identifier, which is used to define its position on the ring. Figure 4.1 shows an example of a Chord ring.

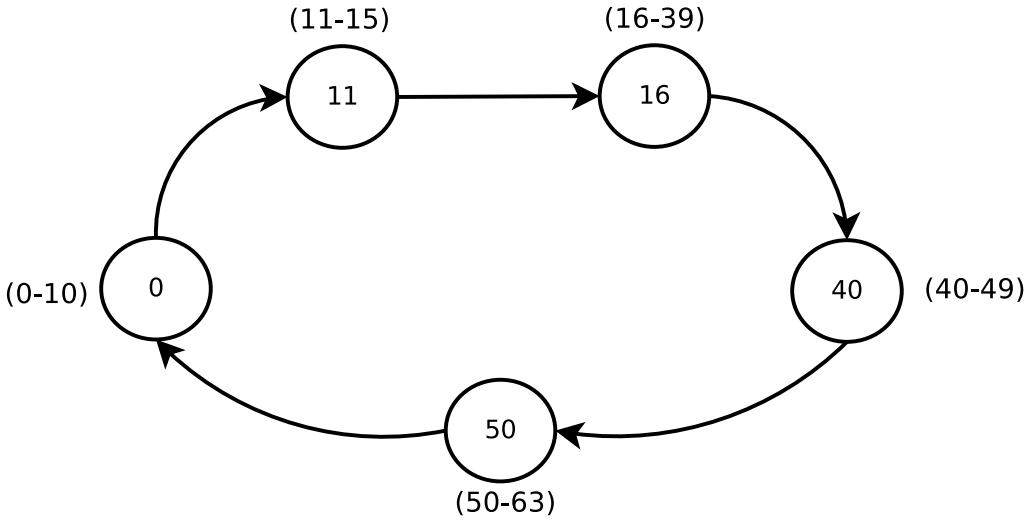


Figure 4.1: A ring used by Chord

The definition of the peer ID is done using a hash function. This function is applied by a ring node over some peer information (generally the IP address) and then the identifier is generated. On the Figure 4.1, each peer also has a set of keys (for example, peer 11 - (11-15)), which defines the interval of keys that the peer is responsible to store. The set

starts on the peer ID and goes until its successor ID minus one.

When a peer wants to store some data, it applies the hash function (know by all peers) over the information, generating a key. With this key, it can discover the node which is responsible to store the data. If a peer wants to retrieve a data, it applies the hash function over some description of the desired data. After, it requests the data to the responsible peer. This technique is called Distributed Hash Table (DHT).

To store or retrieve an information, the interested peer asks its successor to know if it is responsible for the key. If the neighbor is the owner of the key, it accepts the request and sends the data. Else, the request is forwarded to the next peer. With this technique, on the worst case, it is necessary to ask all peers on the ring, which means a complexity of $O(n)$ messages, where n is the number of peers on the ring. To optimize the lookup and the storage process, Chord uses a *finger table*. It contains information about up to m peers, where m is the number of bits used to generate the identifiers. Thus, with the *finger table*, there is no need to perform a request to the neighbor. The peer can lookup the information stored on its *finger table* and ask the peer that is closest to the desired key. Therewith, the average number of messages becomes $O(logn)$, where n is the number of peers on the ring.

On the bootstrap process, Chord assumes that the new peers are able to discover, using an external mechanism, the IP address of a peer on the ring. After connecting with this peer, the new peer asks what will be its ID. After receiving the information, the new peer connects to the correct peer and establishes the necessary connections. Also, the set of keys and its associated information are redistributed and the *finger table* is updated. When a peer leaves the ring, intentionally or not, it is necessary to redistribute the keys, update the *finger table* and reorganize the connections.

4.2 Maresia

The main goal of Maresia is to avoid the SPOFs. As mentioned before, a fault on a SPOF will cause the loss of all processing that was already done. Also, the need for a new proposal is due to the fact that the existing solutions have some limitations, such as the necessity to add new nodes, the dependence of specific services from a platform or the presence of one SPOF. These limitations are not present on Maresia. Below the Maresia assumptions are presented.

- No workers will be inserted during the execution time;
- There is no byzantine behavior;
- There is an external mechanism to perform the bootstrap process.

The first two assumptions also exist on the original version of MapReduce, while the third is from Chord. On this proposal, Maresia is designed to execute on a trusted domain (i.e. cluster), without the insertion of new nodes during the execution time. These assumptions are planned to be removed on a future version.

The original architecture of MapReduce deals with two types of faults. The ones occurred on the workers nodes and the ones on the tasks. However, faults on the master node of the job and on the Name Node of the DFS are not tolerated. Maresia aims to deal with the faults tolerated on the original version of MapReduce and to avoid the SPOFs. The next sections will describe the Maresia architecture and its workflow.

4.2.1 Architecture

As mentioned before, Maresia uses a P2P architecture inspired by Chord. An overview of the proposed architecture is presented on Figure 4.2.

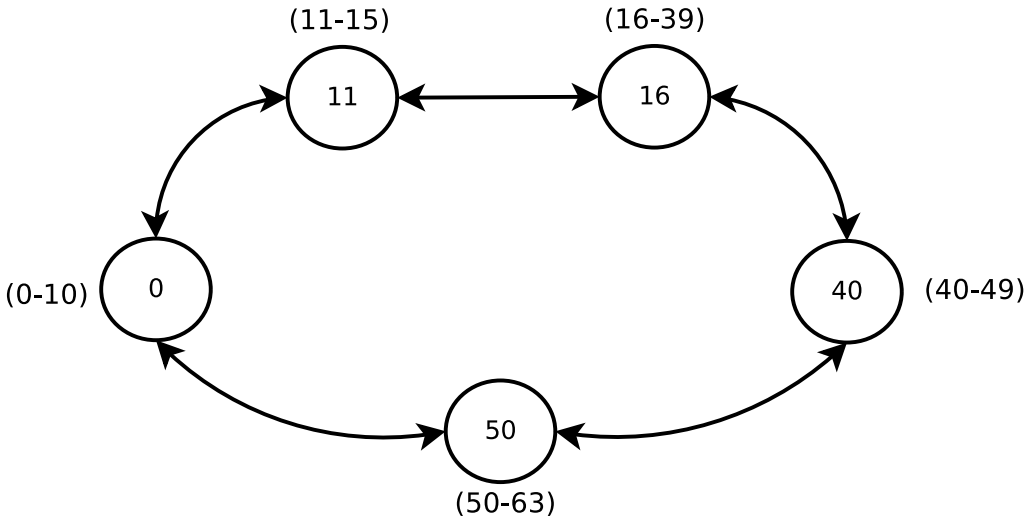


Figure 4.2: Overview of the Maresia architecture

Similar to Chord, the peers are organized on a virtual ring. However, while Chord uses a unidirectional ring, on Maresia the ring is bidirectional. This modification is necessary for the fault tolerance mechanisms, which are discussed on Section A.4.1.3. On the ring, each peer has a unique identifier and a numeric interval, which corresponds to the set of keys that are under its responsibility. The interval of each peer starts on its identifier and goes until the identifier of its successor minus one. Like Chord, a DHT (Distributed Hash Table) function is applied over an information to map it to a peer.

4.2.2 Workflow

On this Section, initially, the workflow of Maresia on scenarios without crash faults is presented. After, the mechanisms used to tolerate crash faults are detailed. The first step is the creation of the virtual ring. Like Chord, Maresia assumes the existence of an external mechanism to perform the bootstrap process. A fault on the bootstrap mechanism can only prevent that new peers join the ring, nevertheless, will be possible to execute the job using the peers that are already connected on the ring. After the bootstrap process, each peer will have a unique identifier and a set of keys, which it will be responsible to store.

The last step of the initialization process is the distribution of the input data among the peers. The "owner" of the input data, which can be part of the ring or not, will split the data into chunks of same size (this is the same process used on the original MapReduce), and distribute them over the peers. To define which peer is the responsible for each chunk, the DHT is used. After the data distribution, the virtual ring will be like Figure 4.3.

For example, after applying the hash function over the *chunk one* (C1), the generated key has a value between 0 and 10, thus, *chunk one* is stored on *peer 0*. Afterward the distribution of all chunks, the job is ready to start. As Maresia has no master node, there is no dynamic scheduling. Thus, each peer computes the map function over its local chunks. *Peer 11*, for example, will process the chunks C4 and C5, while *peer 40* will compute the chunks C9 and C10. After processing a chunk, the peer applies the hash function over the

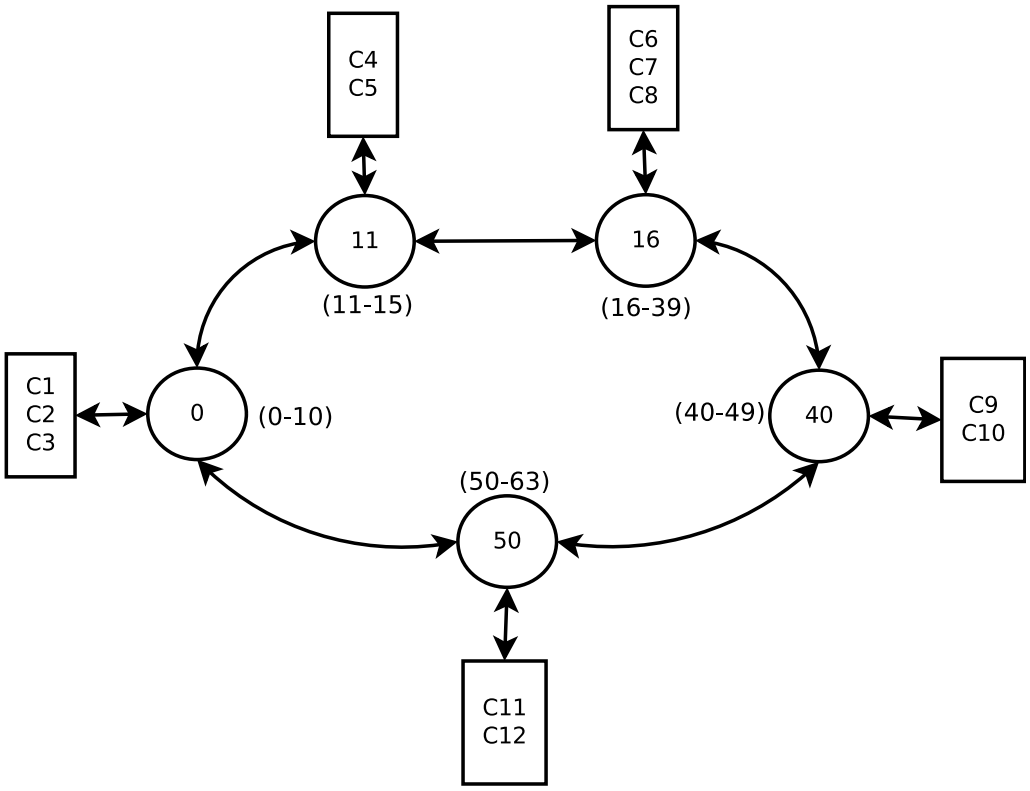


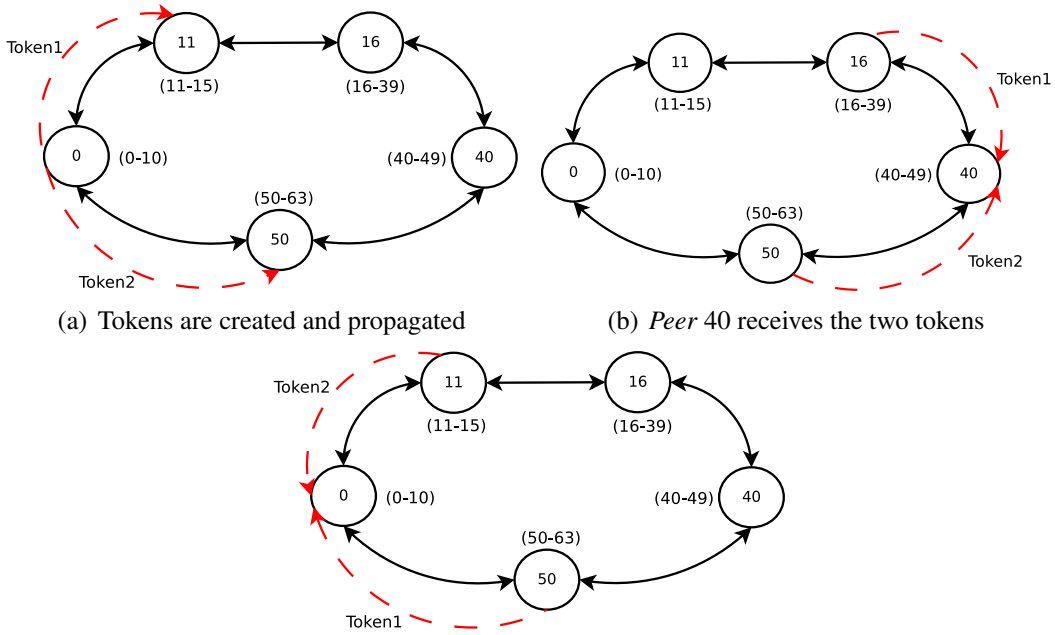
Figure 4.3: Maresia ring after the chunk distribution

key of each tuple (key, value) generated to discover what peer is responsible to execute the reduce task over this data. Then, the intermediate data is sent. Differently from the original version of MapReduce, where the intermediate data is pulled by the reducers from the mappers, on Maresia the pairs are pushed by the mappers to the reducers. As there is no master node to inform which peers have data to the reduce task, it is better reverse the data transferring process.

One constraint of the MapReduce model is that the processing of the reduce phase must only start after the conclusion of the map phase and the send of the intermediate data. On the original version of MapReduce, the master node informs the reducers that the next phase can start. Without the presence of a master node, Maresia needs a way to guarantee that the reduce phase will only start after the end of the map phase. One possible approach is the use of message broadcasting, however, in this case it is necessary that each peer knows all the others to inform about the end of processing of the local chunks. Even if all peers know the complete ring, the complexity of the broadcast all-to-all is $O(n^2)$, where n is the number of peers on the ring.

To solve this problem, Maresia uses a token mechanism. Thereby, when a peer (only one), for example, the one that is the responsible for the key 0, finishes the map phase, it creates two tokens and sends one to its right neighbor and the other to its left neighbor. If a peer receives a token from its left neighbor and it already finished the map phase, it passes the token to its right neighbor. The same thing is done if it receives a token from the right neighbor, except that the destination of the token is the left neighbor. If a peer receives the two tokens (not necessarily at the same time) and it has finished the map phase, it can start the reduce processing. When the creator of the tokens receives the two tokens back, all peers will be processing the reduce phase. This approach has a message complexity of $O(2 * n)$, where n is the number of peers. The factor 2 is because each peer

has to forward two tokens. Also, with this solution, there is no need to a peer know all the others on the ring. Figure 4.4 presents the token propagation phase.



(a) Tokens are created and propagated

(b) *Peer* 40 receives the two tokens

(c) Tokens return to the creator

Figure 4.4: Token propagation

If a homogeneous environment is used to execute the job, and the chunks are distributed using a fair hash function, which means that all peers will have almost the same amount of data to process, on an ideal environment, all peers tend to finish the map phase on the same time. On this scenario, the token phase starts and finishes on a short interval of time. Nevertheless, this is not a real situation. Even on homogeneous environments, the peers execution times are different. This is caused, for example, by hardware problems. Thus, on a real scenario, a slower machine can late the start of the reduce phase (this fact also happens on the original MapReduce (DEAN; GHEMAWAT, 2008)). To minimize this problem, Maresia allows the use of work stealing techniques. If a peer finishes the map phase, it can send a message to its neighbors asking if they have a chunk that is not already computed. Thereby, the peer can steals the chunk and process it. The same considerations are valid to heterogeneous environments, where the probability of slow nodes is higher. This mechanism is like the MapReduce's backup tasks, explained on Section 2.1.

Similar to the map phase, on the reduce phase, each peer computes the data received from the mappers. The definition of where each reduce task will execute is done through the set of keys of each peer. For example, if a peer is responsible for the keys 0, 1 and 2, it will compute three reduce tasks. At the end of the job, the output is stored and the job creator is informed that its results are ready.

4.2.3 Dealing with faults

The previous Section presented the basic workflow of Maresia on scenarios without crash faults, showing the feasibility of executing the MapReduce model on a decentralized architecture. Fault tolerance mechanisms, however, are necessary to guarantee the correct processing of the job if some unexpected fact, such as a machine power down or an error,

occurs. Thus, this Section presents the fault tolerance techniques used on Maresia. The goal of these mechanisms is to tolerate the same faults tolerated by the original version of Hadoop and to avoid the SPOFs.

The first aspect that should be observed is the data availability. Differently from the original version, Maresia does not uses a DFS. Thus, the replication must be done by Maresia. To guarantee the data availability, all peers replicate their input chunks onto their neighbors. Thereby, Maresia will have three copies of each chunk, tolerating up to two crash faults over the same data. To improve the fault tolerance, when a crash fault occurs, new replicas of the data owned by the faulty peer can be generated. Figure 4.5 shows the peers after the chunk replication (the distribution without replication is presented on Figure 4.3).

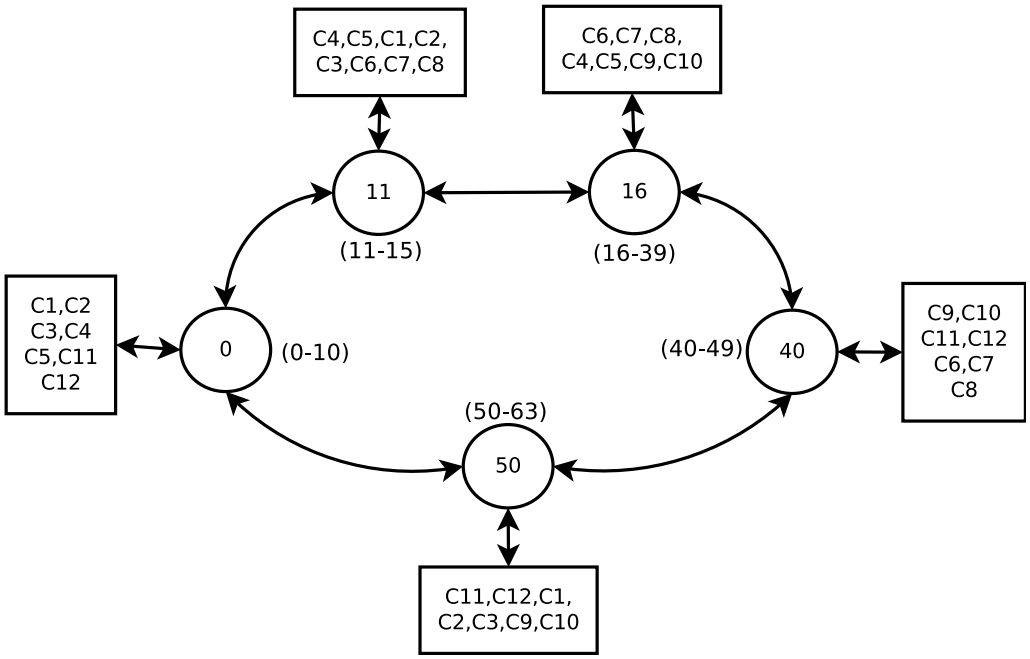


Figure 4.5: Maresia ring after the chunk replication

Another fundamental point is the detection of a fault and the re-execution of the pending tasks. To detect faults, Maresia uses a heartbeat mechanism, similar to the one used by the other implementations of MapReduce, such as Hadoop. On Hadoop, each worker, periodically, sends heartbeat messages to the master node. If during a certain time, the master does not receive messages from a worker, it is considered as faulty and its tasks will be rescheduled to another worker. Maresia has no master node to monitor all peers, thus, the fault detection is performed locally. Thereupon, during the execution of job, each peer exchanges heartbeat messages with its neighbors. These messages contains the following information:

- The running task;
- The task execution progress;
- Token information;
- A space to be used for piggybacking (For example, *finger table* updates).

Every time a peer finishes a chunk processing, it sends this information to its neighbors on the next heartbeat message. Thus, all peers will know about the chunks already processed by their neighbors. Therewith, if a crash fault happens on a peer, its neighbors (which also have replicas of its input chunks) will know what tasks are not yet processed. On this situation, after reorganizing their connections, the neighbors decide who will execute the remaining chunks (if all chunks were processed, the execution follows normally). The same procedure can be applied on the reduce phase to execute the tasks of a faulty peer. A peer will be considered as faulty by its neighbors, if after a certain period of time, they do not receive any heartbeat message from the peer.

At the end of the map phase, two tokens are created. These tokens should be fault tolerant. This first problem to deal is the token creation on the case of a crash fault on the peer responsible for this action. Maresia defines that the peer which owns the key 0 creates the tokens. If this peer faults before the token creation, the tokens will be created by the new owner of key 0. To detect the token faults, different algorithms can be used (ARANTES; SOPENA, 2010).

The last fault tolerance mechanism is used to provide the persistence of the intermediate data. The original version of MapReduce does not persist the intermediate pairs (there are some works to add this feature on MapReduce, for more information see Section 3.1). On the original version, if a reducer faults, its tasks will be rescheduled to another worker and the master will coordinate the process to resend the intermediate data to the new reducer. Also, if there is the necessity to re-execute some map task to produce some data to the reducer, the master reschedules the map task. As Maresia has no master, one way to solve this problem is to persist and replicate the intermediate data. Thus, the peer that executes the reduce task will receive the intermediate data, while its neighbors will store the replicas. Replicating the intermediate data incurs on a significant cost but a small recovery time. Another possibility is to use the QMC algorithm (discussed on Section 3.1) to reduce the replication cost. This algorithm, nevertheless, has a higher recovery cost. Both approaches will be compared on Chapter 5. The next Section will describe how Maresia was implemented.

4.2.4 Implementation

To allow the evaluation of the proposed architecture on a real environment, an implementation was developed. It was done using C language and was built from scratch. Initially, was considered the possibility to implement the proposal having Hadoop as base. Nevertheless, the complexity of Hadoop's source-code, specially its size (more than 300.000 lines), and the number of modifications needed, changed the plans. The implementation language was chosen due to its programming facility and to achieve a better performance. However, to provide a fair comparison between the two architectures, an analytical model is provided. The evaluation of both implementations is presented on Chapter 5.

4.2.4.1 Bootstrap and ring configuration

A peer has two options at the beginning: create a new environment (ring) to execute a job or join an existing ring. To define the main parameters of the ring and the job, a configuration file is used. To create a ring, the configuration file must have:

- The path to the input file;
- The size of the chunks;

- The number of reduce tasks;
- The port used to accept connections;
- The number of peers needed to execute the job.

The job only starts after the number of peers achieves the defined value. To join the ring, each peer should specify a configuration file with the IP address of the creator of the ring and its port. To simplify the implementation, the new peer will be the successor of the last peer that joined the ring. When the last peer joins, the ring is complete.

The division of the keys is made using the number of reduce tasks. If the number of reduce tasks is lesser (or equal) than the number of expected peers, each peer is responsible for one key, which corresponds to a reduce task. If the job has more reduce tasks than peers, each peer is responsible for $round(number\_of\_reduces/number\_of\_peers)$.

4.2.4.2 Data distribution

Before start the computation, the peers must have the input data. There are two possibilities for this. On the first, the data can already be on the peers, on this case, when the ring is created, a parameter is set to inform that the data were already distributed. Also, each peer should receive a list of the local data to be processed. On the second, the creator of the ring splits the data and distribute them. To guarantee that each peer have, approximately, the same amount of data, a *round-robin* algorithm is used, instead of the DHT.

4.2.4.3 Finger table

As Chord, Maresia also has a *finger table*. However, the one used on Maresia is a little different. It contains information about all peers of the ring. Once a peer contacts the ring creator (when it joins the ring), its information (IP address, peer ID, ...) are stored on the *finger table*. After all peers join the ring, the *finger table* is sent to all peers before the job start. In terms of size, the information of each peer can be stored on 24 bytes. Thus, if the ring has, for example, 500 peers, only 11.7 KB are needed to store all the information.

4.2.4.4 MapReduce

The processing of the MapReduce job starts after the propagation of the *finger table*. There is no need to distribute the application code because, on the current version, it is directly embedded on the Maresia code. On the map phase, each chunk is computed and the intermediate data generated is stored on lists, one for each reduce task. After finishing the chunk processing, the intermediate data is sent to the reducers. Before sending the data, the combine function (for more details, see Section 2.1), which is also embedded on the Maresia code, is executed to reduce the amount of messages need. The information about the destination peer is found on the *finger table*. If all chunks were processed, the peer waits for the tokens (which are created by the peer which owns the key 0) to start the reduce phase.

Parallel to the processing of the map phase, there is a communication thread, which is responsible to receive the intermediate data generated by the other peers and store them. The intermediate data received are temporarily stored on memory, and after moved to disk (a similar process, called *spill*, occurs on Hadoop).

When a token is received, the peer verifies if all chunks were processed. If the map phase is complete, the token is propagated instantaneously, if not, the peer waits until the

end of the map phase and then, sends the token. When the second token is received, the reduce phase starts.

On the reduce phase, the intermediate data stored on disk is read and processed. After, the output is generated, persisted on disk, replicated on the neighbors of each peer, and the creator of the job is informed that the results are available.

4.2.4.5 Fault tolerance

To provide data availability Maresia uses replication. The replication of the input chunks is done at the data distribution step. After a chunk is sent to a peer, a copy of this data is sent to its neighbors. The same process is used with the intermediate data.

Maresia has a heartbeat mechanism, which is used to inform the neighbors about the tasks progress and to detect faults. These messages are sent every three seconds (the same period used by Hadoop). To detect crash faults, Maresia monitors the heartbeat messages and, if after a certain time, no heartbeat messages are received from a peer, it is considered as faulty. Implement a correct fault detection mechanism is a hard task, which main problem is to guarantee that a crash fault detected really happened (SCHIPER; TOUEG, 2008). As the implementation of a fault detection mechanism is out of the scope of this work, the current implementation simulates the faults.

To simulate crash faults, when Maresia is initialized, a configuration file is used to specify the characteristics of the fault that will be emulated. This file has the ID of the faulty peer, the phase (map or reduce) of the fault and its moment (after how many chunks or how many reduce tasks). When the moment arrives, the faulty peer ends its execution and the neighbors perform the necessary actions. The connections between the peers are reorganized, the set of keys are redistributed and the pending tasks are rescheduled to the neighbor with the greater ID.

5 EVALUATION

This Chapter presents the experiments and the discussion about the performance of Maresia. The evaluation aims to compare the performance of Maresia against Hadoop (on scenarios without crash faults) and to evaluate the behavior of Maresia on the presence of crash faults. To do this, an analytical model was developed and tests on a cluster environment were executed.

5.1 Methodology

The evaluation process is divided on two parts. The first shows an analytical model to compare the execution of MapReduce on the proposed architecture and on the original one. The analytical model aims to present a comparison of the execution time and the number of messages (supposing that all have the same size) needed to run a job. The second part is focused on the experimental evaluation, which aims to compare Maresia and Hadoop, on scenarios without crash faults, using as metric the execution time. Also, to evaluate the fault tolerance mechanisms of Maresia, tests were performed on scenarios with crash faults, and the recovery time was analyzed.

The experimental tests were executed on the GPPD's cluster SLD, which is composed by 20 machines, each one with a Pentium 4 processor (2.8 GHz), with 2GB of memory and 1TB of storage. All experiments were repeated 30 times and its means, standard deviations and confidence intervals (with 95% of confidence) were calculated.

5.2 Analytical Modeling

This Section shows the analytical model of the execution time and the number of messages needed to compute a job using the Maresia architecture and the original MapReduce architecture. Equation 5.1 presents the execution time of a MapReduce job, on a scenario without crash faults, when the master/worker architecture is used.

$$t_{orig} = t_{map} + t_{reduce} \tag{5.1}$$

Where t_{map} is the time to execute the map phase and t_{reduce} is the time needed to complete the reduce tasks. The time for transferring the intermediate data (shuffle phase) is considered as part of t_{map} because it starts during the map phase and finishes before the starting of the reduce step. The execution time for Maresia, on the same scenario, is given by the Equation 5.2.

$$t_{maresia} = t_{map} + t_{token} + t_{reduce} \tag{5.2}$$

The main difference between the approaches is the time to pass the token, which is defined by t_{token}, on the decentralized architecture. This time, in general, is short. If when a peer receives the token, it already finished the map phase, the only overhead caused on the system by this step is the time necessary to forward a message. The time to process the map and the reduce phase, on both architectures, tends to be the same. For example, the cost to compute the same function over some data (assuming it is locally stored on the worker) is independent of architecture. Due to the distinct methods used to send the intermediate data, the t_{map} can cause a variation among the execution times. Thus, on scenarios without crash faults (assuming that there is no replication of the intermediate data on Maresia), the time needed to execute a job will be almost equal.

The second aspect evaluated is the number of messages used during the job processing. Table 5.1 presents a list of the variables and their meanings. The Equation 5.3 shows the estimated number of messages, considering the absence of crash faults, needed on the master/worker architecture.

Table 5.1: List of Variables

Variable	Meaning
H	Heartbeats
C	Chunks
R	Reduces
N	Workers/Peers
K	Keys
F	Faulty tasks
A	Maximum number of consecutive crash faults

$$msgs = H + C + R + R * (N * K) \tag{5.3}$$

During the job execution there are the sending of H heartbeat messages from the workers to the master. This amount is dependent of the frequency defined to exchange the messages. Also, there is one message for each chunk that will be processed. These messages are used to schedule the map tasks. The same is valid to the reduce tasks, represented by R. The last part of the equation is the number of messages needed to transfer the intermediate data. On the master/worker architecture, the owner of a reduce task asks all workers about the intermediate data (K) associated with its task. For the decentralized approach, the number of messages is given by the Equation 5.4.

$$msgs = H + 2 * N + R * (N * K) \tag{5.4}$$

On the proposed architecture, the heartbeat messages are also present. Similar to the master/worker architecture, its amount is dependent of the defined periodicity. On Maresia, nevertheless, there is no scheduling messages, thus the variables C and R are not present on the equation. This is because Maresia uses a static scheduling, where each

peer process its local data. On the other side, there are the insertion of $2 * N$ messages, which correspond to the passage of the tokens. The number of messages needed to send the intermediate data is equal. The difference, however, is that the mappers will send the data to the reducers, instead of the reducers ask the mappers about the data.

Comparing both equations its possible to assume that the number of heartbeat messages is equal. The same can be applied to the messages needed to pull/push the intermediate data. The difference between the equations lies on the scheduling and the token messages. While the master/worker architecture has no token messages ($2 * N$), Maresia has no scheduling messages ($C + R$). As, in general, the number of chunks is considerably greater than the number of workers/peers, it is possible to affirm that $2 * N < C + R$. Thus, the number of messages needed by the decentralized architecture, on most cases, is smaller than the amount necessary on the original architecture.

Considering the presence of crash faults on the MapReduce job, they can be from two types: crash faults on the workers or a crash fault on the master node. If the crash fault happens on a worker on the original proposal during the map phase, its message cost can be described by the Equation 5.5. If the crash fault occurs on the reduce phase, the message cost is the one presented on the Equation 5.6.

$$msgs = H + C + R + R * (N * K) + F \tag{5.5}$$

$$msgs = H + C + R + R * (N * K) + F + F * (N * K) \tag{5.6}$$

If a crash fault occurs on the map phase, it is only necessary to reschedule the chunks of the faulty node to another worker. Thus, for each faulty task, one message, represented by F, is needed. On the other side, if the crash fault occurs on the reduce phase, will be necessary one message to reschedule each task (F), and to fetch all the keys associated with the faulty task, which incurs on $F * (N * K)$ messages.

On Maresia architecture, this cost can be a little bit higher depending of the algorithm used to persist the intermediate data. Independently of the algorithm, the first step is to define which neighbor will execute the tasks of the faulty peer, which can consume up to 3 messages ($3 * F$). One approach is the traditional replication algorithm, where each tuple will be replicated twice. Equation 5.7 shows the number of messages, needed by Maresia on scenarios with crash faults, where A means the maximum number of consecutive crash faults over a data. Replicating the intermediate data makes the cost independent of the phase in which the crash fault occurs.

$$msgs = H + 2 * N + (R * (N * K)) * (A + 1) + 3 * F \tag{5.7}$$

On these scenarios, all intermediate data will be sent A times. As, in general, there is a significant amount of intermediate data, the increase on the number of messages can be significant. A solution to reduce the message cost on Maresia is to improve the method used to provide the availability of intermediate data. This can be done, for example, using the QMC (Query Meta-data Checkpointing) algorithm (QUIANE-RUIZ et al., 2011). Its goal is to replicate only meta-information about the localization of the intermediate data. Thus, when a crash fault occurs, this information is used on the recovery process. The

number of messages needed using the QMC algorithm, on scenarios with crash faults on the map phase, is presented on the Equation 5.8.

$$msgs = H + 2 * N + R * (N * K) + 3 * F + A * R \tag{5.8}$$

On this case, the increase on the number of messages is caused by the replication of the meta-information files, which is represented by $(A) * R$. Comparing with the master/worker architecture, this cost tends to be similar to the one achieved by it on scenarios with crash faults on the map phase. Supposing a scenario with an input of 4GB, a chunk size of 64MB (which corresponds to 64 chunks), 10 reduce tasks, 16 nodes and 1 map crash fault, the cost in messages of the decentralized approach using the QMC will have 20 messages less than the original architecture. For crash faults on the reduce phase, the cost can be represented by Equation 5.9.

$$msgs = H + 2 * N + R * (N * K) + 3 * F + A * R + F * (N * K) \tag{5.9}$$

Similarly to recover from a reduce crash fault on the original architecture, will be necessary to fetch the intermediate data associated with the faulty task ($F * (N * K)$). Considering the same configuration described above, the difference between the number of messages will be also 20. Thus, it is possible to observe that QMC is an interesting approach to provide the availability of the intermediate data. Nevertheless, its recovery time will be higher than the one obtained using the effective replication of the intermediate data. Finally, it is important to highlight that if a crash fault occurs on the master node, the original version will stop the job execution. On the other side, Maresia will finish the execution normally.

5.3 Experimental Evaluation

With the goal to evaluate Maresia on a real environment, experiments were performed. The tests are presented following and are divided into three groups. The first one assess the performance of Maresia against Hadoop on scenarios without crash faults. The second group evaluates the recovery time of Maresia on scenarios with crash faults. Finally, the third group has focus on measure the overhead caused by the fault tolerance mechanisms, in particular the one caused by the replication of the intermediate data. As mentioned before, the experiments were performed using the SLD cluster of GPPD. The evaluation scenarios have 8 and 16 machines (all machines are equal), while the workloads have 1GB, 2GB and 4GB. The application used to assess the behavior of MapReduce on both architectures is the Word Count. To persist the intermediate data the traditional replication algorithm was used.

5.3.1 Performance

The first aspect evaluated is the performance of Maresia against Hadoop (version 1.0.3) in scenarios without crash faults. The experiments were executed on scenarios with 8 and 16 machines and with different numbers of reduce tasks. The only parameter that was used as factor was the number of reduce tasks because it is one of the most influence over the execution time (HERODOTOU; BABU, 2011). The levels used to define the number of reduce tasks were $number\_of\_workers/2$ and $number\_of\_workers$. The values of the other parameters are presented on Table 5.2.

Table 5.2: MapReduce Parameters

Parameter	Value
Chunk Size	64 MB
Map Slots	1
Reduce Slots	1
Number of replicas	3

The parameters *Map Slots* and *Reduce Slots* define the number of simultaneous map and reduce tasks on each worker. As the machines used to execute the tests have only one core, the value 1 was set. The number of replicas and the chunk size were defined as 3 and 64MB, respectively, because they are the default values of MapReduce (DEAN; GHEMAWAT, 2004). Figure 5.1 presents the mean time for the execution of a job using 8 machines and 4 reduce tasks.

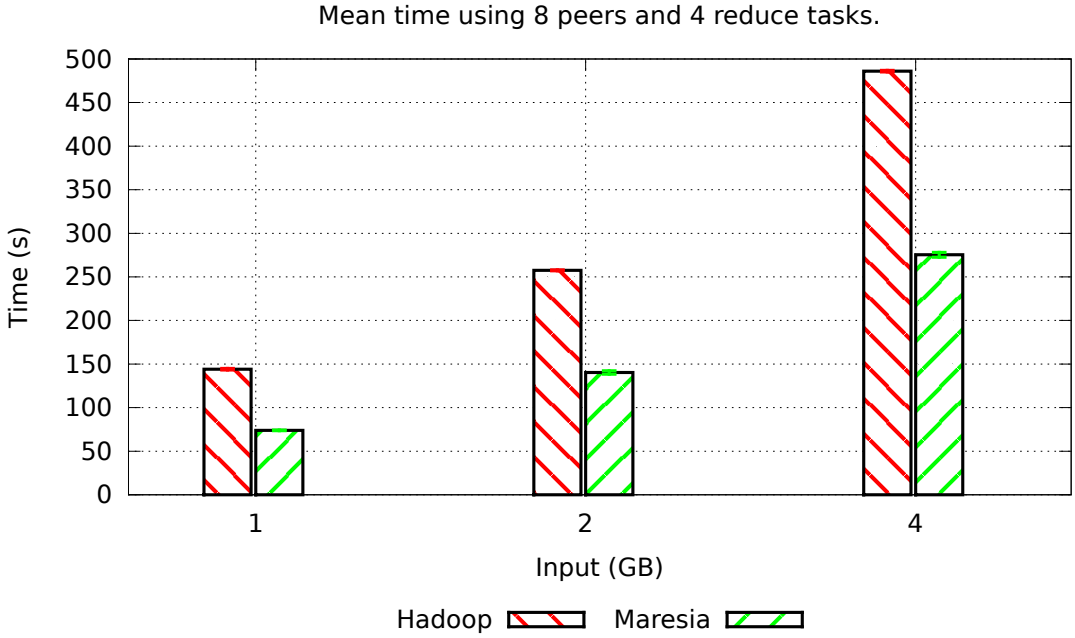


Figure 5.1: Mean time using 8 machines and 4 reduce tasks

For all evaluated workloads, Maresia has a smaller execution time than Hadoop. Using the 1GB workload, Maresia presented a speedup about of 97%, while for the 2GB and the 4GB workloads, the improvement is about of 83% and 77%, respectively. The main cause of this difference, probably, is caused by the programming language. While Maresia was developed using C, Hadoop uses Java. Due to the fact the Java uses a virtual machine to run its applications, this difference was already expected. However, it is also possible to see that, in both implementations, the relation between the workload size and the execution time is almost the same. On Hadoop, if the workload is duplicated, for example, from 2GB to 4GB, there is an overhead of 91%, while on Maresia, the increasing on execution time is about of 95%. The experiments with 8 machines and 8 reduces are presented on Figure 5.2.

Similar to the previous tests, the execution times of Maresia are smaller than the ones obtained using Hadoop. The speedups obtained with Maresia for the workloads of 1GB,

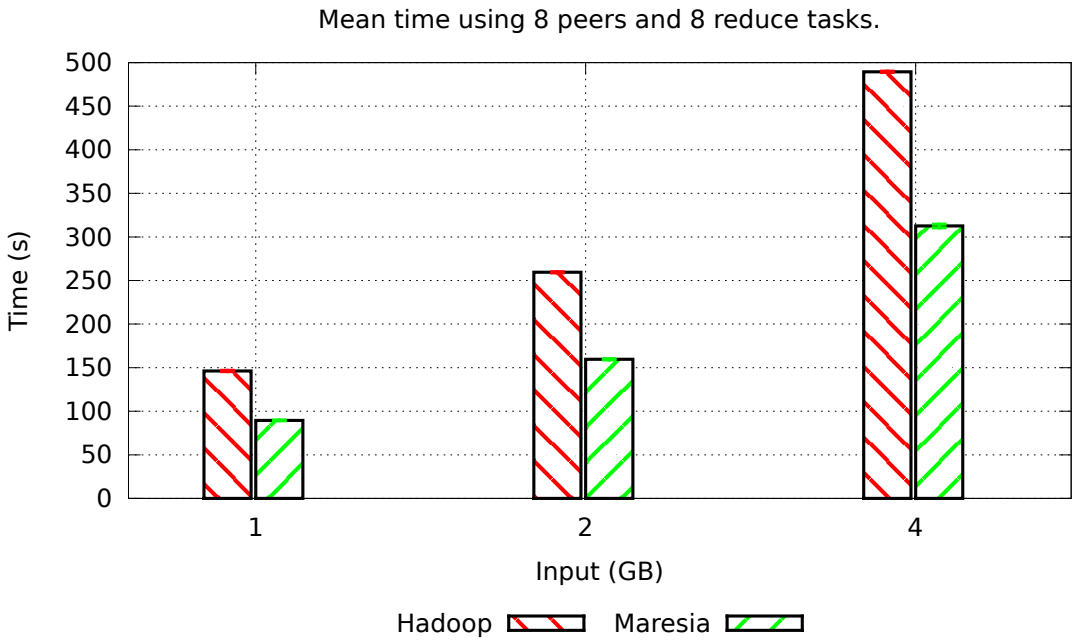


Figure 5.2: Mean time using 8 machines and 8 reduce tasks

2GB and 4GB are, respectively, 64%, 62% and 57%. It is interesting to observe that the execution times using Maresia have a small growth, while the execution times of Hadoop are almost equal to the ones obtained on the tests with 4 reduce tasks. The increasing on the execution time of Maresia is caused by the number of connections needed. With 4 reduce tasks, each one of the four peers that has a reduce task to process, will have up to seven incoming connections and up to three output connections. The other four peers will only have four output connections. On the scenario with 8 reduce tasks, all peers will have seven incoming connections and seven output connections. Maresia also has the connections to replicate the intermediate data. Hadoop also suffers from this situation. The expected behavior on this test was a reduction on the execution time, because with 8 machines and 8 reduce tasks, at least theoretically, the load is more balanced. The results for the tests using 16 machines and 8 reduce tasks are presented on Figure 5.3.

The mean time difference between Hadoop and Maresia on the scenarios with 16 machines is smaller. Maresia still faster than Hadoop, however, the difference is minimum in some cases, for example, when the workload has 1GB, the improvement is about of 6%. For the other workloads, the speedups are about of 10% and 11%. As explained before, one of the causes of this is because Hadoop does a better management of the communication process. Another point regarding the communication process is that when Maresia will send the intermediate data, it performs no other computation. Hadoop, on the other hand has processing threads and communication threads. As the number of machines increased, the same occurs with the number of connections made by each peer (one per reduce task after completing the processing of a chunk). Again, the overhead caused by the duplication of the workload is similar on both implementations. Figure 5.4 presents the results for the tests with 16 reduce tasks and 16 machines.

For this last scenario, the observed behavior was similar to the ones obtained before. Maresia is faster than Hadoop (improvement about of 0.2%, 6% and 7% for the workloads of 1GB, 2GB and 4GB, respectively) and its execution time increased with the increasing on the number of reduce tasks. The comparison of Maresia's performance

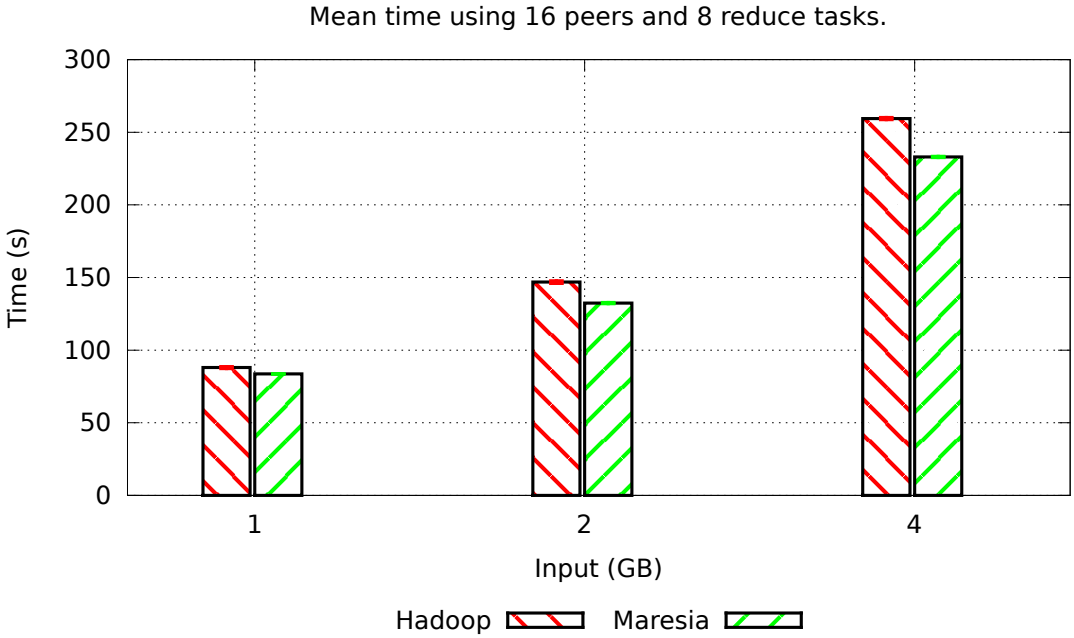


Figure 5.3: Mean time using 16 machines and 8 reduce tasks

against Hadoop exposed that a significant part of the time reduction is caused by the programming language. The analytical modeling showed that, without the fault tolerance mechanism, the execution time of Maresia will be almost the same of the master/worker architecture. However, on all experiments, even with the use of the fault tolerance mechanisms, Maresia was faster than Hadoop. Also, Maresia needs some improvements on its implementation, specially on the sending of the intermediate data. The implementation of Maresia over Hadoop is planed to happen on the future, thus, a better performance evaluation can be made. Nevertheless, it is important to highlight that the experimental tests confirmed the feasibility of use a decentralized approach to execute the MapReduce model.

5.3.2 Fault Recovery

As the focus of this work is fault tolerance, this Section presents an evaluation of the mean time execution of Maresia on scenarios with crash faults, focusing on the recovery time. These tests included cases without crash faults, with one crash fault on the map phase and with one crash fault on the reduce phase. No comparison with Hadoop is performed on these experiments due to the following:

- The difference caused by the programming language, which avoids the direct comparison;
- The absence of a reliable method to inject crash faults on Hadoop, on different executions, at the same moment, which avoids the relative comparison.

The configurations used on these experiments are the same used on the previous ones. On each execution a crash fault occurs on the same peer at the same time. The goal is to evaluate the recovery time of one map task and of one reduce task. Thus, for crash faults on the map phase, the moment is when there is only one chunk left to be processed. On

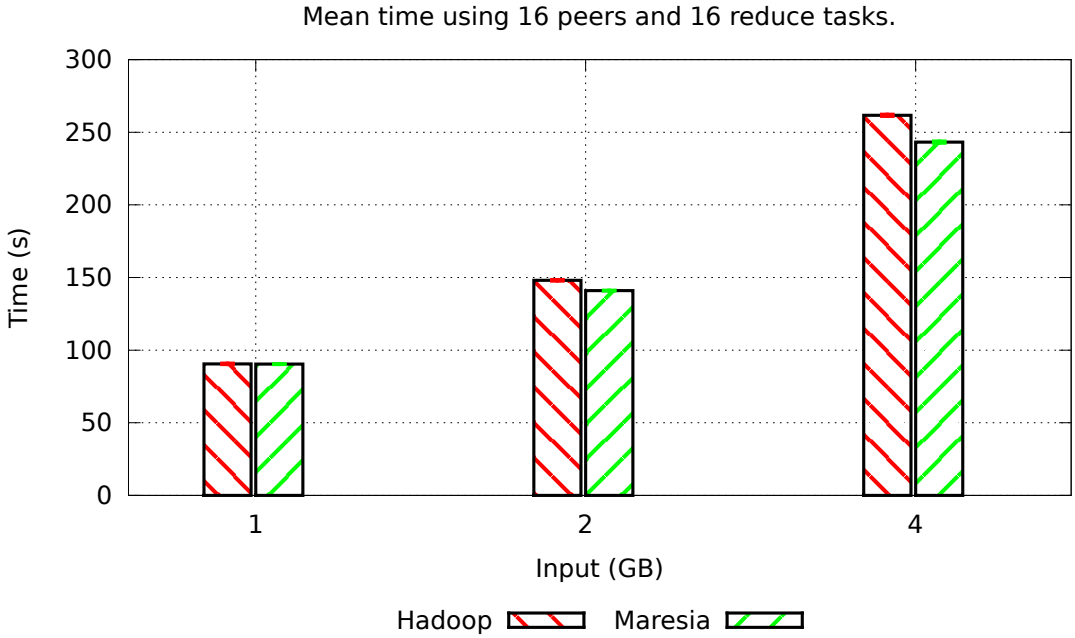


Figure 5.4: Mean time using 16 machines and 16 reduce tasks

the reduce phase, the crash fault happens when there is 1 reduce task left to be computed by the peer. As the crash faults are simulated, the time to detect it does not exist. Figure 5.5 presents the results for a scenario with 8 peers and 4 reduce tasks.

For the workload of 1GB, it is possible to observe that the time needed to finish the job with a crash fault on the reduce phase is almost equal to execution time without crash faults. Both times are around 73 seconds. This happens because the faulty machine, for some reason, was slower than its neighbor (which executed the pending tasks). For a fault on map phase, the job time increased on 12%. An important detail is that a crash fault on the map phase, also incurs on a crash fault on the reduce phase. This happens because the reduce(s) task(s) scheduled to the faulty peer must be executed by one of its neighbors. For workloads of 2GB and 4GB the behavior for a crash fault on reduce phase was similar to the one observed with 1GB. However, for a crash fault on the map phase, the total time to complete the job time decreased about 4.4% for the 4GB workload, and 1.4% for the 2GB workload. Analyzing the logs, on the scenario without crash faults, the faulty peer is 21% slower than its neighbor. On the scenario with crash faults, as there are other peers that are slower than the neighbor which re-executes the tasks of the faulty peer, the job time remains the same or decreases. For the scenario with 8 peers and 8 reduces, the execution times are presented on Figure 5.6.

The behavior of these tests are a little different from the previous experiments. For the 1GB workload, the time execution on the presence of a map crash fault increased the execution time on 14.6%. On the reduce crash fault, the difference of time is almost null. Using the 2GB workload, a fault on the reduce phase resulted on an improvement of 9.5%. The motive for this is the same explained before, the crash fault happened on a slow machine and the tasks are rescheduled to a faster peer. The crash fault on the map stage resulted on a small improvement on the execution time. On the last case, with the 4GB workload, a map crash fault incurred on an overhead of 4.8%, while a reduce crash fault is practically imperceptible. The execution times for the scenarios using 16 peers with 8 and 16 reduce tasks are presented on Figures 5.7 and 5.8, respectively.

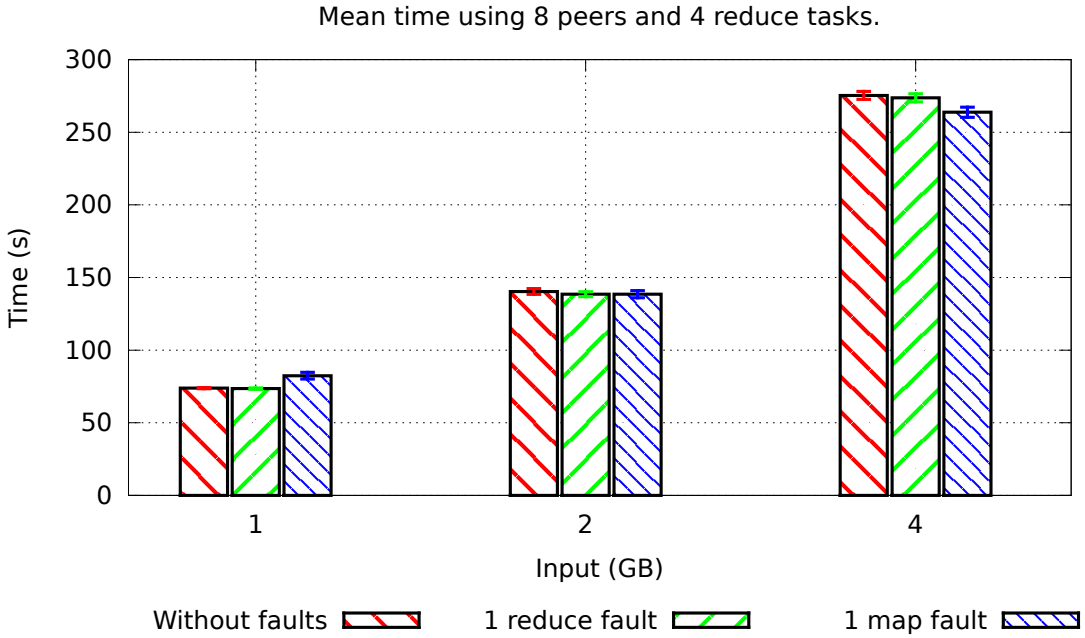


Figure 5.5: Mean time using 8 machines and 4 reduce tasks on scenarios with faults

On these experiments, for the three workloads, the reduce crash faults result on a minimal improvement of the execution time. For the tests with 8 reduce tasks and a crash fault on the map phase, the overhead decreases almost linearly with the grow of the workload. Using 1GB as input the overhead is about 9.8%, on the workload of 2GB it is 4.9%, while the 4GB workload incurs on a 2% overhead. On the experiments with 16 reduce tasks, the considerations are the same. Nevertheless, the overheads on the presence of a map crash fault for the workloads of 1GB, 2GB and 4GB are, respectively, 12.3%, 11.3% and 5.7%.

5.3.3 Replication Overhead

Another important aspect that should be evaluated is the overhead caused by the fault tolerance mechanisms, in particular the one resultant of the intermediate data replication. To evaluate this, tests (without crash faults) were performed using or not the mechanism to replicate the intermediate data. Figure 5.9 shows the results for a scenario with 8 machines and 8 reduce tasks.

For a small workload, the execution time almost duplicated using replication. As the workload increases, the overhead is reduced. With an input of 2GB the overhead is about 55% and for 4GB input it achieves 29%. The advantage of using replication of the intermediate data is that the recovery time after a crash fault is practically null. Another approach is to change the replication approach as discussed earlier. One possibility is the QMC algorithm (QUIANE-RUIZ et al., 2011), which only replicates meta-information about the localization of the intermediate data. For one side, the overhead, on scenarios without crash faults, will be smaller. Nevertheless, on the other side, the recovery time will be higher. On the other scenarios (more peers and reduce tasks) the overheads are very similar to the ones presented above.

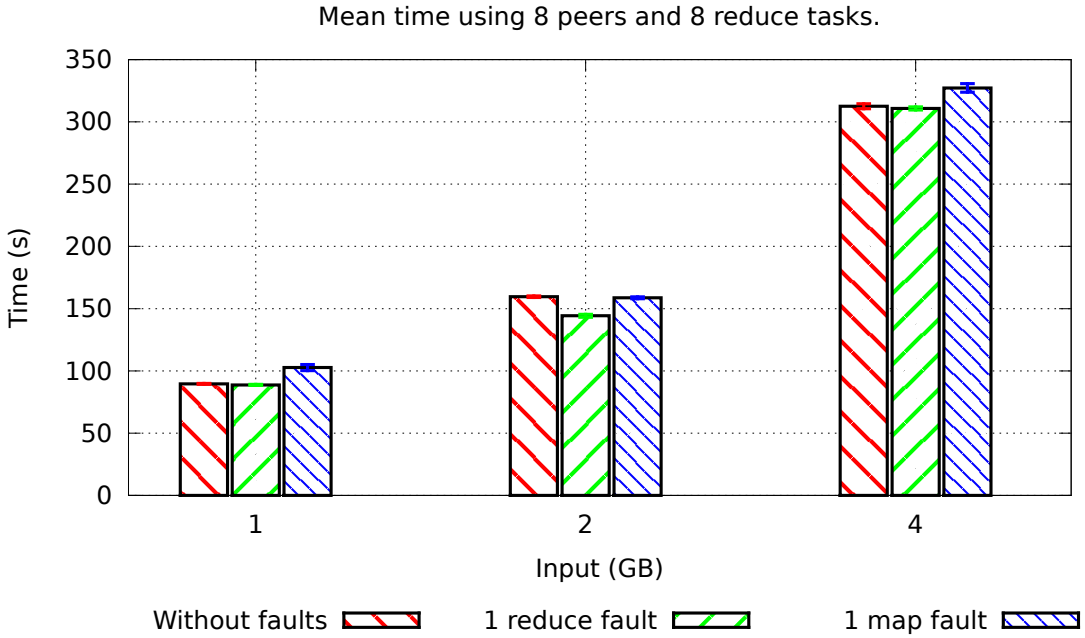


Figure 5.6: Mean time using 8 machines and 8 reduce tasks on scenarios with faults

5.4 Discussion

This Chapter presented the evaluation of the proposed architecture. Initially, an analytical model was developed to assess the time needed to execute a job and the number of messages on scenarios with and without crash faults. Assuming a scenario without crash faults, the time needed tends to be the same while the message cost of Maresia is lesser than the one needed on the original architecture. On scenarios with crash faults, the use of replication of the intermediate data can increase the message cost substantially. A better solution is the use of the QMC algorithm. With QMC the message cost decreases, however, the recovery cost increases.

After, the experimental evaluation of Maresia was presented. First, Maresia was compared against Hadoop to verify which one has the better performance on scenarios without crash faults. The experiments showed a smaller execution time for Maresia in all cases. Nevertheless, after the evaluation of the analytical model it was expected a similar performance between the two approaches, with a small advantage for Hadoop. This advantage is caused by the absence of the token phase on the master/worker architecture and the no need to persist the intermediate data. Thus, the performance comparison between Maresia and Hadoop, at least using implementations on different programming languages, does not present a great relevance. Hence, the analysis of the execution time on scenarios with crash faults were done only using Maresia.

The second group of tests shows that the recovery times are small. Also, in some cases, there is an improvement on the execution time, which happens because the faulty peer is, considerably, slower than its neighbor, which will be the responsible for execute its tasks. Another important consideration is that the decentralized approach achieved its goal to avoid the SPOFs and allow the execution of a MapReduce job even on the presence of crash faults. On Maresia there is no master node, thus, if a peer crashes, the job follows its execution normally. If the faulty peer is acting as the master node on the original architecture, all progress already done by the job is lost.

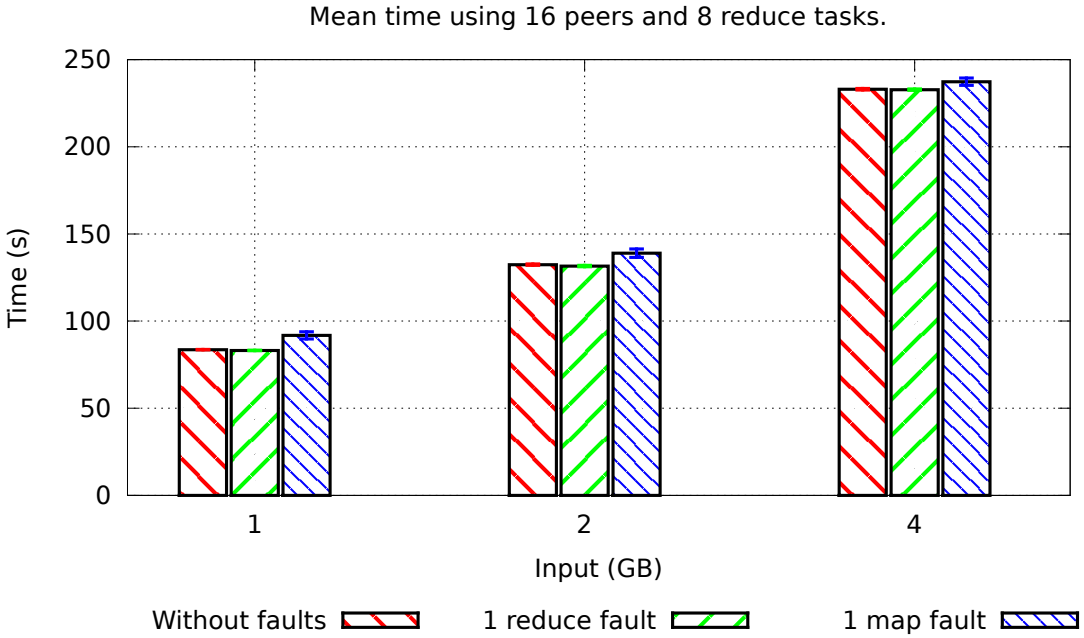


Figure 5.7: Mean time using 16 machines and 8 reduce tasks on scenarios with faults

The last group of tests highlighted that the replication of the intermediate data causes a significant overhead on the execution time. The persistence of the intermediate data is important the reduce the recovery time after a crash fault. However, if the probability of a crash fault is small, the replication algorithm can be replaced by one with less overhead but a higher recovery time, such as QMC.

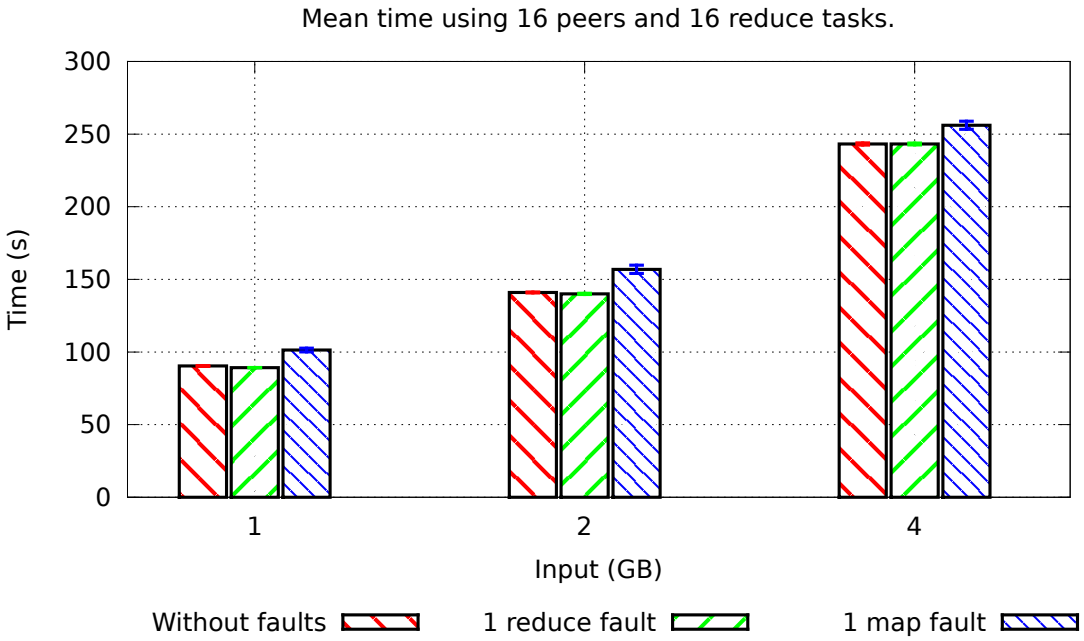


Figure 5.8: Mean time using 16 machines and 16 reduce tasks on scenarios with faults

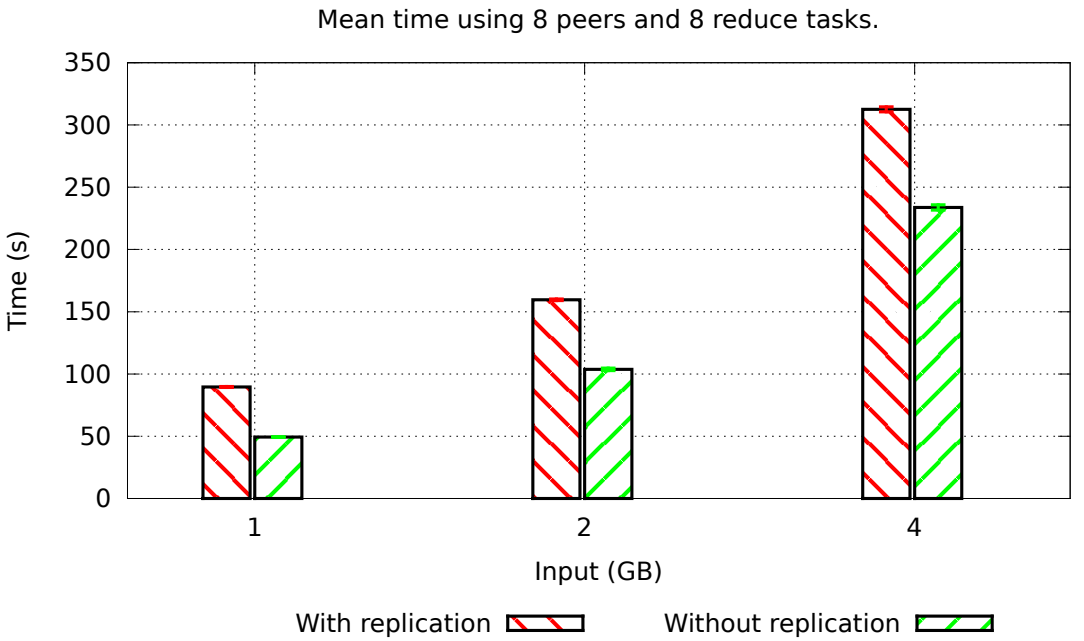


Figure 5.9: Mean time with and without the replication of the intermediate data

6 CONCLUSIONS

This work presented a new proposal to avoid the SPOFs on MapReduce. Inspired by Chord, Maresia architecture uses a decentralized approach to provide fault tolerance. The main contributions of Maresia are an architecture without SPOFs, which is platform-independent and has no need to add new machines to tolerate crash faults.

To avoid both SPOFs, the master of the job and the NameNode, without the necessity of adding new machines, Maresia leverages the main characteristics of Chord: its decentralized architecture and the DHT. The DHT was used to schedule the input chunks and the reduce tasks, while the decentralized approach guarantees that all nodes of the system have the same attributions. The platform independence was achieved with a model that do not use services from specific environments, like the ones used on Cloud MapReduce and Azure MapReduce.

The evaluation presented a comparison of the proposed architecture with the original one. An analytical model was provided to compare the costs of messages and time. On scenarios without crash faults (and without the replication of the intermediate data), Maresia tends to have less messages than the original approach and a similar execution time. Nevertheless, when the intermediate data is replicated (to allow the fault tolerance), the message costs increases. A better approach for the persistence of the intermediate data, like the QMC algorithms, attenuates this problem. The experimental tests, on scenarios with crash faults, shown that the recovery costs are small, including cases where the execution time decreased. Thereby, the proposed solution achieved its goals of avoid the SPOFs and perform the recovery after a crash fault on a feasible time.

6.1 Future Work

As future work, there are some important points that should receive attention. The first is the implementation of the Maresia architecture over Hadoop. The work presented on this dissertation showed the feasibility of using a decentralized approach to execute MapReduce jobs. However, the current implementation does not have all the features existing on Hadoop, such as the compression of the intermediate data and the backup tasks. Also, an implementation of Maresia over Hadoop will turn it more reliable for use. Another positive aspect is the possibility of use the existing Hadoop applications on Maresia (on the current version of Maresia, all applications should be re-coded).

A second aspect that need to be improved is the intermediate data persistence. As presented on Section 5.3.3 the replication overhead can achieve up to 97% for small workloads. An alternative is the use of a solution with a higher recovery time but a lower replication overhead. Thus, the QMC algorithm (QUIANE-RUIZ et al., 2011) can be applied to reduce the replication cost, as showed on Section 5.2.

Another point is the development of a complete fault detection mechanism. Therewith, will be possible to use Maresia on real situations. One more fault tolerance aspect that should be treated is the byzantine behavior of the peers. Thus, the proposed approach can be evaluated, for example, on opportunistic environments. Also, it is planned to evaluate the behavior of Maresia on other scenarios, such as cloud computing environments. Finally, to assist on the evaluation process, an implementation of Maresia over the MRSG simulator is planned.

APPENDIX A RESUMO EM PORTUGUÊS

A.1 Introdução

Durante os últimos anos, a quantidade de dados produzidos por aplicações cresceu consideravelmente. Exemplos disso são o Facebook, que hospeda aproximadamente 700 terabytes de dados em tabelas (THUSOO et al., 2009) e um petabyte em fotos. Outros exemplos são a bolsa de valores de Nova Iorque, que gera um terabyte de dados por dia, e o *Large Hadron Collider* (LHC), que produzirá cerca de 15 petabytes de dados por ano (WHITE, 2009).

Contudo, para transformá-los em informações relevantes é necessário processar estes dados. Desta forma, foram propostos modelos de programação paralela para processamento distribuído sobre grandes volumes de dados. Estes modelos buscam abstrair do programador a necessidade de lidar com questões referentes ao escalonamento de tarefas, distribuição dos dados, comunicação e tolerância a falhas. Com este objetivo a Google desenvolveu o modelo MapReduce (DEAN; GHEMAWAT, 2004).

Além da própria Google, o MapReduce é largamente utilizado em grandes empresas, como por exemplo, Amazon, AOL, Ebay, Facebook, IBM, Last.fm, The New York Times, Twitter, Yahoo!, entre outras (APACHE SOFTWARE FOUNDATION, 2011). Em todos estes casos, o MapReduce é utilizado em grandes *data centers* formados por máquinas de prateleira. Com isso, a probabilidade da existência de falhas aumenta (DEAN; GHEMAWAT, 2004).

Contudo, em sua versão original, o MapReduce trata apenas falhas ocorridas nos *workers*, fazendo com que o *master* e o servidor de nomes do Sistema de Arquivos Distribuíbo caracterizem-se por ser pontos de falha única (SPOF - *Single Point Of Failure*, em inglês). Entre as soluções propostas para este problema encontra-se a replicação do estado atual do *master* em máquinas de *backup* e arquiteturas específicas para ambientes de *cloud computing*, onde utilizam-se serviços oferecidos pelos sistemas operacionais de nuvem para tolerar falhas.

No primeiro caso, as novas máquinas serão efetivamente utilizadas apenas quando ocorrer uma falha em um dos SPOFs. Já no segundo, a utilização dos serviços específicos de ambientes de *cloud computing* restringe os locais onde a solução pode ser aplicada. Assim, este trabalho apresenta uma nova arquitetura para o modelo MapReduce, a qual é denominada MARESIA (*MApREduce in a SImple Architecture*) e tem como inspiração o Chord (STOICA et al., 2001). Nela os SPOFs são evitados sem a necessidade de uso de serviços específicos de um determinado ambiente e sem a adição de novos componentes no sistema.

O restante deste artigo está organizado da seguinte forma: a Seção A.2 apresenta uma visão geral do modelo MapReduce. Já a Seção A.3 apresenta os principais trabalhos

relacionados. Após, na Seção A.4, é apresentada a arquitetura Maresia. Por fim, as Seções A.5 e A.6 apresentam, respectivamente, a avaliação da proposta e as conclusões do trabalho.

A.2 Modelo MapReduce

Criado pela Google, o MapReduce é um modelo de programação paralela para processamento distribuído de grandes volumes de dados. O modelo foi desenvolvido pois a Google percebeu que um número significativo de suas aplicações realizava um pré-processamento dos dados para então realizar a computação dos mesmos. Neste pré-processamento os dados eram separados em pares da forma (chave, valor). Assim, inspirado nas primitivas *map* e *reduce* das linguagens funcionais, a Google desenvolveu o modelo MapReduce. Este é composto de duas fases, uma de mapeamento dos dados e outra de redução. O seu principal objetivo é abstrair do programador a necessidade de lidar com questões referentes a criação e escalonamento das tarefas, distribuição dos dados, comunicação e tolerância a falhas (DEAN; GHEMAWAT, 2004).

O modelo MapReduce é organizado na forma de uma arquitetura *master-worker*. Nela, o *master* é responsável por realizar o escalonamento das tarefas e o tratamento de falhas. Já os nós *workers* são responsáveis por realizar o processamento das tarefas de *map* e *reduce*. Além disso, é utilizado um SAD para armazenar os dados de entrada e saída.

O SAD também utiliza uma arquitetura *master-worker*, onde o nó *master* é responsável por armazenar uma tabela com a localização dos dados e pelo tratamento de falhas, por exemplo, mantendo um número mínimo de réplicas dos arquivos. Já os nós *workers* são responsáveis por armazenar os dados. Para minimizar transferências de dados, os nós utilizados para executar as funções *map* e *reduce* são também os *workers* do SAD. Na implementação da Google do modelo MapReduce o Google File System (GHEMAWAT; GOBIOFF; LEUNG, 2003) é utilizado como SAD.

Tanto no modelo MapReduce quanto no SAD, a comunicação ocorre na forma *master-worker* apenas para atribuição de tarefas. A transferência de dados, seja dos dados de entrada da fase de mapeamento ou dos pares intermediários, produzidos na fase de mapeamento e que serão consumidos pela função de redução, é feita diretamente entre os *workers*. Isto evita que o *master* torne-se um gargalo para a comunicação.

Conforme explicado anteriormente, uma aplicação MapReduce é composta por duas fases principais, uma fase de mapeamento e outra de redução. A solicitação para execução de uma aplicação MapReduce, chamada de *job*, é feita através do *master*. O primeiro passo no *workflow* do MapReduce é a divisão dos dados de entrada e a sua distribuição no SAD. Cada parte dos dados é chamada de *chunk*. Além disso, todos os *chunks* possuem o mesmo tamanho. Após esta etapa, os *workers* começam o processamento da função *map*. A função *map* produzirá pares intermediários na forma (chave, valor). Estes serão consumidos pela função de redução. É importante salientar que os pares intermediários não são armazenados no SAD.

Cabe salientar que a Google desenvolveu o MapReduce para uso próprio apenas. Assim, a partir da especificação do modelo, a Apache Software Foundation desenvolveu uma implementação livre do modelo MapReduce, denominada Hadoop (WHITE, 2009) e um novo SAD, chamado HDFS (Hadoop Distributed File System) (SHVACHKO et al., 2010). Assim, devido a sua disponibilidade, o Hadoop tornou-se a versão mais utilizada para o desenvolvimento de aplicações MapReduce.

A.3 Trabalhos relacionados

Diferentes trabalhos foram propostos para tratar falhas no modelo MapReduce. Estes propõem otimizações nos mecanismos de falhas já existentes no modelo (KO et al., 2010) (QUIANE-RUIZ et al., 2011), tolerância ao fornecimento de respostas incorretas por parte dos *workers*, no caso de ambientes onde os workers não são confiáveis, por exemplo, *desktop grids*, (WEI et al., 2009) (MOCA; SILAGHI; FEDAK, 2011) e formas de evitar os pontos de falha única (CLEMENT et al., 2009) (MAROZZO; TALIA; TRUNFIO, 2011), as quais são o foco deste trabalho.

Conforme mencionado anteriormente, o MapReduce possui dois pontos de falha única, o *master* do *job* a ser executado e o servidor de nomes do SAD. Uma falha em qualquer um dos dois pode resultar na perda de todo o processamento já realizado. Dessa forma, técnicas de tolerância a falhas são utilizadas para evitar desperdício de recursos computacionais.

Uma alternativa proposta é a inserção de novos nós no sistema (WANG et al., 2009) (MAROZZO; TALIA; TRUNFIO, 2011). Em ambas as propostas, os novos nós atuarão como réplicas do *master* do *job* MapReduce. Estas irão manter-se sincronizadas com o *master* durante a execução do *job*. No caso de uma falha do *master* um algoritmo de eleição é utilizado entre as réplicas para decidir quem será o novo *master*. As duas soluções mostram-se funcionais para evitar que o *master* seja um ponto de falha única, contudo, nenhuma delas tolera falhas no servidor de nomes do SAD. Além disso, os novos nós inseridos no sistema só serão efetivamente utilizados caso ocorra uma falha no *master*, podendo resultar em recursos ociosos caso estas não ocorram.

Além do *master*, o outro ponto de falha única do modelo MapReduce é o servidor de nomes do SAD. Para lidar com esta questão foi proposta a biblioteca *UpRight* (CLEMENT et al., 2009). Esta é uma biblioteca de uso geral para tornar aplicações tolerantes a falhas. A sua principal característica é permitir que programas tornem-se tolerantes a falhas sem a necessidade de mudanças significativas em seus códigos. O HDFS foi utilizado como aplicação teste da biblioteca, sendo necessária a inserção de 1800 linhas em seu código fonte para torná-lo tolerante a falhas.

Outra alternativa para evitar que o servidor de nomes do SAD seja um ponto de falha única do sistema é a sua substituição por um SAD P2P. Para isso foi proposto o Brisk, que busca integrar o Hadoop com o Cassandra, um SAD P2P (DATASTAX, 2011). Nesta arquitetura, todos os nós são capazes de realizar a busca por arquivos, evitando assim que o servidor de nomes seja ponto de falha única. Entretanto, os autores não apresentam testes e nem detalham o funcionamento do sistema. Embora nenhuma das soluções para evitar o ponto de falha única do SAD necessite de novas máquinas, elas não apresentam solução para lidar com falhas no *master* do MapReduce.

Além das soluções já apresentadas, (LIU; ORBAN, 2011) e (GUNARATHNE et al., 2010) apresentam soluções descentralizadas para execução do MapReduce em ambientes de *cloud computing*. A descentralização da arquitetura faz com que os pontos de falha única deixem de existir. A ideia das propostas é aproveitar os serviços oferecidos pelos sistemas operacionais das nuvens da Amazon e da Microsoft, respectivamente, de forma a simplificar a implementação e melhorar o desempenho do modelo.

Em ambos os casos o SAD é substituído pelo sistema de arquivos da nuvem e o *master* é substituído por um mecanismo de mensagens. Neste são armazenadas mensagens contendo a descrição de cada uma das tarefas de *map* e de *reduce*, além da localização dos dados a serem processados. Assim, cada um dos *workers* consulta o sistema de mensagens e fica responsável por processar uma determinada tarefa. Os dados produzidos

pelas tarefas de *map* são armazenados em um serviço de filas, onde cada posição de fila corresponde a uma chave. Após a conclusão das tarefas de *map*, as tarefas de *reduce* serão executadas e a saída será armazenada em uma fila de saída.

Embora as propostas descentralizadas evitem os dois pontos de falha única presentes no modelo MapReduce, estas utilizam mecanismos oferecidos pelos provedores de *cloud computing* para tolerar falhas. Assim, torna-se inviável a utilização destas implementações em ambientes que não sejam os próprios provedores. Desta forma, verifica-se a ausência de uma arquitetura sem pontos de falha única, independente de ambiente e que não necessite a inserção de máquinas de *backup*.

A.4 Proposta

Com o objetivo de evitar os pontos de falha única no MapReduce propõe-se uma nova arquitetura para o modelo. Esta é descentralizada e tem como principal base o Chord (STOICA et al., 2001).

A.4.1 Maresia

A arquitetura Maresia tem como objetivo não possuir os pontos de falha única da arquitetura original do modelo MapReduce. A principal motivação para isto é evitar que o processamento já realizado de um *job* seja perdido caso ocorra uma falha no *master* ou no servidor de nomes do SAD. A seguir serão apresentados detalhes de seu funcionamento.

A.4.1.1 Restrições

Assim como a original, a arquitetura Maresia possui restrições. São elas:

- Não serão adicionados *workers* durante a execução de um *job*;
- Não há comportamento bizantino por parte dos *workers*;
- Há disponibilidade de rede e baixa latência de comunicação.

Todas as restrições presentes existem também na versão *master/worker* proposta pela Google. Estas são decorrentes do ambiente de execução sobre o qual o MapReduce será executado, no caso *clusters*. A versão da Google ainda assume que os *workers* terão a mesma capacidade computacional, e, consequentemente, processarão uma determinada quantidade de dados em um mesmo intervalo de tempo. A arquitetura Maresia prevê a possibilidade de *workers* heterogêneos, não sendo necessária esta restrição. Cabe salientar também que, embora utilize ideias originárias de sistemas P2P, a arquitetura Maresia foi projetada para execução em *clusters*. Isto ocorre pois o foco principal deste trabalho é apresentar uma arquitetura sem pontos de falha única para o modelo MapReduce.

A.4.1.2 Funcionamento

Primeiramente será apresentado o funcionamento da arquitetura Maresia na ausência de falhas. Após, serão mostrados os mecanismos utilizados para tornar a arquitetura tolerante a falhas. A primeira etapa de funcionamento do modelo MapReduce na arquitetura Maresia é a formação do anel virtual. Assim como o Chord, assume-se que existe um mecanismo externo para realizar o *bootstrap* dos *workers*. Uma falha neste mecanismo impediria apenas que novos *workers* sejam adicionados ao anel antes do começo do *job*. Contudo, o *job* será capaz de iniciar com os *peers* já presentes no anel.

Após a formação do anel, cada *peer* possuirá um identificador único e ficará responsável por um conjunto de chaves. Diferentemente do Chord utiliza-se um anel bidirecional. Esta modificação é necessária para realizar o tratamento de falhas das tarefas do MapReduce, o qual será explicado na subseção A.4.1.3. A última etapa antes de iniciar a execução do *job* é a distribuição dos dados de entrada entre os *peers*. Assim, o *peer* que possui os dados de entrada, chamado de originador, irá separá-los em *chunks* de mesmo tamanho para então enviá-lo aos demais *peers*. Para determinar qual *peer* ficará responsável por cada *chunk* utiliza-se uma DHT. Será utilizada um função de *hash* de forma que os dados sejam distribuídos de maneira aproximadamente homogênea.

Com o término da divisão dos dados a fase de *map* poderá ser iniciada. Nesta etapa cada *peer* será responsável por processar os *chunks* armazenados localmente. Após processar cada *chunk* o *peer* aplica a função de *hash* sobre cada um dos pares intermediários produzidos e os envia para o *peer* responsável. Na fase de envio dos pares intermediários é adotada uma estratégia diferente da utilizada no modelo original do MapReduce. No modelo original os *workers* que irão executar a função de *reduce* solicitam os dados produzidos na etapa de *map*. Já na arquitetura Maresia é realizado o *push* dos dados para os *peers* que executarão as tarefas de *reduce*.

Para o início da fase de *reduce* é necessário que todos os dados produzidos na fase de *map* já tenham sido enviados para os *peers* que executarão a função de *reduce*. Assim, é necessário que todos os *peers* sejam comunicados que o envio dos pares intermediários produzidos na fase de *map* foi concluído. Uma alternativa para isto é o *broadcast* de mensagens indicando que o *peer* terminou o processamento da fase de *map*. Contudo, esta alternativa, além de ter uma complexidade de mensagens $O(n)$ e necessita que todos os *peers* sejam conhecidos por cada nó.

Assim, optou-se pelo uso de *tokens* para indicar o término da fase da *map*. Quando o *peer* que forneceu os dados de entrada para o *job* termina de processar os dados sob sua responsabilidade dois *tokens* são criados. Um será enviado para o vizinho da direita e o outro para o vizinho da esquerda. Quando um *peer* receber os dois *tokens* ele estará apto a iniciar a fase de *reduce*. É importante salientar que um *peer* só repassará os *tokens* quando os dados sob sua responsabilidade já tiverem sido processados pela função de *map*. A complexidade de mensagens desta alternativa é $O(2 * n)$. Além disso, não é necessário saber o número total de *chunks* nem ter conhecimento sobre todos os *peers*.

A função de *hash* prevê que os *chunks* serão distribuídos de forma aproximadamente balanceada entre os *peers*. Assim, no caso de ambientes homogêneos o processamento da etapa de *map* encerrará de forma quase sincronizada entre todos os *peers*. Já em ambientes heterogêneos está condição não é necessariamente verdadeira. Com isso, uma *peer* mais lento poderia reter a passagem dos *tokens* enquanto o restante dos *peers* está ocioso. Para lidar com esta questão utiliza-se um mecanismo de roubo de tarefas. Quando um *peer* termina de processar os seus dados ele consulta os vizinhos para saber se existe alguma tarefa pendente. Em caso afirmativo ele rouba a tarefa e a processa.

Por fim, cada *peer* executa a função de *reduce* sobre os dados oriundos da fase de *map*. Ao término do processamento cada *peer* informa ao nó originador do *job* os resultados ou então a localização dos resultados produzidos.

A.4.1.3 Tratamento de falhas

A subseção A.4.1.2 apresentou o funcionamento da arquitetura Maresia em ambientes sem falhas. Desta forma verificou-se a viabilidade de executar um *job* MapReduce em um ambiente descentralizado. Contudo, o foco principal deste trabalho é uma arquitetura

capaz de tolerar os mesmos tipos de falhas da versão original do MapReduce e ainda evitar os pontos de falha única. Assim, esta subseção apresentará os mecanismos de tolerância a falhas da arquitetura Maresia.

O primeiro aspecto a ser garantido é a disponibilidade dos dados. Diferentemente da versão original, a arquitetura Maresia não utiliza um SAD. Logo, a replicação dos dados fica a cargo da nova arquitetura. Assim, para lidar com este aspecto, cada *peer* replica os dados de entrada que estão sob sua responsabilidade nos *peers* vizinhos.

Outro aspecto fundamental é a reexecução de tarefas no caso de falha em um *peer*. Desta forma, para detectar falhas de *peers* utiliza-se um mecanismo de *heartbeat* similar ao utilizado no MapReduce original. Como não existe um *peer* central, cada *peer* troca mensagens de *hearbeat* com seus vizinhos, de forma periódica. A mensagem de *hearbeat* contém informações sobre as tarefas de *map* e *reduce* já processadas pelo *peer* e sobre a posse dos *tokens*.

Quando um *peer* termina de processar uma tarefa de *map*, o que inclui o envio dos pares intermediários para os outros *peers*, ele comunica através do *heartbeat* a conclusão da tarefa. Se um *peer* falhar, os seus vizinhos, os mesmos que possuem as réplicas dos dados, decidirão qual dos *peers* irá processar os *chunks* ainda não computados (caso existam). O mesmo procedimento é aplicado para as tarefas de *reduce*.

Ao término da fase de *map* ocorre o lançamento de *tokens* por parte do *peer* originador do *job*. Caso tenha ocorrido uma falha no *peer* originador, o *peer* que ficou responsável pelas chaves do originador será responsável por lançar os *tokens*. A mesma decisão é adotada para o *peer* que receberá o resultado final. Já o controle para verificar se um *token* foi perdido é realizado através do *heartbeat*. Quando o *peer* está de posse de um *token* ele comunica aos seus vizinhos. No momento em que o *token* é repassado, ele encaminha, via *heartbeat*, um aviso dizendo que o *token* foi repassado. Caso o *peer* falhe antes de repassar o *token*, quando o seu vizinho anterior detectar a falha do *peer*, ele gerará um novo *token* e o encaminhará ao seu sucessor. Antes, porém, os vizinhos verificarão se o *peer* que falhou possuía alguma tarefa pendente. Em caso afirmativo, a tarefa será executada.

A persistência dos pares intermediários é outro ponto a ser considerado. Para isto, duas opções são possíveis. A primeira é replicar os pares intermediários nos vizinhos do *peer* que recebeu os dados. Assim, no caso de falhas durante a fase de *reduce* utiliza-se o mesmo mecanismo utilizado para tratar falhas nas tarefas de *map*. Contudo, devido à grande quantidade de pares intermediários o custo de replicação pode ser alto. Uma alternativa é utilizar o algoritmo QMC (QUIANE-RUIZ et al., 2011). Este replica apenas meta-informações sobre a localização dos pares intermediários, assim, o custo é reduzido. Porém, o tempo de recuperação em caso de falha aumenta visto que será necessário realizar transferências de dados. Desta forma, pode-se optar por uma alternativa com custo maior e tempo de recuperação baixo ou então, baixo custo e maior tempo de recuperação.

A.5 Avaliação

Como forma de avaliar a proposta foi realizada a modelagem analítica do custo de mensagens e do tempo de execução. Além disso, foi feita uma avaliação experimental com foco no desempenho da nova arquitetura, no tempo necessário para recuperação de uma falha e na sobrecarga causada pelos mecanismos de tolerância a falhas.

Através da modelagem analítica pode-se constatar que, em cenários sem falhas e sem o uso dos mecanismos de replicação, a arquitetura Maresia apresenta um custo de

mensagens inferior ao da arquitetura original.Por outro lado, quando é considerando um cenário com a existência de falhas, a abordagem proposta acaba tendo um custo de mensagens superior. Isto ocorre, principalmente, pelo fato de ser necessário realizar a replicação dos pares intermediários. Com isso, faz-se necessária uma outra alternativa. A diminuição das mensagens em cenários com falhas pode ser obtida através do uso de um algoritmo que não necessite replicar os pares intermediários, como por exemplo o QMC. Neste caso, mesmo em cenários com falhas, a arquitetura proposta tem um custo de mensagens menor. Já em relação ao tempo de execução, a modelagem analítica mostra que a diferença reside apenas na etapa de passagem de *token*.

Para avaliar a arquitetura de forma experimental foi desenvolvido um protótipo. Na primeira avaliação foi comparado o seu desempenho em diferentes cenários sem falhas. Além disso, os mesmos testes foram repetidos no Hadoop. Constatou-se que a linguagem de programação utilizada impactou de forma significativa nos resultados. Com isso, os demais testes foram feitos apenas usando o protótipo da arquitetura Maresia.

Na avaliação do tempo de recuperação de falhas observou-se que este é pequeno, independente do momento em que a falha ocorreu. Além disso, em alguns casos houve uma melhora no tempo de execução. Isto ocorreu pois o *peer* que falhou era o mais lento e as suas tarefas foram executadas por um *peer* mais rápido. Já na avaliação da sobrecarga causada pelo mecanismo de replicação dos pares intermediários confirmaram-se os resultados da modelagem analítica. Assim, fica evidente a necessidade de substituição do algoritmo de replicação por um com menor sobrecarga, como por exemplo o QMC.

A.6 Conclusões e Trabalhos Futuros

Este trabalho apresentou uma nova proposta para lidar com pontos de falha única no modelo MapReduce. Inspirada pelo Chord, a arquitetura Maresia usa uma abordagem descentralizada para garantir a tolerância a falhas. As principais contribuições deste trabalho são uma arquitetura sem pontos de falhas única, com uma implementação independente de plataforma e que não precisa da inserção de novos componentes para atingir a tolerância a falhas. A avaliação realizada mostra a viabilidade de uso da arquitetura Maresia tanto em cenários com falhas quanto sem.

A.6.1 Trabalhos Futuros

Como trabalhos futuro existem alguns aspectos que devem receber atenção. O primeiro deles é a implementação da arquitetura proposta sobre o Hadoop. Isto permitirá que as aplicações já existentes possam executar de forma decentralizada sem que seja necessário modificar o seu código. Adicionalmente, espera-se melhorar o mecanismo de persistência dos pares intermediários, de forma a diminuir a sobrecarga causada por este. Por fim, espera-se realizar modificações no simulador MRSG para permitir que a arquitetura proposta seja simulada e avaliada nesta ferramenta.

REFERENCES

APACHE SOFTWARE FOUNDATION, A. **Powered by - Hadoop Wiki**. Available at http://wiki.apache.org/hadoop/PoweredBy - Last access on November, 2012.

ARANTES, L.; SOPENA, J. Garantindo a Circulacao e Unicidade do Token em Algoritmos com Nós Organizados em Anel Sujeitos a Falhas. In: WORKSHOP DE TOLERâNCIA A FALHAS (WTF), 2010. **Proceedings...** [S.l.: s.n.], 2010. p.17–30.

CHU, C. T. et al. Map-Reduce for Machine Learning on Multicore. In: TWENTIETH ANNUAL CONFERENCE ON NEURAL INFORMATION PROCESSING SYSTEMS. **Proceedings...** MIT Press, 2006. p.281–288.

CLEMENT, A. et al. Upright cluster services. In: ACM SIGOPS 22ND SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, New York, NY, USA. **Proceedings...** ACM, 2009. p.277–290. (SOSP '09).

DATASTAX. **Getting Started with Brisk**: hadoop powered by cassandra. Available at http://www.datastax.com/docs/0.8/brisk/index - Last access on October, 2012.

DEAN, J. Large-scale distributed systems at google: current systems and future directions. In: ACM SIGOPS INTERNATIONAL WORKSHOP ON LARGE SCALE DISTRIBUTED SYSTEMS AND MIDDLEWARE (LADIS), 3. **Proceedings...** [S.l.: s.n.], 2009. Keynote speech.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In: SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - OSDI '04, 6. **Proceedings...** [S.l.: s.n.], 2004. p.137–150.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Commun. ACM**, New York, NY, USA, v.51, n.1, p.107–113, Jan. 2008.

DEAN, J.; GHEMAWAT, S. MapReduce: a flexible data processing tool. **Commun. ACM**, New York, NY, USA, v.53, n.1, p.72–77, Jan. 2010.

FADIKA, Z.; GOVINDARAJU, M. DELMA: dynamically elastic mapreduce framework for cpu-intensive applications. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2011. v.0, p.454–463.

FANG, W. et al. Mars: accelerating mapreduce with graphics processors. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.22, n.4, p.608–620, Apr. 2011.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S. T. The Google file system. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, New York, NY, USA. **Proceedings...** ACM, 2003. n.5, p.29–43. (SOSP '03, v.37).

GUNARATHNE, T. et al. MapReduce in the Clouds for Science. In: IEEE SECOND INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE (CLOUDCOM), 2010. **Proceedings...** [S.l.: s.n.], 2010. p.565–572.

HERODOTOU, H.; BABU, S. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. **Proceedings of the VLDB Endowment**, [S.l.], v.4, n.11, 2011.

ISARD, M. et al. Dryad: distributed data-parallel programs from sequential building blocks. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.41, n.3, p.59–72, Mar. 2007.

KADIRVEL, S.; FORTES, J. A. B. Towards self-caring mapreduce: proactively reducing fault-induced execution-time penalties. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND SIMULATION (HPCS), 2011. **Proceedings...** IEEE, 2011. p.63–71.

KO, S. Y. et al. Making cloud intermediate data fault-tolerant. In: ACM SYMPOSIUM ON CLOUD COMPUTING, 1., New York, NY, USA. **Proceedings...** ACM, 2010. p.181–192. (SoCC '10).

LIN, H. et al. MOON MapReduce On Opportunistic eNvironments. In: ACM INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 19., New York, NY, USA. **Proceedings...** ACM, 2010. p.95–106. (HPDC '10).

LIU, H.; ORBAN, D. Cloud MapReduce: a mapreduce implementation on top of a cloud operating system. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID 2011. **Proceedings...** [S.l.: s.n.], 2011.

MAROZZO, F.; TALIA, D.; TRUNFIO, P. A Framework for Managing MapReduce Applications in Dynamic Distributed Environments. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2011. **Proceedings...** IEEE, 2011. p.149–158.

MARTIN, A. et al. Low-Overhead Fault Tolerance for High-Throughput Data Processing Systems. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2011. v.0, p.689–699.

MATSUNAGA, A.; TSUGAWA, M.; FORTES, J. CloudBLAST: combining mapreduce and virtualization on distributed resources for bioinformatics applications. In: IEEE FOURTH INTERNATIONAL CONFERENCE ON ESCIENCE, 2008. ESCIENCE '08., Los Alamitos, CA, USA. **Proceedings...** IEEE, 2008. v.0, p.222–229.

MOCA, M.; SILAGHI, G. C.; FEDAK, G. Distributed Results Checking for MapReduce in Volunteer Computing. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WORKSHOPS AND PHD FORUM (IPDPSW), 2011. **Proceedings...** IEEE, 2011. p.1847–1854.

NICOLAE, B. et al. BlobSeer: next-generation data management for large scale infrastructures. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.71, p.169–184, February 2011.

QUIANE-RUIZ, J. A. et al. RAFTing MapReduce Fast recovery on the RAFT. In: IEEE 27TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), 2011. **Proceedings...** IEEE, 2011. p.589–600.

RANGER, C. et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: IEEE 13TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, 2007., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2007. v.0, p.13–24.

SCHIPER, N.; TOUEG, S. A robust and lightweight stable leader election service for dynamic systems. In: DEPENDABLE SYSTEMS AND NETWORKS WITH FTCS AND DCC, 2008. DSN 2008. IEEE INTERNATIONAL CONFERENCE ON. **Proceedings...** IEEE, 2008. p.207–216.

SHVACHKO, K. et al. The Hadoop Distributed File System. In: IEEE 26TH SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.1–10.

STOICA, I. et al. Chord: a scalable peer-to-peer lookup service for internet applications. In: APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, 2001., New York, NY, USA. **Proceedings...** ACM, 2001. p.149–160. (SIGCOMM '01).

TANG, B. et al. Towards MapReduce for Desktop Grid Computing. In: INTERNATIONAL CONFERENCE ON P2P, PARALLEL, GRID, CLOUD AND INTERNET COMPUTING, 2010., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2010. p.193–200. (3PGCIC '10).

THUSOO, A. et al. Hive: a warehousing solution over a map-reduce framework. In: VLDB ENDOW. **Proceedings...** VLDB Endowment, 2009. v.2, n.2, p.1626–1629.

WANG, F. et al. Hadoop high availability through metadata replication. In: FIRST INTERNATIONAL WORKSHOP ON CLOUD DATA MANAGEMENT, New York, NY, USA. **Proceedings...** ACM, 2009. p.37–44. (CloudDB '09).

WANG, Y.; WEI, J. VIAF: verification-based integrity assurance framework for mapreduce. In: IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING (CLOUD), 2011. **Proceedings...** IEEE, 2011. p.300–307.

WEI, W. et al. SecureMR: a service integrity assurance framework for mapreduce. In: ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE, 2009., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society, 2009. v.0, p.73–82.

WHITE, T. **Hadoop - The Definitive Guide**. Original.ed. [S.l.]: O'Reilly Media, 2009.

WILEY, K. et al. Astronomy in the Cloud: using mapreduce for image co-addition. **Publications of the Astronomical Society of the Pacific**, [S.l.], v.123, n.901, p.366–380, Mar. 2011.

ZAHARIA, M. et al. Improving MapReduce performance in heterogeneous environments. In: USENIX CONFERENCE ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 8., Berkeley, CA, USA. **Proceedings...** USENIX Association, 2008. p.29–42. (OSDI'08).

ZHENG, Q. Improving MapReduce fault tolerance in the cloud. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING, WORKSHOPS AND PHD FORUM (IPDPSW), 2010. **Proceedings...** IEEE, 2010. p.1–6.

ZHU, H.; CHEN, H. Adaptive Failure Detection via Heartbeat under Hadoop. In: SERVICES COMPUTING CONFERENCE (APSCC), 2011 IEEE ASIA-PACIFIC. **Proceedings...** IEEE, 2011. p.231–238.