Providing fault tolerance and scalability of the MapReduce JobTracker using the Infinispan platform

Strahinja Lažetić

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Luís Eduardo Teixeira Rodrigues
Supervisor: Prof. João Coelho Garcia
Member of the Committee: Prof. Johan Montelius

September 2013
Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. João Coelho Garcia, for giving me this unique opportunity to work under his supervision, and for providing constant motivation, encouragement and clear and creative discussions.

I would also like to thank my classmates from the EMDC course for making me previous two years unforgettable.

Last, but not the least, I would like to thank my wife Marija, my parents Nebojša and Gordana and my brother Miloš for great support and understanding they provided me during my studies.

Lisbon, September 2013
Strahinja Lažetić
European Master in Distributed Computing, EMDC

This thesis is a part of the curricula of the European Master in Distributed Computing (EMDC), a joint master's program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Commission via the Erasmus Mundus program.

My track in this program has been as follows:
First and second semester of studies: IST
Third semester of studies: KTH
Fourth semester of studies (thesis): IST
For my wife Marija.
Abstract

The Apache Hadoop MapReduce is a distributed framework for processing large amounts of data in parallel on a cluster of computers. As many large scale processing distributed systems, MapReduces uses a master-slave architecture, with the JobTracker serving as a central master component of the system and multiple slave workers called TaskTrackers. The JobTracker runs on a single physical node and represents a single point of failure of the system; when it fails the systems looses all of its running jobs and becomes unavailable to the clients. In addition, a single physical node represents a constraint to the system’s scalability.

This thesis makes a contribution to both previously explained problems. We design and implement a fully functional solution for the JobTracker fault tolerance. We create a simple and efficient solution for providing the scalability of the JobTracker. As a support for providing both solutions we use the Infinispan data grid and distributed processing framework. We perform different tests in order to evaluate our solutions and discuss the obtained results.

Keywords

Fault tolerance, Scalability, JobTracker, Infinispan, Replication, Distributed execution
Resumo

O Hadoop MapReduce da Apache Foundation é uma framework para processar grandes quantidades de dados em paralelo num cluster de computadores. Tal como muitos sistemas de processamento de grande escala, o MapReduce usa uma arquitetura mestre-escravo, servindo o JobTracker de componente mestre central do sistema e múltiplos nós escravos chamados Task-Trackers. O JobTracker executa-se num único nó físico e constitui um ponto único de falha do sistema: quando falha, o sistema perde todos os trabalhos em execução e fica indisponível para os clientes. Adicionalmente, um único nó físico representa uma limitação de escalabilidade do sistema.

Esta tese apresenta contribuições para a resolução dos problemas referidos. Foi desenhada e implementada uma solução completamente funcional para a tolerância a falhas do JobTracker. Foi criada uma solução simples e eficiente para a escalabilidade do JobTracker. Ambas as soluções apoiam-se na framework de processamento e armazenamento distribuídos Infinispan. Apresentam-se vários testes avaliativos da solução e os seus resultados são discutidos.

Palavras-chave

Tolerância a falhas, Escalabilidade, MapReduce, Infinispan, Replicação, Execução distribuída
# Contents

1 Introduction ................................. 1
   1.1 Motivation .................................. 1
   1.2 Objectives .................................. 3
   1.3 Proposed Solution .......................... 3
   1.4 Contributions / Results .................. 3
   1.5 Thesis Overview ............................ 4

2 Related Work ................................. 5
   2.1 Introduction ................................ 5
   2.2 Motivation ................................ 5
      2.2.1 Distributed computing issues ............ 5
      2.2.2 Apache Hadoop ............................ 6
   2.3 Hadoop fault tolerance and scalability .... 9
      2.3.1 Community work .......................... 9
         2.3.1.1 Jira issues .......................... 9
         2.3.1.2 YARN ................................ 11
         2.3.1.3 Facebook Corona ..................... 12
         2.3.1.4 HDFS NameNode ...................... 13
      2.3.2 Scientific work .......................... 13
      2.3.3 Commercial solutions .................... 16
2.4 Overview of the related work ........................................ 16
2.5 Data storage systems ............................................. 17
    2.5.1 Infinispan .................................................... 18
2.6 Conclusion ......................................................... 21

3 Architecture .................................................. 23
    3.1 Introduction .................................................. 23
    3.2 System Requirements .......................................... 23
    3.3 Master Server in Distributed Systems .......................... 24
    3.4 JobTracker Fault Tolerance .................................... 27
    3.5 JobTracker Scalability ......................................... 31
        3.5.1 TaskTracker Distribution .................................. 32
        3.5.2 JobClient Load Balancing ................................. 33
    3.6 Conclusion ....................................................... 34

4 Implementation ............................................... 37
    4.1 Introduction .................................................. 37
    4.2 Source Code Versions ......................................... 38
    4.3 JobTracker Fault Tolerance .................................... 38
        4.3.1 Saving the Internal State ................................. 39
        4.3.2 Stored State .............................................. 39
        4.3.3 Serialization Optimization ............................... 41
        4.3.4 Cache Key Formation ...................................... 42
        4.3.5 Transactions and Locking ................................. 43
        4.3.6 Asynchronous Replication ................................. 44
        4.3.7 Correctness of the Operations in Case of Failure ..... 44

xii
4.3.8 Running the JobTracker on the Infinispan Distributed Execution Framework (DEF) .................................................. 46
4.3.9 JobTracker Node Election Policy .................................................. 47
4.3.10 JobClient and TaskTracker Re-connection to the JobTracker ........ 48
4.3.11 Infinispan Configuration .......................................................... 48

4.4 Scalability ............................................................................. 49
4.4.1 Running multiple JobTrackers in the Cluster .............................. 50
4.4.2 TaskTrackers Assignment and Load Balancing ......................... 50
4.4.3 Consistency Issues ............................................................... 51

4.5 Infinispan Data Structures ....................................................... 52

4.6 Hadoop Configuration ............................................................. 53

4.7 Changes to the Hadoop Execution Scripts .................................. 53

4.8 Infinispan Improvements .......................................................... 54

4.9 Conclusion ........................................................................... 55

5 Evaluation ............................................................................. 57

5.1 Introduction ........................................................................ 57

5.2 Benchmarking Tools ............................................................ 57
5.2.1 HiBench Benchmark Suite ...................................................... 57
5.2.2 YourKit Java Profiler .......................................................... 58

5.3 Testing Environment and Settings ......................................... 59

5.4 JobTracker Fault Tolerance Efficiency ..................................... 60

5.5 System Performance ............................................................... 63
5.5.1 State saving Overhead ......................................................... 63
5.5.1.1 Single Job Variations ....................................................... 63
List of Figures

2.1 Data and operational flow of a MapReduce job .......................... 8

3.1 Main architecture components of the solution ............................. 28

3.2 JobTracker failover .......................................................... 30

3.3 JobTracker scalability overview ........................................... 31

3.4 JC load balancing and TT re-assignment .................................. 34

4.1 Mapping memory data space into persistent storage space ............. 42

5.1 Single job execution time comparison .................................... 65

5.2 Multiple parallel jobs execution time .................................... 67

5.3 Memory consumption over the time for varying number of parallel jobs.. 68

5.4 Maximum memory consumption for varying number of parallel jobs .... 69

5.5 Performance comparison between scalable solution and original implementation. 71
List of Tables

2.1 *Comparison of the current solutions for Hadoop fault tolerance* .................. 17

4.1 *Cache key formation example* ................................................................. 43

4.2 *Hadoop configuration* ............................................................................. 54

5.1 *Cluster nodes characteristics* ................................................................. 59

5.2 *JobTracker failover time distribution* ...................................................... 61

5.3 *Testing applications settings* ................................................................. 64

5.4 *Summary of the jobs’ executions time and overhead* .............................. 66
Widespread adoption of the Internet and computer networks in general has led to the development of a new class of software systems that can run distributed on multiple hardware nodes. Nowadays, distributed systems cover a wide range of systems from the most famous World Wide Web to large scale processing frameworks (1)(2).

One of the most complex challenges in the design of a large scale processing distributed system is how to provide efficient coordination and communication between the system server nodes. A quite common approach supposes the use of a master - slave architecture (3)(4).

Although master servers can have diverse roles in different systems, they are usually centralized and run on a single physical node, thus representing a single point of failure. The implications of a possible crash of the master are in many cases severe. Another problem connected to a single node master server is the issue of scalability. In some usage scenarios, the master can have a too demanding task in terms of processing and memory consumption and thus can reach its operational threshold. This can slow down the system or even, in some cases, cause total unavailability of the master server and consequently of the whole system. These two issues impose a new task on the design of a distributed system - providing high availability and scalability of the master server.

In this thesis we address the fault tolerance and scalability issues of the master node of a specific distributed system and propose a solution that can be applied to all other systems with the master - slave architecture.

1.1 Motivation

The Apache Hadoop MapReduce is a distributed framework for processing large amounts of data in parallel on a cluster of computers (1). The main components of the framework are a single master node of the system, the JobTracker, and multiple slaves, one per cluster-node, called
TaskTrackers. The JobTracker receives client jobs to be executed on the cluster, delegates the jobs’ tasks to the TaskTrackers, monitors them and restarts in case of a failure. The TaskTrackers are the workers who process the jobs’ tasks, occasionally report their progress to the JobTracker and ultimately return the result of the jobs’ execution to the JobTracker. Although MapReduce executes jobs in a reliable, fault-tolerant manner, the JobTracker, which represents the central authority of the system, runs on a single physical node and represents a single point of failure of the system. In case of JobTracker failure, all ongoing jobs are lost, the MapReduce system is brought into an undefined state and cannot continue normal execution. Moreover, in the current implementation, an administrator of the system must detect the failure and manually restart the JobTracker and re-submit all of its previously running jobs. This can be a serious problem and it is unacceptable for long-running, time-critical jobs which must be completed within a specific time frame.

In addition to the fault intolerance, a less critical problem involves the unavailability of the system, making clients unable to submit new jobs until the JobTracker starts again. Although there is a vast amount of work on the topic of solving the JobTracker fault tolerance and high availability issues, there is no solution reliable and efficient enough that could be adopted by the Apache Hadoop community.

The JBoss Infinispan is a transactional in-memory key/value NoSQL data store and data grid [5]. The data is saved in memory and could be asynchronously replicated to a configurable number of nodes, thus making the Infinispan a very fast system for storing highly available data. Operations on the data can be processed reliably using the Infinispan transaction system. Infinispan supports distribution of data amongst the nodes, which provides high scalability of the system in a situation where the amount of data rapidly grows. Since it is written in Java and its core is an extension of the Java Map interface, Java programs do not require additional drivers or other mediators to interact with it, which makes communication fast and simple. Infinispan also offers a distributed execution framework which can run a distributed task on the Infinispan cluster of nodes and provides the ability to detect task failure and initiate its failover. The features listed above make the Infinispan suitable as a support for the JobTracker fault tolerance implementation.
1.2 Objectives

The main goal of this thesis is to present an easy and efficient solution for solving the issue of the MapReduce JobTracker fault tolerance, without significant impact on the system performance. The results of the system performance benchmarking are shown in the Evaluation chapter. Another two additional goals, evolved from the central one, are:

- Providing a solution for the JobTracker scalability issue with simple load balancing
- Improving the Infinispan Distributed Execution Framework to completely support a custom failover of the running distributed tasks

1.3 Proposed Solution

In this thesis we propose the Infinispan data grid and the Distributed Execution Framework for providing fault tolerance and scalability for the JobTracker. Since it is a very fast, scalable and highly available data storage, Infinispan can be used by the JobTracker to persist its internal state and reload it in case of failure, upon a restart.

We suggest running the JobTracker as a distributed task on the Infinispan Distributed Execution Framework, which will provide automatic failover of a failed JobTracker, out of the box. This would remove the need for running a spare JobTracker replica on an additional piece of hardware, coordination between the two JobTrackers and an implementation of a failure detector. High availability and data consistency of Infinispan enables us to use it as a coordination point for Hadoop cluster components, when providing the JobTracker scalability by running multiple JobTracker instances in the cluster.

1.4 Contributions / Results

The main contributions of the thesis can be summarized as follows:

- A fully functional fault tolerant implementation of the MapReduce JobTracker was provided in an efficient way, with no significant impact on the system performance. The JobTracker failover is automatic with a minimum of JobTracker downtime and no data/jobs
loss. The solution can tolerate multiple failures without the need for an administrator’s intervention.

- A simple and efficient solution for providing JobTracker scalability was implemented. Multiple JobTrackers can run simultaneously in the cluster, while load balancing components distribute jobs and TaskTrackers amongst the running JobTrackers.

- An improvement has been made on the Infinispan Distributed Execution Framework to completely support a custom failover policy for the running tasks.

- Experimental evaluation of the JobTracker fault tolerance and scalability implementation was performed, showing the main benefits and trade-offs.

1.5 Thesis Overview

The rest of the document is organized as follows: in the next chapter we give more details about the Hadoop MapReduce framework and its main issues and the Infinispan and its Distributed Execution Framework. In the Hadoop fault tolerance and scalability section of the chapter we give an overview of the research currently going on the topic of fault tolerance and scalability of the Hadoop ecosystem with an accent on the MapReduce framework. In the Chapter 3 a basic architectural design of the proposed solution is presented. In the Implementation chapter of the thesis, we describe the implementation details of the solution and give a more exhaustive explanation of the processes. Finally, in the Evaluation chapter we present the tests performed on our solution and the results obtained.
2.1 Introduction

In this chapter we introduce the fundamental concepts and main problems in the area of large scale distributed computing that will help to understand the problem addressed in this thesis and the proposed solution. The Apache Hadoop is an influential representative of large scale distributed systems that includes both distributed storage and a distributed computational framework. We give a detailed description of the Apache Hadoop basics and elaborate on the topic of fault tolerance and scalability issues of its master components, JobTracker and NameNode, in order to explain the motivation and justify the need for this thesis’s work. We present the state of the art in this area by discussing current works that deal with these two issues and put a special accent on the MapReduce JobTracker fault tolerance and scalability problems. Since the Hadoop NameNode suffers from similar problems as the JobTracker, we also look into current courses of action for providing the NameNode fault tolerance which can be applied to the JobTracker as well. From the related work we will see that the JobTracker and NameNode fault tolerance is mainly based on preserving their operational state, either by replicating it to the other server components or storing it to a persistent storage. Thus, we continue this chapter by giving an overview of an influential open source storage system that can be utilized for providing the fault tolerance and scalability of the Hadoop components.

2.2 Motivation

2.2.1 Distributed computing issues

In most general terms we can define a distributed system as one that consists of heterogeneous hardware and software components which are located on networked computers and coordinate their actions only by passing messages [6]. It covers a wide range of systems, from the World
Wide Web, that presupposes a request / response client - server communication between the two network nodes, to large-scale processing frameworks \(^1\)\(^2\) that require complex inter servers communication and coordination.

The development of a distributed computing system leads to many design challenges and issues. Some of them that have to be taken into account are the security of the system, fault tolerance, scalability, concurrency of the system components and system transparency. Beside client - server communication, which usually imposes simpler requirements to the design, a more complex challenge in large-scale processing distributed systems is how to provide efficient coordination and communication between the server nodes. Beside pure decentralized distributed systems such as P2P and DHT \(^7\), where all servers are equal, the use of a master - slave architecture is a quite common solution \(^3\)\(^4\).

Although master servers can have diverse roles in different systems, they are usually centralized and run on a single physical node, thus representing a single point of failure. Depending on the master role in the system, the implications of its crash can be of different severity. In some cases, when the master server fails, the whole system becomes unavailable to the clients but the current work is preserved \(^8\). On the other hand, in some other systems, the executing operations \(^1\) or even the persisted data are lost. Another issue that results from the fact the master server usually runs on a single node is the scalability issue. In some usage scenarios, the master can have a too demanding task in terms of processing and memory consumption and thus can reach its operational threshold (e.g. processor congestion, memory overload). This can slow down the system or even, in some cases, cause total unavailability of the master server and consequently the whole system. These two issues impose a new task on the design of a distributed system - providing high availability and scalability of the master server.

### 2.2.2 Apache Hadoop

The Apache Hadoop is an influential representative of large scale distributed systems that includes both distributed storage and distributed processing framework which allow creating reliable and scalable distributed applications. It is an open source, top level Apache project \(^9\). Hadoop is designed to scale to thousands of nodes and each node can be used as a data storage as well as a computational node. Two basic components of the Hadoop framework are an open source implementation of a distributed file system, the HDFS, and a distributed processing
2.2. MOTIVATION

HDFS is a Hadoop distributed file system which is used as a distributed storage in many other Hadoop projects. An HDFS cluster has a typical master / slave architecture and consists of a single master NameNode, and multiple slaves DataNodes. The NameNode manages file system metadata and coordinates clients access to the HDFS. A DataNode holds the actual data and manages the storage located on its physical node. The data are distributed and replicated across the several DataNodes in the cluster. Each HDFS file is split into multiple blocks which are spread across the cluster DataNodes, and the NameNode determines the mapping of these blocks to the actual DataNodes. Using data replication and distribution, HDFS provides high fault tolerance and scalability to the system. The data are highly available and the system can tolerate multiple sequential failures of the DataNodes (the degree depends on the replication factor). Nevertheless, the availability of the whole system depends on the single master NameNode, and as such it represents a single point of failure. In a situation where the NameNode crashes, the cluster becomes unavailable and after the NameNode restarts all the cluster nodes need to be restarted as well.

The Apache Hadoop MapReduce is a distributed framework for processing large amount of data in parallel on a cluster of computers. It is an implementation of the popular programming map-reduce model and represents an open source version of Google’s proprietary technology \(^{[10]}\). The framework architecture was also developed in a master - slave fashion and consists of a single master JobTracker and multiple slaves, one per cluster-node, called TaskTrackers. The JobTracker is the main authority in the system. It receives user jobs, delegates their execution to TaskTrackers, monitors them and restarts in case of a failure. The TaskTrackers are the workers who process the jobs, occasionally report their progress to the JobTracker and ultimately return the status of the jobs' execution to the JobTracker. MapReduce uses HDFS to write the initial and final results of the jobs execution. The detailed data and operational flow of a MapReduce job are presented in the Figure \[2.1\].

For each of the user jobs one instance of the MapReduce JobClient component is created. JobClient retrieves a unique identifier for the new job from the JobTracker and then generates the input splits for the job execution and writes these splits together with the job resources to the HDFS. Finally, Job client submits the job to the JobTracker \(^{[11]}\). The JobTracker receives the submitted job, reads its input splits from the HDFS and creates tasks for the job. For
CHAPTER 2. RELATED WORK

Figure 2.1: Data and operational flow of a MapReduce job

each split read from HDFS, the JobTracker creates one map task and several reduce tasks, depending on the job configuration parameters. The JobTracker uses a FIFO priority queue for scheduling incoming jobs to TaskTrackers. TaskTrackers heartbeat to the JobTracker in fixed time intervals and report the current status of the executing tasks (if any) and the status of its internal resources. As a response, the JobTracker sends job tasks to the TaskTracker. When it gets a task from a new job, the TaskTracker copies the job's resources and jar file from the HDFS to the local node. For each task, TaskTracker runs a new child process (starts a new JVM) in order to isolate possible task failures, that can be caused by a bug in the user program, from the rest of the framework. The TaskTracker then reports the status of the executing tasks to the JobTracker which then reports back to the JobClient, which constantly prints the progress of job execution to the user. The final execution results of the job are written to the HDFS as well. The JobTracker also manages the tasks' lifecycle and restarts them in case of a failure. Moreover, the JobTracker is responsible of detecting TaskTracker failures and re-sending their tasks to another TaskTracker.

MapReduce processes jobs in a reliable, fault-tolerant manner, since it provides a fault tolerant mechanism for TaskTracker nodes. When a TaskTracker fails, the JobTracker reschedules all of its running tasks to another TaskTracker, which re-executes them from the beginning. On the other hand, the JobTracker, which represents the central authority of the system, runs on a single physical node and represents a single point of failure of the system. In case the JobTracker fails, all ongoing jobs are lost, the MapReduce system is brought into an undefined state and cannot continue normal execution. Moreover, in the current implementation, an administrator of the system must detect the failure and manually restart the JobTracker and re-submit all its
previously running jobs. This can be a serious problem and it is unacceptable for long-running, time-critical jobs which must be completed within a specific time frame. In addition to the fault intolerance, the problem involves unavailability of the system, making clients unable to submit new jobs until the JobTracker starts again. Another issue the MapReduce framework imposes is the scalability of the system. While the scalability of the workers can be achieved just by adding new TaskTracker node to the system, there is only one master JobTracker node. This raises the question of the number of jobs it can handle in parallel.

Although Apache Hadoop represents an open source project with a very large community and it is widely used in both educational institutions and production environments of big companies, there is still no efficient solution for the problems of NameNode and JobTracker fault tolerance and scalability. This opinion is also supported by the fact that the Apache community still has not incorporated any of the proposed solutions into their official releases. These two problems represent a serious defect and can be a constraint to further expansion of the framework and its adoption by new interested parties. Bearing that in mind, further work on these problems could be of great importance.

2.3 Hadoop fault tolerance and scalability

There are many solutions and ongoing work on the topic of Hadoop fault tolerance and scalability and we can divide them into several categories: work of the Hadoop community, researchers’ work and commercial solutions.

2.3.1 Community work

The Hadoop community still has not incorporated an efficient solution for the MapReduce fault tolerance into their project. Nevertheless, in several Hadoop MapReduce distributions there are simple solutions that cover this problem to some extent.

2.3.1.1 Jira issues

The *Jira Hadoop-3245* issue was implemented in the Hadoop 0.19 version. This approach uses MapReduce JobHistory to recover running jobs upon a JobTracker failure. The
JobHistory is a regular JobTracker component which logs changes of the running jobs and tasks to the file system. For each job, the JobHistory makes a separate plain text file, where each line is of the format "type (key=value)" \[14\]. The type represents the component for which the status is logged and can be a Job, Task, MapAttempt or ReduceAttempt, while the key and the value represent the component’s properties and current values, respectively. JobHistory provides a state preserving restart of the JobTracker, since the JobTracker state before the failure can be reconstructed, using the JobHistory files. Still, in case of failure, the JobTracker has to be restarted manually. Moreover, the re-start time could be unpredictably long due to the parsing of the JobHistory file and possible rollbacks of the new JobTracker state, since the recording of some changes can be lost in the moment of a failure. The biggest issue with the Jira Hadoop-3245 implementation is the fact that it has proved to be unstable, and the approach was abandoned in the following Hadoop releases.

A design for a JobTracker automatic failover is proposed in the Jira MapReduce-225 issue \[15\] as an extension to the previously mentioned work. It considers running multiple redundant JobTrackers in a system which forms active - standby relations and uses a passive master - slave replication model. At one moment, there is only one master JobTracker and JobClients and TaskTrackers communicate only with the master. The solution uses the JGroups library for members management, leader election and system health checking. There is one additional component per each JobTracker, called FTManager, which uses JGroups for providing JobTracker fault tolerance operations. Upon system start, FTManagers starts the leader election algorithm and chooses a new master JobTracker. The master announces itself (its IP address) on the HDFS and TaskTrackers pick it up and establish connections to the master JobTracker. During execution of the jobs, every operation that affects the JobTracker state also triggers a state update to the slaves. In case of a failure of the master, FTManagers detects the failure, starts a new leader election and repeats the previous steps. The main disadvantage of this solution represents running multiple slave JobTrackers in the system. They occupy additional hardware nodes (intended to be between 2 and 4) and add an overhead and complexity to the system. Specifically, running multiple slaves in the system requires additional coordination during state replication in normal operation and a leader election algorithm in case the master fails.
2.3. HADOOP FAULT TOLERANCE AND SCALABILITY

2.3.1.2 YARN

Although Yahoo reported that they managed to run 5000 MapReduce jobs and connect 4000 TaskTrackers to a single JobTracker (16), the Hadoop community has already started working on the MapReduce scalability in the new Hadoop YARN project. YARN, or MapReduce 2.0, represents a new version of the MapReduce framework which has undergone radical changes. The main innovation represents the separation of the responsibilities for the main functions of the JobTracker, resources and jobs (called "applications" in YARN) management, from a single to several components. In YARN, two main components of the system are the ResourceManager which represents a master node of the system, and a per-node slave, the NodeManager (17). The ResourceManager is the central authority in the system and it is responsible for managing all the resources of the system and distributing them among the running applications, as well as managing the lifecycle of the applications. The ResourceManager contains three main components: ApplicationsManager, Scheduler and ResourceTracker. The ApplicationsManager is responsible for accepting the submission of a new application from a client and managing the application’s execution and lifecycle. For every submitted application, it starts a separate process, the application’s specific ApplicationMaster component, on a particular computational node. The ApplicationMaster executes the application, reports back its execution progress and negotiates new resources with ResourceManager based on the application’s requirements. The ApplicationsManager is also responsible for managing the lifecycle of the ApplicationMaster (start, stop and restart on failure). The ResourceTracker and the Scheduler work together in order to allocate and schedule resources of the computational nodes to the running applications based on their resource requirements. The resources are allocated in the form of abstract resource containers which represent a resource bundle of the specific computational node: a portion of its CPU power, memory, disk space and network bandwidth. In particular, resource requirements are computed by an ApplicationMaster and sent back to the Scheduler. The ApplicationMaster can ask for several resource containers running on different physical nodes. A NodeManager is responsible for managing resource containers on its physical node and reporting back the resource consumption and container status to the ResourceManager.

In the YARN design, several issues from the previous MapReduce version have been solved. The crucial one was the separation of resource management and job life-cycle management into separate components (18) and starting one managing daemon for each application, which allows
CHAPTER 2. RELATED WORK

scaling the framework. Still, the ResourceManager, the main authority of the system, similarly to the JobTracker in the MapReduce, runs on a single physical node and represents a single point of failure of the system. In case of its failure, all the running applications also fail and cannot continue their execution, even if the ResourceManager recovers afterwards.

Several works on these issues already exist in the Hadoop community. A very simple solution which already comes with the recent YARN releases considers a manual restart of the ResourceManager. Moreover, the implementation does not preserve the work that was ongoing when the ResourceManager fails, but it automatically restarts all the previously running applications, upon the ResourceManager recovery [19]. The solution uses Zookeeper to store the information about the running applications which is updated each time a new application is submitted. This is a quite limited solution for the ResourceManager fault tolerance, and unacceptable in many cases, but it was intended to be a starting point for a more advanced work-preserving restart.

The work on YARN represents a very ambitious and novel MapReduce implementation and the Hadoop community has big expectations from it. Nevertheless, this framework is still not mature enough to be released as a Hadoop stable version (currently only alpha) and many users have reported different types of bugs and shortcomings. As such, it is still rarely used in production environments.

2.3.1.3 Facebook Corona

Facebook Corona is another solution which had in mind JobTracker downtime and scalability issues [20]. Corona is a scheduling framework that, similarly to YARN, separates JobTracker basic functions, cluster resource management and job coordination. A new node, called ClusterManager, is developed with the main role to coordinate resources in the cluster. In Corona, one JobTracker process is created for each running job and coordinates only the job’s execution. Unlike the community’s Hadoop MapReduce, Corona utilizes the push-model instead of the pull-model, in the sense that the ClusterManager grants resources to a JobTracker, and each JobTracker pushes job tasks to its TaskTrackers. There is no periodical heart beating between JobTrackers and TaskTrackers, thus the scheduling latency is removed. Creating one JobTracker per job represents a step towards high availability of the system, since the failure of a single JobTracker causes only the loss of its job and does not affect other, already running, jobs in the cluster. However, the initial problem of fault tolerance is now transferred to the ClusterManager.
2.3. HADOOP FAULT TOLERANCE AND SCALABILITY

which, again, runs on a single physical node, thus presenting a single point of failure of the system.

2.3.1.4 HDFS NameNode

There are several solutions for solving the problem of the HDFS NameNode high availability, and some of them have been integrated into production Hadoop distributions. Apache Hadoop community offers several of them. The SecondaryNameNode (21) was introduced in the 0.18 version. Although its name can indicate that, it does not represent a NameNode replica, in the sense that DataNodes cannot connect to it and it cannot replace the NameNode in case of its failure. The only purpose of the SecondaryNameNode is to occasionally make checkpoints of the NameNode state, merge it to its current file system image and also send it back to the NameNode. In case the NameNode fails, the new NameNode can read the previous state from the stored file system image located either on the previous NameNode physical node or in case the whole node crashes, from the SecondaryNameNode physical node. In both cases, the new NameNode must be started manually. Although the SecondaryNameNode is still present in the Hadoop distributions, it has showed many flaws and proved to be unreliable.

2.3.2 Scientific work

Several scientific papers were released on the topic of MapReduce fault tolerance. In *Evolving fault-tolerance in Hadoop with robust auto-recovering JobTracker* (22), the authors propose an auto-recovery system against the fail-stop of the JobTracker which does not require additional hardware. The running jobs can continue to execute, after the JobTracker restart, with a small rollback.

The recovery mechanism is based on the JobTracker state snapshots. When every N tasks completes (N can be configurable), the JobTracker makes a snapshot of its current state and stores it to the HDFS. The state stored includes most of the JobTracker dynamic data structures. A failure of the JobTracker is automatically detected by a TaskTracker using the regular heartbeat mechanism. After that, a specific TaskTracker, chosen as a top TaskTracker according to the configured candidate list, terminates itself and starts a new JobTracker process on its node. The candidate list is stored on HDFS, and thus it is available to the other TaskTrackers which also use
it to discover the address of the new JobTracker node. The new JobTracker loads the snapshot from the HDFS and recovers the state of the previous JobTracker. Since the checkpoints are done periodically, it is possible that a part of the JobTracker state is not persisted, in the moment when it fails. To overcome this, upon a restart, JobTracker queries its TaskTrackers for the status of their tasks, in order to update its state. In addition, the system considers the execution of the tasks during the failure time as a problem, and requires a synchronization on these changes as well. After one of the TaskTrackers terminates itself, the JobTracker re-assigns its tasks to some other TaskTracker. If the communication breakdown between a TaskTracker and the JobTracker was caused by a network partitioning, the start of a new JobTracker would still proceed, after which the new JobTracker would have to detect and terminate the old one in order to put the system into a consistent state.

The authors evaluated system performances on the Hadoop WordCount sample program in a case when no JobTracker failure occurred. The checkpointing time overhead obtained was less than 4.3% when checkpointing on each 10 completed tasks, while the overhead increased when using higher frequency of checkpointing. The size of the snapshot is proportional to the input data size, since the number of tasks increases with the data size.

The presented system provides automatic detection of a JobTracker failure and its automatic failover. In addition, it tolerates multiple failures without requiring additional hardware to restart the JobTracker. On the other hand, the main overhead of the system represents the state checkpointing to the HDFS, which requires serialization and de-serialization of the data. A fair amount of time is also needed to start a new JobTracker after failure, and it is caused by the synchronization of the missing state with TaskTrackers and additional re-execution of the terminated TaskTracker tasks. The system also supposes redundant JobTracker recoveries each time a misdetection caused by a network partitioning occurs and in these cases it also requires additional time for the new JobTracker intervention.

In *Hadoop High Availability through Metadata replication* ([23]) authors addressed both problems of NameNode and JobTracker high availability by removing the single points of failure of the Hadoop system. The general architecture is similar for both components, thus the authors put the main accent on the NameNode fault tolerance. The solution supports two types of topology architectures of Hadoop nodes: active / standby and primary / slave architecture. Fault tolerance is based on the replication of the crucial metadata between these nodes.
2.3. **HADOOP FAULT TOLERANCE AND SCALABILITY**

The metadata consist of the most important management information required for NameNode and JobTracker failover. In case of the NameNode, the metadata are the initial metadata, which contain the software version information and the NameNode initial file system image, and the runtime metadata, which contain all write operations submitted by clients and the file system leases state. Write operations metadata are constantly replicated to the standby / slave nodes which eventually apply them to their file system image, thus keeping the state close to the master NameNode file system. Since the data in HDFS are accessed only by a single writer, the leases state metadata are important for the failover process to guarantee operations consistency after system recovery.

Upon system initialization, the slave nodes register to the master node and perform initial metadata synchronization. Each node sends its IP address to the master, which then broadcasts the slaves’ IP address table to all slave nodes, so they can discover each other. During the normal operation, the master node replicates metadata to all slave nodes and slave nodes use the regular heartbeat mechanism to keep track of the master node availability. In case a slave node loses contact with the master, it initiates a new leader election. The leader election is mainly based on assigning a unique identifier and ordering the nodes using the identifier. After the new master is elected, it invokes a system command for IP address re-configuration to switch its IP address to the one of the previous master node. In that way other nodes can continue communicating with the primary node without starting a discovery process. The remaining tasks for full recovery consist of reconstructing the data blocks mapping information. That is achieved using block list sent by DataNodes, in case of the NameNode or using MapReduce job information loaded from JobHistory, in case of the JobTracker.

The solution does not take into account possible network partitioning and the possibility that the master is still alive during the master fault detection and the leader election process. The authors also conclude from the experiments that a vast amount of failover time goes to the leader election phase, since it can contain an unpredictable number of conflicts between slave nodes. Next, the cost of all metadata processing shows that the total time doubles comparing to the normal process, which is mostly caused by the intensive network communication between master and slave nodes. Finally, the overhead also includes the necessity of IP addresses reconfiguration in order to switch to a new master node.
2.3.3 Commercial solutions

There are a few commercial distributions of Hadoop MapReduce which cover the JobTracker fault tolerance in some way. MapR is a popular and widely used commercial Hadoop MapReduce distribution developed by the MapR Technologies company \[24\]. MapR covers both NameNode and JobTracker high availability. When the JobTracker fails, MapR automatically restarts the JobTracker on some other candidate node and TaskTrackers reconnect to the new JobTracker. The running tasks are not affected by the failure and they can continue with their execution during the failover time. Candidate nodes are chosen by an administrator, before the JobTracker is started, and any of them can serve as a possible destination of the restarted JobTracker. The process does not require further administrator intervention and it enables the JobTracker to tolerate simultaneous failures.

In its Hadoop distribution, Facebook implemented an automatic failover of the HDFS NameNode. They created a wrapper around the ordinary NameNode, called AvatarNode, and deployed two instances of it to form an active-passive-hot-standby pair. The Standby AvatarNode represents a replica of the primary AvatarNode. Both of the nodes communicate with all the DataNodes, thus the standby node always has the most recent information of the DataNodes blocks locations. Two AvatarNodes synchronize their states through the NFS \[25\], where the primary writes the current file system image to the transaction log and the standby reads it and applies to its namespace. In this way, the standby AvatarNode keeps its state as close as possible to the primary one and it can become active in less than a minute, in case the primary AvatarNode fails. For the coordination of the system the Zookeeper coordination service is used. The Zookeeper coordination service \[26\] is in charge of detecting a failure and performing a failover of the master. The DataNodes and the clients use Zookeeper to determine which AvatarNode is the master in a particular moment of time. The Hadoop community made an open source implementation based on the Facebook Avatar Node in the Hadoop 2.0.0 version and called it HDFS high availability.

2.4 Overview of the related work

In the Table 2.1 we summarize the basic features of the presented solutions. We divided the features by the categories we defined in the Architecture chapter, as requirements for our
2.5 Data storage systems

From the presented related work we can conclude that the JobTracker and NameNode fault tolerance is mainly based on preserving their operational state, either by replicating it to the other server nodes or storing it to a persistent storage. In our solution we chose the second solution.

From the presented table we can see that none of the solutions covers all of the requirements we identified as crucial for a fault tolerant and scalable solution. In addition, only two solutions offer scalability for Hadoop components but, on the other hand, do not cover fault tolerance at all. For fault tolerance, several solutions do not preserve the work that was executing before a failure, but require restarting the operations from scratch. In all solutions, the state is preserved either by replication to the slave servers or by checkpointing to some persistent storage. Only the commercial solution offers the stable fault tolerance for the JobTracker and the NameNode, provided that they do not support the scalability of these components.

In our solution we combine the previously explained approaches and cover as many of the basic requirements as possible for fault tolerance and scalability of the JobTracker.

<table>
<thead>
<tr>
<th>System / Fault tolerant issue</th>
<th>Work preserving failure</th>
<th>Automatic failover</th>
<th>State preserving technique</th>
<th>Performance impact</th>
<th>Scalability support</th>
<th>Status of the implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop-3245</td>
<td>Yes</td>
<td>No</td>
<td>Checkpointing to HDFS</td>
<td>Variable restart time</td>
<td>No</td>
<td>Unstable</td>
</tr>
<tr>
<td>MapReduce-225</td>
<td>Yes</td>
<td>Yes</td>
<td>Master - slave replication</td>
<td>Unknown</td>
<td>No</td>
<td>Concept</td>
</tr>
<tr>
<td>VARN</td>
<td>No</td>
<td>No</td>
<td>Checkpointing to Zookeeper</td>
<td>Unknown</td>
<td>Yes</td>
<td>Unstable</td>
</tr>
<tr>
<td>Corona</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Small</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>NameNode</td>
<td>Yes</td>
<td>No</td>
<td>Master - slave replication</td>
<td>Small</td>
<td>No</td>
<td>Stable</td>
</tr>
<tr>
<td>Robust auto-recovering JobTracker</td>
<td>Yes</td>
<td>Yes</td>
<td>Checkpointing to HDFS</td>
<td>Variable restart time, increased execution time</td>
<td>No</td>
<td>Concept</td>
</tr>
<tr>
<td>Metadata replication</td>
<td>Yes</td>
<td>Yes</td>
<td>Master - slave replication</td>
<td>Variable restart time, doubled execution time</td>
<td>No</td>
<td>Concept</td>
</tr>
<tr>
<td>MapR</td>
<td>Yes</td>
<td>Yes</td>
<td>Unknown</td>
<td>Small</td>
<td>No</td>
<td>Stable</td>
</tr>
<tr>
<td>AvatarNameNode</td>
<td>Yes</td>
<td>Yes</td>
<td>Master - slave replication</td>
<td>Small</td>
<td>No</td>
<td>Stable</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of the current solutions for Hadoop fault tolerance

2.5 Data storage systems

From the presented related work we can conclude that the JobTracker and NameNode fault tolerance is mainly based on preserving their operational state, either by replicating it to the other server nodes or storing it to a persistent storage. In our solution we chose the second
approach, i.e. storing the JobTracker state in a persistent storage, and we will justify this choice in the Architecture chapter.

Since the JobTracker state we want to preserve is crucial for the proper operations upon a JobTracker failover, we need a distributed storage that provides high availability and reliability of the data. The state size can grow over the time, when adding new jobs to the system, thus the storage should offer partitioning of the data amongst the cluster nodes. The checkpointing of the state can be frequent and the storage should provide fast operations and easy access to the data. Since we do not have complex relations among JobTracker data structures, relational databases such as (37) would be inappropriate solution for our problem. On the other hand, the big data stores (3) (38) contain too complex data models for simple operations, such as get, put and delete, that we need to perform on the data. In addition, we do not want any data transformation, but rather to store the JobTracker data structures as they are.

In-memory storages offer fast operations on the data and do not require data serialization (39). Key/value stores provide an easy access to the data by a specified key (40) (5). Thus, we continue this chapter by giving an overview of JBoss Infinispan, a very flexible and reliable distributed key/value in-memory data storage with a rich set of features that can be utilized for providing fault tolerance and scalability of the Hadoop components.

2.5.1 Infinispan

JBoss Infinispan is a transactional in-memory key/value NoSQL data store and data grid (5). Infinispan is written using the Java programming language and in its core it exposes a Cache interface, which represents the Java Map interface extended with features like remote access, replication and data distribution. Since it is a pure Java data storage implementation, it provides a fast, direct access to all Java user programs with no need for an additional driver library.

Infinispan uses the JGroups library (27) to handle the network communication. JGroups provides detection of the cluster nodes, handles the nodes group membership and enables reliable data transfer between the nodes.

Although Infinispan stores all the data in memory, the data intended to be saved must be serializable in order to be transferred over the network to the other cluster nodes.
uses JBoss Marshalling framework to serialize and de-serialize data. To be suitable for using JBoss Marshalling, the data objects must implement either Java Serializable interface or Java Externalizable interface which both require modification of the user's source code. In addition to these two methods, Infinispan allows users to define their own custom serializers, through its Externalizer interface implementation. As opposed to the previous two methods, Externalizer can be a separate class that controls how an object of a specific class is created from or written to a byte stream, thus not interfering with the existing source code.

Infinispan can be started either locally or as a data grid cluster (28). Local mode assumes running only one Cache instance on a local node and as such provides similar features to the ordinary Java Map. When started as a cluster, Infinispan runs one Cache instance on each physical node in the cluster and they communicate with each other using JGroups underlying protocols. Infinispan supports two main cluster modes. In the replication mode, entries added to a particular node will be replicated to all instances in the cluster and thus can be retrieved locally on any node. Replication can be configured to be synchronous and asynchronous. Asynchronous replication is faster, since the caller thread does not wait for the operation to be performed on all the nodes, but it can produce data inconsistency. Synchronous mode guarantees to the caller that when an operation returns, it is successfully executed on all the cluster nodes. Distribution mode provides high availability of the data without affecting the cluster scalability. Data are replicated to a configurable subset of all the nodes and they are also partitioned and distributed across the cluster. Distribution mode uses a consistent hashing algorithm to determine on which node the data should be located. Using this mode, Infinispan can scale linearly just by adding new servers to the cluster.

There are two modes of operation in Infinispan, peer-2-peer and client-server. Peer-2-peer is a simple running mode where a user program runs in the same process (same JVM) as an Infinispan Cache instance. Each instance of the user program starts a single Infinispan Cache instance and different instances discover each other and form a data grid. Using this mode, Infinispan is practically integrated into the user application and makes it cluster aware. In the client-server mode, the Infinispan instance is started on a separate server and user programs access it using a client connector. This mode is practical when the client code is not written in Java or there is an architectural requirement to have a separate data layer. In this mode, clients can use several server endpoints and protocols for the remote communication to the Infinispan.
CHAPTER 2. RELATED WORK

The server modules are: REST, Memcached, HotRod and WebSocket.

Infinispan provides a very flexible and rich transaction model. It exposes Java Transaction API (JTA) with a support for distributed transactions and a particular implementation of the Java TransactionManager can be configured. Infinispan transactions offer two locking modes: optimistic and pessimistic mode [29]. In the optimistic model, Infinispan tries to put locks for data on all nodes in the transaction prepare time, while pessimistic locking acquires cluster wide locks on each write operation and releases them only after the whole transaction completes. Obviously, the first approach leads to more transaction rollbacks, due to possible intermediary updates and it is not suitable for systems where there are many concurrent transactions accessing the same storage data. On the other hand, the pessimistic transactions are more costly since they require a RPC (remote procedure call) for each lock acquisition and thus reduce the throughput of the system. Still, they are more desirable when transaction rollbacks should be avoided.

Besides the data grid, the Infinispan also offers a distributed execution framework (DEF), which allows distributed tasks to run on the Infinispan cluster of nodes and also utilize existing data storage [30]. Each user task should implement the Infinispan DistributedCallable interface, which extends the Java Callable interface, and must be serializable. Instead of running on the local node only, a DistributedCallable can be started on any node in the cluster (user defined) or even on all cluster nodes. Each DistributedCallable can directly access the Infinispan Cache on its local node. This feature allows a user program to be split into multiple tasks and run in parallel on Infinispan grid nodes. Each node will execute its task and return the result to the calling node. Infinispan DEF also provides the ability to detect a failure of the running task and restart it on another node. For this purpose, DEF uses DistributedTaskFailoverPolicy interface which defines the node on which to restart the failed task. After starting a task, Infinispan DEF holds a remote reference to it and in case of failure it automatically restarts the task on some other node, using the information provided in the user defined failover policy implementation. The failure can be a crash of the distributed task process or the whole physical node where the task runs.

Infinispan flexibility and the rich set of features presented above lead us to the conclusion that it can be suitable as a support for the JobTracker fault tolerance and scalability implementation. We will elaborate this decision in more detail in the Architecture chapter.
2.6 Conclusion

In this chapter we introduced the relevant theoretical background for understanding the problem addressed in this thesis and gave a review of the work related to the problem which served as a starting point for the proposed solution. We started by shortly describing the main problem in the master-slave architecture in distributed computing. Next, we gave an overview of the Hadoop HDFS and MapReduce technologies and explained their main flaws, which served as a motivation for this thesis. We explored the most relevant research work on the topic of HDFS and JobTracker fault tolerance and scalability issues as well as existing community’s and commercial solutions. We also made an insight into the JBoss Infinispan distributed storage, which we proposed as a support for the solution for the described Hadoop problems. From the related work we can induce that all of the solutions for the HDFS NameNode and MapReduce JobTracker fault tolerance use some kind of master state preservation. Some of them replicate the state to the slave components while other save it to a persistent storage. The approaches for the master restart vary from a manual restart of the master to an automatic failover to another slave node. All of the presented solutions suffer from more or less severe flaws, and as such are not good enough to be incorporated into official Hadoop releases. The main solutions for scalability rely on dividing the framework resources and jobs lifecycle management into separate components. This always supposes having an additional component which takes over the former master’s role, thus transferring the existing problem to another component. We will elaborate in more details about the explained problems as well as the critics of the existing solutions at the beginning of the next chapter after which we will present our design for the fault tolerant and scalable JobTracker.
3.1 Introduction

In this chapter we present a design of the fault tolerant and scalable JobTracker. We start by defining the goal of the work and the requirements imposed on the design. Next, we discuss potential problems and possible solutions in the more general area of distributed systems and refer to the solutions presented in the Related Work chapter. After that, we concretize the discussion on the Hadoop MapReduce framework. Finally, we give an overview of the proposed design for JobTracker fault tolerance and scalability.

3.2 System Requirements

The goal of the thesis is to provide an efficient implementation of the fault tolerant and scalable MapReduce JobTracker, which will not significantly affect the system performances. From the defined goal, we identify the following requirements imposed on our solution:

- **Fault tolerance** - The solution must provide the JobTracker fault tolerance and ensure it continues functioning normally even when multiple simultaneous JobTracker failures occur, given that enough physical resources are provided.

- **Correctness** - In case the JobTracker restarts, the system must continue executing already running jobs correctly and produce correct final results.

- **Availability** - The system must be as available to the clients as possible in case the JobTracker fails. Specifically, the failover should be very fast and the system downtime should be minimal.

- **Automatic failover and transparency** - The failure of the JobTracker should be transparent to the end users and system administrators, and the system should provide auto-
matic crash detection and failover of the JobTracker to some other available node, without administrator intervention.

- **Scalability** - The system must allow JobTracker functionalities to scale, in case when the number of running jobs rapidly grows, given that enough physical resources are provided.

- **Performance** - The solution must provide fault tolerance of the JobTracker without significant impact on the system performance in terms of the total execution time of user jobs and memory occupied by additional framework components.

Having the ultimate goal and the listed system requirements in mind, we first discuss the possible approaches for achieving them in general area of distributed systems, after which we focus on the specific problem of MapReduce. Finally, we present the design of our solution.

### 3.3 Master Server in Distributed Systems

The main problem of the master-slave architecture in distributed systems is explained in the Related Work chapter. The problem of the fault tolerance of the master node can be approached in different ways. We saw that most of the related work supposes preserving the crucial state of the master server, which, in case of its failure, can be used by other system components to recover the system and continue with normal operation. The master’s state can be preserved in different ways and it is tightly related to the deployment scheme of the server components. From the presented work we can extract two different approaches for state preservation. In both cases, in the following text, when we use word "server" we consider the master management node of a system.

The first approach represents data replication between the server nodes in the system [15][23]. There are two most commonly defined types of replication [6]. In passive (or Primary - Backup) replication there is at any time a single primary replica server and one or more secondary replica servers called backups or slaves. Clients always communicate only with the primary replica to obtain the service. The primary receives requests from the clients, executes the operations and sends copies of the updated data to the backups. The backup servers periodically check if the primary is alive. If the primary fails, one of the backups is promoted to be the new primary, which creates the system overhead of electing and agreeing upon the new master.
In a modified version of this approach, clients can contact passive replicas for read operations only, thus relieving the load from the primary. In active replication, all replicas are identical and represent state machines. Clients multicast requests to all the replicas, which process it independently but identically and return the result to the clients. A crash of a replica does not have any impact on the system performance, since all of the other replicas can continue normally, and the clients can perform an additional check by comparing results obtained from all the replicas.

The second approach supposes checkpointing the master internal state into a persistent storage \((19)(25)(14)(22)\). The storage needs to be reliable, which requires some kind of data replication, and it needs to support data distribution in order to provide scalability. Since the storage is used as a central point for storing the data, we are flexible in the design of the components deployment. We can have a single master server, as in \((14)(22)\), which holds all of the system management data, in which case the new master would have to start from scratch upon the old master’s failure and load the management state from the storage. In the other case, there can be multiple masters, with a single active one and an arbitrary number of passive masters. In this case, when the active one fails, one of the passive masters can takeover and proclaim itself as a master, load the state from the storage and continue managing the system.

The scalability of the master component can be also achieved in different ways. Several works proposed starting one master component per unit of work (e.g. job, application) \((18)(20)\) and thus limiting the responsibility of the master to only managing the lifecycle of its unit. This approach always requires adding an additional component which would become a new ultimate master of the system and whose role would be the management of the "sub-masters" and system resources. Besides the complexity of the implementation, this approach also transfers the problem of scalability from sub-masters to the new master component. An alternative course of action could be running multiple independent original master components in the same time and distributing clients’ requests for services amongst them. For this purpose, an additional load balancer component could be designed. In this approach each master would have its dedicated slave servers and each slave server would communicate with a single master, or in more complex solutions with multiple masters. This approach would be much simpler than the previous one and it would not require extensive framework refactoring. Moreover, it would provide scalability by simply adding new masters to the system. On the other hand, the main drawback is an impossibility of dynamic adjustment of the system resources to the current needs of the framework.
Specifically, each additional master would have to be started and removed manually when the need for that occurs.

The aforementioned approaches for achieving fault tolerance and scalability of the master component in a distributed system can be applied to solving the concrete problems of the MapReduce JobTracker. Regarding fault tolerance, the first mentioned approach would always require running multiple copies of the same JobTracker in the system. They would form an active-passive relationship and use passive replication, always with one active master JobTracker which would replicate the changes of its state to the slaves. The main reason for the single active master limitation is the fact that the JobClient and TaskTracker components are implemented to communicate only with a single JobTracker and beside the coordination between masters in the system it would also require coordination between the masters and TaskTrackers and JobClients. When the master fails, some of the passive JobTrackers would take over its function, become the master and continue its operations, since it has all the state of the previous master. In this scenario, we would have to take care of several aspects and we could face several issues. First, when the master fails, a question of the state consistency can arise, since the state could be replicated only to some of the passive JobTrackers. Next, the passive JobTrackers would have to implement a failure detector, in order to reliably discover the master failure. Finally, in case we run multiple passive JobTrackers, they would have to run a leader election process to choose a new master. Another problem with this approach is a difficulty to integrate it with a solution for JobTracker scalability. In order to implement it by using the second approach for scalability, i.e. running multiple independent JobTrackers, we would double the number of JobTrackers in the system, since every running JobTracker would have to have at least one additional spare JobTracker which it would replicate its state to. Moreover, in order to tolerate multiple failures of the JobTrackers, we would have to run multiple passive JobTrackers for each master, whose only purpose would be to take over from the master.

In the second approach for fault tolerance, we would use a single persistent storage to store the JobTracker state. The storage should be distributed and provide automatic data replication between the storage nodes, in order to provide scalability and high availability of the data. Transactional support is also required in order to maintain consistency of the data in case the JobTracker fails. In this scenario we do not need to run passive JobTrackers, since the state is now stored in the persistent storage, and upon a JobTracker failure, a new single JobTracker
can start, load the state and continue where the previous one stopped. In this approach we still need to implement a component which would be in charge of detecting the JobTracker failure and restarting it on another node.

In our solution we decided to use the second approach and chose Infinispan as the persistent storage. The Infinispan was described in detail in the previous chapter and here we give the main reasons for choosing it. Since it is an in-memory data storage, it is very fast and thus suitable for frequent read and write operations which are necessary in the state checkpointing approach. Infinispan is reliable and offers data replication, and as such provides high availability of the data crucial for our system. Infinispan transactions system is suitable for ensuring data consistency and atomic execution of a group of operations. The Infinispan distribution mode enables partitioning the data across the cluster nodes and supports our system to scale when more data are added to the storage. The system is written in Java and does not require any additional drivers for client applications to communicate with it, thus providing full compatibility with MapReduce framework components and fast access. Infinispan DEF is convenient for providing failure detection and automatic failover of the JobTracker on another node, out of the box.

In order to simplify deployment and configuration, we integrated Infinispan into MapReduce, so each physical node can serve as a potential site for running both JobTracker and Infinispan instances. This also allows us to implement a scalable solution for the JobTracker, by running multiple independent JobTrackers in the system. We chose the second approach for providing the JobTracker scalability, since it is a simpler and clearer way for providing scalability and it does not impose new scalability constraints. As a central, highly available, persistent component in our system, we can also leverage Infinispan as a coordination and synchronization point for other components of our system.

### 3.4 JobTracker Fault Tolerance

The proposed system supposes running the JobTracker on a cluster of Infinispan nodes using the Infinispan DEF. Therefore, the system architecture contains four main components: Infinispan Cache instances, modified JobTracker, the Infinispan DEF master and the DEF failover policy.

Infinispan Cache instances form the Infinispan data cluster. Each Cache instance runs a new
process on a physical node. Beside serving the data, it also represents a possible location for executing a DEF distributed task.

Infinispan DEF master is a component that submits a new task to be executed on the DEF, controls its execution and restarts it in case of failure. DEF master uses a failover policy in order to decide on which node of the cluster to restart the task.

The DEF failover policy is a helper component which is a part of the DEF master. The failover policy defines where to restart the JobTracker upon a failure, based on the current conditions in the Infinispan cluster, and each time a failover occurs the DEF master consults it. The failover policy also sets custom parameters that define the JobTracker specific behavior during a new startup.

The modified JobTracker represents a MapReduce JobTracker instance which is adapted to store its data in the Infinispan Cache and it is also suitable to run on the DEF as a distributed task. The main architecture components are shown in the Figure 3.1.

In the figure we can notice two different clusters of nodes which overlap. The first is the Infinispan cluster which consists of the Infinispan Cache instances that act both as a data grid and a distributed execution environment. One Cache instance should be located on one physical node. The gray rectangles on the diagram represents physical nodes. The second cluster is the Hadoop cluster containing HDFS and MapReduce components: NameNode (NN), DataNodes (DN), TaskTrackers (TT) and JobClients (JC). The overlap of these two appears right in the
3.4. JOBTRACKER FAULT TOLERANCE

JobTracker which now physically runs on an Infinispan node, together with a Cache instance, but still interacts with other Hadoop nodes as usual.

During normal operation, the Infinispan DEF master starts a new instance of the JobTracker on one of the Cache nodes, already present in the cluster. Since the DEF master dynamically decides the IP address on which to start the JobTracker, the address is not publicly known, and the JobTracker, after the startup, will have to publish it to the cluster, storing it in the distributed Cache. This way it makes itself discoverable by the other Hadoop components. When choosing a node on which to start a JobTracker, Infinispan DEF master sorts all available destination nodes by the number of JobTrackers they already run and picks the least loaded node. The cluster statistics is stored in the Infinispan and it is updated by the DEF master each time a JobTracker is started or failed over. Note that for providing the fault tolerance of the JobTracker, the DEF master must not start the JobTracker on the same physical node where it resides, since if the node crashes both DEF master and the JobTracker would be permanently lost. To overcome this, when starting the whole system from scratch, we must have a node where we will initially start only an Infinispan instance and it will serve as a destination node for running a JobTracker in the future.

The JobTracker executes jobs received from JobClients and delegates tasks to the TaskTrackers, like in a standard Hadoop MapReduce framework implementation. Additionally, the JobTracker stores all of its state in the local Cache instance (located on the same physical node) which is replicated to the other nodes later on, in order to provide high availability of the data. Persisting the JobTracker data structure in the Infinispan represents a first step towards providing the JobTracker fault tolerance.

In a situation when the JobTracker fails, whether due to a crash of the JobTracker’s process or the whole physical node, the JobTracker DEF master will detect the failure, consult the failover policy in order to determine the next JobTracker node and restart the JobTracker on that node. In addition, the JobTracker DEF master will migrate all of the JobTracker’s state, from the other Cache node to the chosen node, in case it is not already present there, thus making its data locally available again. During the recovery startup, the JobTracker will load its previous state from the local Cache instance and continue with normal operation. The TaskTrackers and JobClients will also detect the failure of the JobTracker using their regular heartbeat mechanism, fetch the new JobTracker address from the cluster and reconnect to it, whereupon they can continue with
their normal operation. This process is shown in the Figure 3.2.

During the JobTracker downtime, all the MapReduce jobs’ tasks already generated and assigned for execution to particular TaskTrackers can continue with their execution and their status will be updated when the JobTracker starts again. Moreover, users can continue submitting new jobs, since the new JobClients components created for these jobs will continue retrying to connect to the JobTracker until it starts again and then submit the user jobs to the JobTracker.

Using Infinispan DEF we totally delegate the effort for detecting a failure and restarting the JobTracker to the framework, thus relieving the work from a failure detector implementation, which would be mandatory in case of a Master - Slave JobTracker approach. In addition, keeping only one JobTracker running in the system we eliminate the need for leader election and any additional coordination between the multiple JobTrackers. This design also supports multiple simultaneous JobTracker failures, provided that enough physical nodes are available, since there is one DEF master component for each JobTracker in the system.
3.5 JobTracker Scalability

We decided to choose an easy and efficient way for providing the JobTracker scalability which considered running multiple independent JobTrackers in the same cluster. In this design, TaskTrackers are divided amongst the JobTrackers and each of them receives tasks from a single JobTracker. On the other side, JobClients submit new jobs and a load balancing component distributes these jobs amongst the JobTrackers currently running in the cluster. This is shown in the Figure 3.3.

![Figure 3.3: JobTracker scalability overview](image)

Since we already explained the procedure for starting a new JobTracker in the Infinispan cluster in the previous section, those details are omitted in the Figure 3.3 for reasons of clarity.

We assume that for an optimal operation of the system there should be equal or more TaskTrackers than JobTrackers in the cluster. Each JobTracker must have at least one TaskTracker assigned all the time and that will be a permanent assignment. This is important in order to keep our load balancing and TaskTracker distribution design working and avoid gathering...
TaskTrackers around a single JobTracker.

3.5.1 TaskTracker Distribution

In the Master Server in Distributed Systems section, we mentioned static distribution of the slaves amongst running masters as a part of the solution for the master scalability. In many cases that distribution cannot be optimal, since the load of different master nodes varies over time and we could get into a situation where some masters and their slaves are idle while others are overloaded. In order to overcome this problem we design an algorithm for dynamic assignment and re-assignment of TaskTrackers to JobTrackers.

A TaskTracker can be assigned to a JobTracker on two occasions: when it starts for the first time and when it needs to reconnect. When a TaskTracker initially starts, it first tries to find a JobTracker without any TaskTrackers already assigned. If there is a single one, it connects to it. If there are several JobTrackers, it connects to a random one. In both cases, the TaskTracker stores the JobTracker identifier as permanently assigned. If all the JobTrackers have at least one TaskTracker assigned, the TaskTracker finds a JobTracker with the highest load and connects to it. If there are more JobTrackers with the same load, it picks a random one. The overview of the explained algorithm is presented in the following pseudo code:

if exist JTs without TTs
    TT joins a random JT from JTs
    TT stores the JT as a permanent JT
else
    TT finds the most loaded JT(s)
    TT joins a random JT from JTs

A TaskTracker will change the JobTracker it is currently connected to only in the case when it is not a permanent assignment and the TaskTracker is idle for a specified time. A TaskTracker is idle if there are no running jobs in the MapReduce subsystem that contains the TaskTracker and its JobTracker, i.e. neither the TaskTracker reports any task in progress nor does JobTracker have a job which is not completed. When changing the JobTracker, the TaskTracker repeats the same procedure for finding a new JobTracker as when it was started for the first time.
3.5. JOBTRACKER SCALABILITY

3.5.2 JobClient Load Balancing

A simple load balancing algorithm was designed to distribute the users’ jobs amongst the JobTrackers running in the cluster. When a JobClient wants to submit a new job to the cluster, it first tries to find the least loaded JobTracker and submit the job to it. If the least loaded JobTracker has no TaskTrackers assigned (JobTracker started and not enough TaskTrackers), JobClient will skip it and take the second least loaded one. If there are multiple JobTrackers with the same load, the JobClient will pick a random one.

The JobTracker load is calculated as a sum of all of its running jobs’ loads. The load of a single job represents the size of its splits on the hard disk, that were generated by a JobClient. A possible alternative for calculating the load could be the number of map and reduce tasks that JobTracker jobs contain in conjunction with the number of available slots of the assigned TaskTrackers.

Since Infinispan is a replicated, highly available, persistent storage, we use it as coordination support when providing JobTracker scalability. JobTrackers currently running in the cluster hold their IP addresses in the Infinispan store so the TaskTrackers and JobClients can always discover their current locations. JobTrackers also regularly update the status of their current load and store it in the Infinispan. TaskTrackers and JobClients read it when implementing JobTracker assignment and load balancing procedure, respectively. The whole process explained in the previous two sections is presented in the Figure 3.4.

Using the simple load balancing and TaskTrackers reassignment processes explained above, it is possible to achieve a better distribution of the running jobs and higher utilization of the cluster. That will speed up the jobs’ execution, lead to a smaller load of the running JobTrackers and provide for the scalability of the system. In the presented design, the JobTracker scalability can grow beyond any practical needs by simply adding a new physical node and starting a JobTracker instance on it.
3.6 Conclusion

In this chapter we presented a design for a fault tolerant and scalable JobTracker. We integrated the Infinispan data grid and Distributed Execution Framework into our solution. We provided JobTracker fault tolerance by storing its state into Infinispan and loading it upon the JobTracker failover, so it could smoothly continue normal execution. In our solution we preserved the work upon the JobTracker restart, i.e. there was no loss of the running jobs. The automatic failover was provided using Infinispan DEF and the failover was transparent to the end users and did not require any administrator intervention. We managed to achieve the JobTracker scalability by running multiple independent JobTrackers in the system. We designed a simple load balancing algorithm that distributed jobs amongst the running JobTrackers. Moreover, we created an algorithm for dynamic assignment and re-assignment of the TaskTrackers to the running JobTrackers, thus improving the utilization of the system resources.
3.6. CONCLUSION

The main drawback of the presented design is the necessity of running one spare Infinispan node in order to provide proper behavior of the system. Still, since the node serves as a possible JobTracker failover destination, when the first crash of the JobTracker happens, the system nodes will be fully utilized. The second flaw of the solution concerns the fact that the Infinispan DEF master is not fault tolerant. In case it fails, the JobTracker that was managed by it cannot be automatically restarted anymore in case it also fails later on. Solving this DEF master flaw will be a part of our future work. In the following chapter we will discuss the implementation of the presented design.
4.1 Introduction

In this chapter we give implementation details and further elaborate the presented design of the solution. We start by giving the details about the source code versions of the Apache Hadoop and JBoss Infinispan used in the implementation.

Next, we continue with the details about the JobTracker fault tolerance implementation. As mentioned in the previous chapter, we chose to preserve the JobTracker internal state by storing it in the Infinispan storage. Thus, we first cover all the important issues related to this process and how we solve them in our implementation. We use Infinispan DEF framework to provide the JobTracker failure detection and automatic failover, so we give details about the two new Hadoop daemons we developed for this purpose. We explain the algorithm that JobClients and TaskTrackers use to re-connect to the JobTracker after it recovers from a failure. At the end we give an overview of the Infinispan configuration used for the implementation of our fault tolerant JobTracker.

We continue this chapter explaining the implementation of our scalable JobTracker. As a reminder, we provide scalability by allowing multiple JobTrackers to run in parallel in the system and distributing the load amongst them. Accordingly, we first explain the adjustments done on the JobTracker to be able to run multiple instances of it in parallel. Next, we explain the algorithm implemented on the JobClient side that distributes job submission requests to the JobTrackers running in the system as well as the process of the TaskTrackers (re-)assignment to the running JobTrackers.

Then, we explain in detail each Infinispan data structure that was used for coordinating the running components of the system and collecting statistics used for load balancing and assignment algorithms. Finally, we show the new configuration options required by our solution.

\footnote{The implementation is publicly available at https://subversion.assembla.com/svn/hadoopft/release-1.0.4}
that we added to the MapReduce configuration file, as well as the changes made to the main Hadoop execution script in order to start additional Hadoop daemons we implemented.

At the very end of the chapter we describe the third contribution of our work which includes minor improvements done on the Infinispan DEF framework. In order to completely support a custom failover of the JobTracker we made several changes to the Infinispan failover policy mechanism and these changes are presented here.

4.2 Source Code Versions

The presented architectural design was implemented in the Hadoop MapReduce source code, 1.0.4. release. This version of Hadoop belongs to the Hadoop 1.0 distributions family with the label "stable", which is the most frequently used Hadoop release in big production environments such as the ones in Facebook, Yahoo and LinkedIn. The chosen Infinispan source code distribution was 5.2.1, the latest at the time of the implementation work.

4.3 JobTracker Fault Tolerance

We can divide the JobTracker fault tolerance implementation into several steps. The first and at the same time the basic step concerns the mechanism for checkpointing the JobTracker internal state into the Infinispan storage. That requires identifying the crucial JobTracker data structures, their optimal amount and checkpointing frequency, the data serialization procedure, and a mechanism for providing data consistency in the moment of a JobTracker failure. When we ensure that the JobTracker state is stored correctly, we can approach the second step which considers running the JobTracker on the DEF framework, which would provide an automatic failover. That requires adjusting the JobTracker code to run as a DEF distributed task and implementing the JobTracker failover policy. Finally, the last step includes changes made on the JobClient and TaskTracker daemons in order to be able to find the re-started JobTracker and to connect to it again. This final step completes the JobTracker failover process and puts the MapReduce system in an operating state.
4.3. JOBTRACKER FAULT TOLERANCE

4.3.1 Saving the Internal State

In order to provide the fault tolerance of the JobTracker in such a manner that it can smoothly continue its normal execution upon a failover, we need to store the JobTracker state in a persisted storage. Infinispan data storage saves the data in memory, thus the data saved in the local Cache instance are never serialized. However, when replication is used, the data have to be serialized in order to be sent to the other nodes. Infinispan uses JBoss Marshaling mechanism to transform the data into bytes so it can be sent over the network to the other nodes in a cluster and to transform the data again so it can be stored in memory in the underlying Cache stores. JBoss Marshaller represents a more efficient serialization framework for serializing Java classes than the standard JDK Serialization mechanism, while remaining fully compatible with java.io.Serializable interface. In order to take advantage of the Jboss Marshaller, objects that are to be stored in the cluster have to implement either the java.io.Serializable or the java.io.Externalizable interface.

For the first iteration of the state saving implementation we decided to make the JobTracker stateless, i.e. the JobTracker would read and write all its internal data objects from/to the Infinispan all the time. Nevertheless, due to a need for synchronization on some of these objects in multiple places in the source code, the stateless approach requires additional effort for implementing a new mechanism for object synchronization. Thus, we decided to leave all the JobTracker data structures in memory, which should be accessed in normal operation, and also on every update of the data, to persist them into Infinispan. In this way, we always have an up-to-date redundant copy of the JobTracker state objects and we can also control the access to these objects in the JobTracker code. The stored data is loaded from the storage into memory only upon the JobTracker restart, before it starts its internal services and becomes visible to other daemons.

4.3.2 Stored State

The JobTracker maintains many data structures for tracking the jobs, tasks and TaskTrackers’ statuses and they are updated regularly with each TaskTracker heartbeat. Thus, the JobTracker state we store in the Infinispan consists of the most of the JobTracker class fields. The fields that are not persisted are Java Thread objects, the MapReduce jobs listeners, file system
instances and Hadoop RPC server and client instances - basically the objects that do not maintain the JobTracker management state required for the successful fail-over or the objects that maintain the state already persisted on HDFS. The internal data structures of the JobTracker class which are persisted into Infinispan are presented in the following listing in the format in which they are declared in the source code:

```java
DNSToSwitchMapping dnsToSwitchMapping
Set<JobID, JobInProgress> jobs
TreeMap<String, ArrayList<JobInProgress>> userToJobsMap
Map<String, Set<TaskAttemptID>> trackerToTasksToCleanup
TreeMap<String, Set<JobID>> trackerToJobsToCleanup
Map<String, Set<TaskTracker>> hostnameToTaskTracker
TreeMap<String, Set<TaskAttemptID>> trackerToTaskMap
Map<String, HeartbeatResponse> trackerToHeartbeatResponseMap
TreeMap<TaskAttemptID, String> taskidToTrackerMap
TreeMap<String, Set<TaskAttemptID>> trackerToMarkedTasksMap
Map<String, FaultInfo> potentiallyFaultyTrackers
HashMap<String, TaskTracker> taskTrackers
Map<String, Integer> uniqueHostsMap
Map<String, JobIDStatusMap> jobIDStatusMap
LinkedList<RetireJobInfo> jobRetireInfoQ
```

The state objects of the JobInProgress class persisted in the Infinispan are the following:

```java
Map<Node, List<TaskInProgress>> nonRunningMapCache
Map<Node, Set<TaskInProgress>> runningMapCache
Set<TaskInProgress> nonLocalRunningMaps
Set<TaskInProgress> nonRunningReduces
Map<TaskTracker, FallowSlotInfo> trackersReservedForMaps
Map<TaskTracker, FallowSlotInfo> trackersReservedForReduces
Map<TaskInProgress> cleanup
Map<TaskAttemptID, Long> launchingTasks
Map<TaskTracker, FallowSlotInfo> trackersReservedForMaps
Map<TaskTracker, FallowSlotInfo> trackersReservedForReduces
```

As we mentioned before, MapReduce jobs listeners, that are in charge of handling submissions of new jobs, are not persisted as a whole but rather their substructures are. The two structures
4.3. JOBTRACKER FAULT TOLERANCE

of JobQueueJobInProgressListener and EagerTaskInitializationListener classes persisted in the Infinispan are shown in the following listing respectively:

\[
\text{Map<JobSchedulingInfo, JobInProgress> jobQueue} \quad | \quad \text{List<JobInProgress> jobInitQueue}
\]

### 4.3.3 Serialization Optimization

Since the serialization of Java object adds a significant overhead to the system\(^2\), we tried to optimize this process as much as possible.

The JobTracker class, as a main starting point of object serialization, is not serialized as such, but rather its sub-structures are. All the objects that need to be serialized implement the Java Serializable interface and in that way they become suitable for using the JBoss Marshalling framework. The main JobTracker data structures are objects that represent running jobs, their respective tasks and connected TaskTrackers. Their Java class implementations are JobInProgress, TaskInProgress and TaskTracker respectively. These objects are frequently accessed and they are contained in multiple collections in the JobTracker as well as in several other classes. In order to avoid loading a whole collection from the storage, when we only want to access a single object in it, as well as to maintain consistency of the object state, we keep the same object contained in different collections in a single place in the persistent storage\(^3\). We store only the unique identifiers of the objects in a collection when we persist it, and the objects themselves we save as separate Cache entries, one for each object of the collection. The same holds for the JobInProgress and its data structures, since it is a container for the collections of the TaskInProgress objects. This is shown in the Figure 4.1. In this way, collections are loaded from the storage only when there is a need for adding a new object to it or removing an existing one and each object is updated independently of the collections. In addition, upon a JobTracker restart, when loading the whole state from the storage to memory, an object is loaded only once and stored in the respective collections, based on its identifier. In that way, we get a consistent state in the memory as it was before the JobTracker crash, which excludes a possibility for errors. (e.g. change of an object in one collection did not affect all the other collections that contain the same object).

---

\(^2\)De-serialization of a Java object is especially demanding since the object has to be re-created from a byte array which consumes significant time.

\(^3\)This is the same approach as objects references in main memory: different collections reference the same object.
4.3.4 Cache Key Formation

Each data structure that should be stored in Infinispan has to be uniquely identified by its key and thus special care has to be taken when storing the data structures that do not already have a Hadoop identifier class object assigned.

For each JobTracker data structure, one Cache entry (key, value) which holds its data is created in Infinispan. For most of the data structures, keys are represented as DataKey enum values, and the key names are capitalized names of the respective Java fields (e.g. runningTasks -> RUNNINGTASKS). On the other hand, for storing the main JobTracker objects (JobInProgress, TaskInProgress and TaskTracker) their unique class identifier objects are used as a key. The unique identifier is a simple Java object with several String fields. In addition, since there are multiple JobInProgress instances for one JobTracker, all the data structures of one JobInProgress instance that need to be saved have to be "personalized" in the storage, by adding the JobInProgress unique identifier to the keys of these data structures. In order to support multiple JobTrackers running simultaneously in the cluster, all the data structures stored for one JobTracker that are not globally (cluster wide) uniquely identified have to be additionally "personalized" by adding the JobTracker unique identifier to the keys of all of its Cache entries.

\footnote{Many Hadoop classes reference other classes that represent their unique identifiers, e.g. JobInProgress class instances use JobID class instances as their identifiers.}
### 4.3. JOBTRACKER FAULT TOLERANCE

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Cache key components</th>
<th>Cache key type</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;TaskInProgress&gt; tasks</td>
<td>jobTrackerIdentifier: String, jobInProgressId: JobID, DataKey:TASKS: String</td>
<td>String</td>
</tr>
<tr>
<td>TaskInProgress task</td>
<td>taskInProgressId: TaskID</td>
<td>TaskID</td>
</tr>
</tbody>
</table>

Table 4.1: Cache key formation example

An example of creating keys for storing one collection field and one single object field of JobInProgress class is given in the Table 4.1.

As can be noticed from the previous example, TaskInProgress already has an object identifier of type TaskID which consists of several String fields that uniquely identify it across the cluster, thus it does not need an additional JobTracker or JobInProgress identifier to be added.

#### 4.3.5 Transactions and Locking

Although the relationship between JobTracker and TaskTracker can be classified as master-slave, the TaskTracker is the one who actually sends heartbeats to the JobTracker and initiates a request for new tasks to be generated. Every heartbeat can be seen as a cycle of JobTracker operations. After it receives a heartbeat from a TaskTracker, the JobTracker updates its internal state based on the notifications about the progress of the tasks and generates the new tasks to be executed on that TaskTracker. All of these calculations are executed inside the JobTracker heartbeat() method which is called by a TaskTracker. The TaskTracker waits until the JobTracker finishes and the method returns with the JobTracker heartbeat response object. The JobTracker stores the unique identifier of each heartbeat, so when it receives a new one it knows whether it was already processed or not. In order to ensure the consistency of the JobTracker in-memory state with the Infinispan replicas and consequently the correctness of the JobTracker operations in case of a failover, we use the Infinispan transactions mechanism to provide a guarantee that all of the operations of the heartbeat() method are executed atomically, i.e. all of them or none. Thus, the whole heartbeat() method is put into a single transaction. We will elaborate later about the correctness of the JobTracker operations in case of a failover.

Several other places in the JobTracker code require an atomic execution of the operations, such as JobTracker threads for expiring launching tasks and faulty TaskTrackers and removing
finished jobs. Each of these threads runs an endless loop which accesses the JobTracker state data every specified time interval, so one iteration of the loop represents a transaction.

A significant flaw of the Infinispan transactions mechanism is that it does not support a transaction retry in case of a rollback. In our implementation that could be a big issue, since if a transaction aborts for some reason which does not include a JobTracker process crash, the state will be saved in the JobTracker data structures in memory but not in the Infinispan persistent storage. To avoid this situation, we use pessimistic locking with a big lock acquisition timeout. In this way we try to avoid deadlocks and unnecessary rollbacks which, on the other hand, optimistic locking would cause. From the performance point of view, pessimistic locking slows down the system and adds an overhead. Nevertheless, there are only a few threads in the JobTracker code and all of them, except the main JobTracker thread, access a limited amount of data infrequently, so a conflict between two transactions will rarely happen.

4.3.6 Asynchronous Replication

As we mentioned earlier, storing an object to the replicated Cache requires serialization process to transfer it to the other nodes and then de-serialize it on each node. This process greatly slows down the execution of the MapReduce framework, since a call to the Cache instance to store an object blocks the execution of the JobTracker thread until the object is successfully replicated to the other nodes in the cluster. Consequently, the higher the replication level, the slower the process will be. Nevertheless, Infinispan offers an optimization of this process, i.e. asynchronous replication, which allows a caller method to return immediately after the data has been stored in the local Cache. The replication itself will happen in a separate thread in the background. The main drawback of the asynchronous replication is a potential inconsistency of the data on the other nodes. Nonetheless, since we access the data of the Infinispan cluster only from the JobTracker process, the inconsistency issue does not cause a problem in our case.

4.3.7 Correctness of the Operations in Case of Failure

Depending on the status of the currently executing operations, the JobTracking failure can create different effects and inconsistency between the JobTracker memory and Infinispan storage and consequently the correctness of the JobTracker operations upon a recovery can be questioned.
4.3. JOBTRACKER FAULT TOLERANCE

Here, we discuss several doubtful scenarios.

From the perspective of the JobTracker - TaskTracker communication we can differentiate between two cases:

- In case the JobTracker fails while processing a TaskTracker heartbeat in the middle of a transaction, the ongoing Infinispan transaction will discard all the changes made on the Infinispan cache. The TaskTracker will receive an error from the heartbeat call and it will retry to execute the same heartbeat next time it connects to the JobTracker. When the JobTracker recovers, it does not have the information about the heartbeat that the TaskTracker resends, so it will process it again.

- In a rare case where the JobTracker fails after the transaction is committed but the heartbeat method still has not returned, the modifications will be persisted in the Infinispan but the TaskTracker will receive an exception from the heartbeat call. Upon the JobTracker restart, the TaskTracker will try to execute the same heartbeat again but the JobTracker will ignore it, having the information that it already processed it.

Taking into account the consistency of the data between the replicated nodes in the cluster, when using asynchronous replication, we can also discuss two situations:

- If the crash happens before the transaction commit is called, all the writes are lost and none of the replication nodes sees the changes. The TaskTracker will repeats the previous heartbeat and the JobTracker will process it again as in the previous example.

- In case the JobTracker call successfully returns from the commit method, it means that Infinispan has already sent the modification to the other nodes. Assuming the network is reliable and the messages are not lost, all the other nodes will apply these modifications eventually. Therefore, a crash of the JobTracker that occurs at any point after the commit returns does not cause any inconsistency and this scenario can be reduced to the second case described in the previous paragraph.

---

5The JobTracker heartbeat() method is synchronized, i.e. at each point in time JobTracker can process only one heartbeat call from a single TaskTracker.
4.3.8 Running the JobTracker on the Infinispan Distributed Execution Framework (DEF)

As explained in the Architecture chapter, we use the Infinispan Distributed Execution Framework to provide an automatic failover of the JobTracker. In order to be suitable for starting it on the Infinispan DEF, several changes have to be made in the JobTracker code in order to satisfy the requirements posted by the DEF distributed task component.

The fault tolerant JobTracker implements the DEF DistributedCallable interface, which in its basics extends the Java Callable interface with the possibility to start a thread on the remote node. Two methods from the DistributedCallable interface that are required to implement are the call() method, which basically replaces the JobTracker main() method and setEnvironment() method, used to set the Infinispan Cache instance and custom parameters upon a failover. The custom parameters important for the JobTracker are: a boolean parameter indicating whether the JobTracker is restarted or freshly started, and the JobTracker port and its identifier, which are generated in the moment of the first start.

For the purpose of moving the JobTracker instance from one node to another, as required by the DistributedCallable, the JobTracker has to implement either the Java Serializable interface or provide for a custom Externalizer implementation. Since we want to serialize only the JobTracker port and its identifier, when sending it to another node, we chose to implement a custom Externalizer, which provides us control over the serialization process.

For the purpose of the JobTracker failover, two types of daemons have been additionally implemented and when started, they represent logical nodes that form an Infinispan cluster:

- The IspnNode daemon starts an Infinispan Cache instance on some physical node and initiates or joins the Infinispan cluster depending on whether it is the first instance or not, respectively. This is the peer-to-peer running mode of Infinispan, where each Cache instance starts in the same JVM process as the IspnNode program. Besides the Cache instance, IspnNode also starts a Hadoop RPC Server listener on a specified port that waits for the TaskTrackers and JobClients remote requests for fetching the Infinispan data from the cluster.\(^6\)

\(^6\)We suppose that TaskTrackers and JobClients run on separate physical nodes, thus they cannot access the Infinispan cluster directly. For the remote access to the cluster, Infinispan provides the HotRod client - server architecture. Still, this kind of running mode cannot be mixed with the p2p mode.
The JobTrackerDef daemon also starts an Infinispan Cache instance and a Hadoop RPC Server listener. However, the main feature of this daemon is that it starts a new JobTracker instance on one of the already running Infinispan nodes, using Infinispan DEF framework. After starting the JobTracker, JobTrackerDef holds a remote reference to the running remote JobTracker and, in case of failure, restarts it on some other, healthy node. The failure can be a crash of the JobTracker process or a crash of the whole physical node. For the purpose of a failover, the JobTrackerDef uses the JobTracker failover policy implementation of the Infinispan DistributedTaskFailoverPolicy interface, which decides on which node to restart the JobTracker. In this way, a JobTracker address is not statically configured and read from the configuration as it was before, and now it depends on the IP address of the node chosen by the DEF components. When the JobTracker starts its internal services on the assigned node, it will "publish" its IP address to the Infinispan storage to enable other components to discover it.

4.3.9 JobTracker Node Election Policy

Upon a fresh start as well as after a failover, the DEF framework needs to decide on which node to start the JobTracker. Accordingly, both JobTrackerDef and JobTracker failover policy use the same procedure which is based on the cluster statistics data they fetch from an Infinispan Cache variable. Based on that information they will first try to find a free node in the cluster (a node without JobTrackers already started) and start the JobTracker there, also skipping the current node. If there is no such node, the JobTracker will start on the least occupied node (based on the number of running JTs), and the JobTracker port will be dynamically assigned, based on the number of already executing JobTrackers on the chosen node. In case the current node, which runs the JobTrackerDef and JobTrackerFailoverPolicy, is the only node in the cluster in that moment, the JobTracker will start on the current node. This is a worst case scenario and a crash of the node would mean the ultimate failure of the JobTracker, since the JobTrackerDef daemon would fail as well, thus no failover would be possible.

4.3.10 JobClient and TaskTracker Re-connection to the JobTracker

When a failover of the JobTracker occurs, JobClients and TaskTrackers that previously communicated with that JobTracker need a procedure to establish a new connection to it, since the
JobTracker now runs on a different physical node. They will first try to reconnect to the JobTracker on the same address as before several times (in case there was an intermittent fault and not a real failure) and after that they will contact one of the Infinispan nodes to obtain the new address of the JobTracker. They will get it from a variable which contains the information about JobTrackers in the cluster. These cluster variables will be explained in detail later. We make the assumption that, if the JobTracker really fails, during the time JobClients and TaskTrackers try to reconnect to the same address, it will manage to restart on some other node and publish its new address to the cluster. To communicate with remote Cache instances, TaskTrackers and JobClients use the client side of the Hadoop RPC infrastructure. Both JobClients and TaskTrackers have the address of an Infinispan node which they should contact first already assigned. They initiate the address upon a first start, choosing the first one from the list of available node addresses, which are defined in the configuration file. They will first try to contact that node and in case the node failed together with the JobTracker, JobClients and TaskTrackers will have to re-connect to another Infinispan node first, before getting the information about the new JobTracker address. When the JobClients and TaskTrackers successfully reconnect to their JobTracker, they can continue with the normal operation.

4.3.11 Infinispan Configuration

Several Infinispan settings that we specify in the configuration in order to support our solution are worth mentioning and they are summarized here.

The Infinispan cluster is formed using the P2P running mode which is required by the nature of our JobTrackerDef and InfinispanNode daemons deployment. For the Infinispan cluster mode, we chose the distribution mode, which would provide the scalability of the system, especially when the JobTracker runs a large number of jobs which generate a vast amount of data to be stored in the cluster. We chose replication level two, with the assumption that the second replica will not crash before an administrator discovers the crash of the original node and starts the second node again. To improve the performance of the system, as we already mentioned, we use asynchronous replication, since we access the data from a single node and we do not have a problem with the consistency issue. For the Infinispan transactions mechanism, we use the org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup implementation of the JTA Transaction manager. The locking mode we chose is pessimistic locking in order to
4.4 SCALABILITY

In this chapter we give some implementation details in accordance with the scalability design explained in the Architecture part. Since our scalability solution is based on running multiple JobTrackers in the system, we first explain the changes made on the JobTracker component in order to support multiple running instances. Next, we discuss the core of the solution which consists of the automatic distribution of users' submitted jobs and (re)distribution of the running TaskTrackers amongst the JobTrackers running in the system. Finally, we discuss some consistency questions raised by simultaneous access to the same Infinispan data from multiple JobTrackers.

4.4.1 Running multiple JobTrackers in the Cluster

In our implementation, adding a new JobTracker to the system is as easy as calling the JobTrackerDef daemon, which starts a new JobTracker instance on one of the cluster nodes, using the regular procedure previously explained in the JobTracker fault tolerance section. Nevertheless, although now running on an Infinispan cluster node, JobTracker(s) is still a part of the Hadoop cluster and uses HDFS for storing and retrieving its jobs metadata. Bearing that in

```xml
<infinispan>
  <global>
    <transport />
  </global>
  <default>
    <clustering mode="distribution">
      <async />
      <hash numOwners="2" />
    </clustering>
    <transaction
      transactionManagerLookupClass="org.infinispan.transaction.lookup
        .JBossStandaloneJTAManagerLookup"
      transactionMode="TRANSACTIONAL"
      lockingMode="PESSIMISTIC" />
    <locking
      useLockStriping="false"
      lockAcquisitionTimeout="360000" />
  </default>
</infinispan>
```
mind, several adjustments have to be made in order to support multiple JobTrackers running in the cluster at the same time.

The problem that is imposed first is the static assignment of IP address and port to the JobTracker RPC server. Since JobTrackers have to be running on different nodes, their IP addresses are now assigned dynamically by the Infinispan DEF instance which starts and fails over the particular JobTracker. Since our system allows multiple JobTrackers to run on the same physical node, JobTracker ports have to be assigned dynamically as well, based on the number of JobTrackers already running on a particular Infinispan node. In our system the port is also assigned by Infinispan DEF instance.

The JobTracker Http server, used to provide an HTML view of JobTracker operations also uses a fixed port and in the new implementation it is dynamically assigned as well. The JobTracker uses a system directory, located on the HDFS, to store several system files. The path of the directory is configured statically in the "mapred.system.dir" configuration property, in the original distribution. In order to avoid overwriting the files in that directory by different JobTracker instances, the directory name is now customized for every JobTracker running in the cluster. The easiest way to implement this is to add the JobTracker identifier to the name of its system directory and that way distinguish it from the other JobTrackers' directories.

### 4.4.2 TaskTrackers Assignment and Load Balancing

The most important pieces of our scalability solution are automatic (re)distribution of the running TaskTrackers to the running JobTrackers and the balancing of the jobs' load submitted from the client side. These two aspects of the implementation are presented in this section.

As explained in the Architecture chapter of the thesis, a TaskTracker can change the JobTracker it is currently connected to only in case it is not a permanent assignment and the TaskTracker is idle for a specified time. If a TaskTracker does not get a permanent assignment, upon startup (and restart as well), it will start a separate thread, called IdleChecker thread, which serves to change the assigned JobTracker after a specified idle time. Specifically, the thread calculates the time interval in which the TaskTracker is idle. A TaskTracker is idle if there are no running jobs in the subsystem containing the TaskTracker and its JobTracker, i.e. neither the TaskTracker reports any task in progress nor does JobTracker have a job which is
4.4. SCALABILITY

not completed. To check the status of the JobTracker jobs, the IdleChecker thread contacts the JobTracker every heartbeat time interval. After the maximum idle time expires, the IdleChecker thread will initiate a change of the JobTracker by checking the other JobTrackers from the cluster, using Infinispan cluster statistics and finding the most loaded JobTracker. If it finds a more loaded JobTracker, it will send the current JobTracker a special "leaving" signal in the next heartbeat, which will cause the JobTracker to issue the ReinitTracker action for the TaskTracker. After that, the TaskTracker will restart and re-initialize its internal state and repeat the same procedure for finding a JobTracker as it was started for the first time. The maximum idle time for a TaskTracker is defined as an additional parameter in the MapReduce configuration file.

The load balancing component is implemented in the JobClient code. Each time a user wants to submit a new job, the JobClient will contact the Infinispan to fetch the statistics of the currently running JobTrackers and find the least loaded JobTracker which has TaskTrackers already assigned. It will then submit the user job to that JobTracker.

Every time a new job is received from a JobClient and its splits are generated, the JobTracker updates its load in the appropriate data structure in the Infinispan. When the job finishes, the JobTracker updates the load again.

4.4.3 Consistency Issues

In the JobTracker fault tolerance section we explained that our implementation accessed Infinispan cluster data from a single process, i.e. a single running JobTracker. This is not the case when providing JobTracker scalability, since we run multiple JobTrackers simultaneously. Besides their own, personalized data, now they also require access to shared Infinispan statistics objects. This may cause inconsistency issues, since multiple JobTrackers can read and modify the same data at the same time, especially when using asynchronous replication of the data. In order to avoid this scenario, we use the Infinispan named cache feature which allows us to start multiple independent clusters of Cache instances that can share the same physical nodes. For the purpose of storing Infinispan statistics objects, we start one separate cluster of Cache instances which are configured to execute operations synchronously. Additionally, for each scope of dependent read and write operations we create a transaction and use optimistic locking. Using these two mechanism we ensure the isolation of the modifications of the shared Infinispan data and the consistency of the data on replicated nodes.
4.5 Infinispan Data Structures

As briefly mentioned in previous sections, in addition to storing JobTracker internal data structures, we utilize Infinispan to store different cluster statistics which are used to support both fault tolerance and scalability implementations. These data structures are explained in detail in this section.

The definitions of two main data structures held in Infinispan are shown below:

\[
\text{JTSTATS : Map\{String, JobTrackerStats\}}
\]
\[
\text{ISPNSTATS : List\{IspnStats\}}
\]

- The JTSTATS data structure contains a map that for each JobTracker in the system holds its unique identifier as a key and its statistic, contained in the JobTrackerStats object, as a value. JobTrackerStats contains information such as the current IP address of the JobTracker, its current load and a list of its task trackers and their capacities.

A JobTracker publishes its IP address to the cluster and stores it in the JTSTATS variable every time it starts internal services; it can be at the initial start or upon a failover. This information is used by TaskTrackers and JobClients when they want to discover the current location of the JobTracker.

The JobTracker load is updated each time the status of its jobs changes, i.e. a new job is submitted or a running job is completed. This information is used again by TaskTrackers and JobClients to find the least / most loaded JobTracker.

The information about currently assigned TaskTrackers contains their unique identifiers mapped to their map and reduce capacities. This information is updated each time a TaskTracker connects to the JobTracker or sends the "leaving" message to it. It is used by TaskTrackers and JobClients to detect JobTrackers without assigned TaskTrackers, and it could be also used for the alternative procedure for the load calculation, mentioned in the Architecture chapter.

- The ISPNSTATS data structure contains basic statistics about the Infinispan cluster. It is a collection that holds one object for each Infinispan node. The statistics consists of the IP address of a node and the number of JobTrackers currently running on that node.
### 4.6. HADOOP CONFIGURATION

<table>
<thead>
<tr>
<th>Name</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapred.infinispan.node.addresses</code></td>
<td></td>
<td>Comma separated list of IP addresses (without port) of the physical nodes which the Infinispan Cache instances are run on</td>
</tr>
<tr>
<td><code>mapred.infinispan.node.port</code></td>
<td>6666</td>
<td>Infinispan node port for starting Hadoop RPC server</td>
</tr>
<tr>
<td><code>mapred.jobtracker.port.default</code></td>
<td>9990</td>
<td>Default port for JobTracker RPC server</td>
</tr>
<tr>
<td><code>mapred.tasktracker.idlemaxtime</code></td>
<td>30000</td>
<td>TaskTracker max idle time (in milliseconds) before it starts looking for a new JobTracker</td>
</tr>
<tr>
<td><code>mapred.ispnstats.ascending</code></td>
<td>true</td>
<td>Whether to sort Infinispan nodes by the number of running JobTrackers ascending or descending (used for testing purposes only)</td>
</tr>
</tbody>
</table>

**Table 4.2: Hadoop configuration**

The number of JobTrackers on a node is updated every time a new JobTracker starts (updated by the JobTrackerDef instance) or an existing JobTracker fails over (updated by the JobTracker failover policy). The information is used by the same components to choose on which node to start the JobTracker as well as to assign an appropriate port to the JobTracker, in case there are multiple JobTrackers running on the same node. An additional boolean property has been added to the statistics object which is used for testing purposes only. It tells the JobTrackerDef and JobTracker failover policy whether to start a JobTracker on a node with the most or the fewest JobTrackers. If it is the first case, it will basically provide a gathering of JobTrackers on a single node.

### 4.6 Hadoop Configuration

Several configuration parameters that are important for the fault tolerance and scalability features are exported into an external MapReduce configuration file (mapred-site.xml), so they can be dynamically changed and can take effect without recompiling the source code. These are summarized in the Table 4.2.
4.7 Changes to the Hadoop Execution Scripts

Since we developed two new Hadoop program daemons, IspnNode and JobTrackerDef, we had to update the main Hadoop execution script file, bin/hadoop, to add the ability for running them. Both can be started like all the other Hadoop daemons, i.e. "bin/hadoop ispnnode" and "bin/hadoop jobtrackerdef". The IspnNode daemon will start an instance of the Infinispan cache and initialize or join the Infinispan cluster, depending on whether it is the first node in the cluster or not, respectively. The JobTrackerDef daemon will also start an Infinispan Cache instance and additionally a fault tolerant JobTracker on a specific node in the cluster.

4.8 Infinispan Improvements

There are several differences in the startup process of a new and a restarted JobTracker. Thus when starting up, the first thing a JobTracker needs to know is whether it was restarted or freshly started. The restarted JobTracker has to load all of its previous state from Infinispan before starting its internal services. Next, since we store the data for each JobTracker under keys distinguished by the JobTracker identifier, the restarted JobTracker needs to know its identifier before it starts loading the appropriate storage data. Finally, in order to start its internal services and make itself publicly available, the JobTracker needs to know a new port within the physical node, assigned by the DEF failover component. Because of these requirements, every time the JobTracker starts or fails over, it needs to get this information from the JobTrackerDef instance or JobTracker failover policy, respectively. In the current implementation of the DEF this operation is not possible in the latter case (discussion at [32]). This work makes a contribution to the Infinispan source code by adding a possibility to the failover policy mechanism to set a user specific parameters map which can be loaded from a user Callable later on. The changes have been incorporated into the DEF framework itself, believing that all the other DEF distributed tasks can benefit from it. The signature of the method added to the DistributedTaskFailoverPolicy interface is presented in the following snippet of code:

Map<Object, Object> getEnvironmentParameters()

At the DistributedCallable side, the method setEnvironments was changed to read previously set parameters in the following way:
setEnvironment(Cache<K, V> cache, Set<K> inputKeys, Map<Object, Object> params)

4.9 Conclusion

In this chapter we gave implementation details and further elaborated the presented design of our solutions for JobTracker fault tolerance and scalability issues.

We started with the JobTracker fault tolerance implementation and identified the JobTracker state that was required to be stored in Infinispan in order to provide a smooth failover and the techniques and optimizations for the state data structures serialization. We explained the necessity of using transactions and locking mechanisms in our implementation and, regarding that, we discussed the correctness of the JobTracker operations in case of a failure. Next, we showed the changes that have been made to the JobTracker code in order to start it as an Infinispan DEF distributed task and described the two new Hadoop daemons we developed for the purpose of running a fault tolerant and scalable JobTracker. We explained the JobTracker node selection policy and the JobClient and TaskTracker procedure for re-connecting to the JobTracker.

For the scalability implementation we first described the changes done on the JobTracker in order to run multiple instances of it in the cluster. Next, we gave implementation details for JobClient load balancing algorithm and TaskTracker assignment procedure. Finally, we discussed the consistency issues that could appear in the multi-JobTrackers environment.

After the two main implementation parts of our solution we explained additional data structures we kept in the Infinispan that were used for both fault tolerance and scalability implementations and listed the configuration parameters that were newly added and externalized into MapReduce XML configuration file.

At the end of the chapter we described the minor improvements we made to the Infinispan DEF framework in order to support a custom failover policy.

In the next chapter, we will show the evaluation of the described implementation.
5.1 Introduction

In this chapter we describe the methodology used to evaluate our Hadoop fault tolerant and scalable implementation and discuss the obtained results. The main evaluation goal was to compare the solution with the official Hadoop release in terms of the performance and cost and to identify potential bottlenecks and overheads.

The chapter starts by explaining the tools used for benchmarking and discussing different data sets used in experiments. Next, we describe the testing environment and give details about its settings. Thereafter, we present different experiments performed on the solution. For each experiment we explain its goal and the specific experimental set-up, and discuss the obtained results. As stated in the Introduction chapter, the main goal of the thesis is to provide an efficient implementation of the fault tolerant and scalable MapReduce JobTracker, which will not significantly degrade the system performance. From this generic goal we extracted several system requirements and listed them in the Architecture chapter. All the experiments presented here have been performed in accordance to these requirements in order to show the level of their fulfillment. Thus, in each particular experiment we will refer to its related requirements.

5.2 Benchmarking Tools

Here we present two main software tools we used for measuring results in our experiments.

5.2.1 HiBench Benchmark Suite

HiBench is a new and increasingly popular benchmark suite for evaluating and characterizing the Hadoop framework. It consists of a set of Hadoop programs, including both synthetic micro-benchmarks and real-world Hadoop applications [33]. Its source code is publicly available [34],
thus allowing customization and adjustments to the specific needs. HiBench consists of 8 Hadoop MapReduce benchmark programs classified into four categories: Micro benchmarks, Web search, Machine learning and HDFS benchmarks.

In our experiments we used several HiBench MapReduce programs to evaluate our solution in terms of the execution performance and these are explained below.

- WordCount, Sort and TeraSort are micro benchmarks widely used in the Hadoop community and they are also contained in Hadoop MapReduce distributions. WordCount is a basic benchmark where each map task produces a pair (word, 1) for each word in the input data set, the combiner partially sums the numbers of the occurrences of each word in a map task, and the reduce task computes the final count for each word. The Sort benchmark sorts the final results, and both its map and reduce tasks are simple functions that directly pass the input key-value pairs to the output. The TeraSort program sorts 10 billion 100-byte records. The HiBench uses Hadoop RandomWriter and RandomTextWriter applications to produce the inputs for the Sort and WordCount benchmarks and the Hadoop TeraGen application to generate records for the TeraSort.

- K-means workload implements K-means clustering algorithm for knowledge discovery and data mining. The benchmark is taken from the Mahout [35], an Apache open-source machine learning library built on top of Hadoop. The benchmark runs the job iteratively, until different iterations converge or the maximum number of iterations, configured by user, are reached, in order to calculate the centroid of each cluster. As a final step, it runs a clustering job that assigns each sample to a cluster. For generating the workload, HiBench uses a random generator based on statistical distributions.

5.2.2 YourKit Java Profiler

YourKit Java Profiler is one of the industry leading tools for profiling Java applications [36]. It provides advanced CPU and memory analysis of the running applications. It also offers the possibility of finding memory leaks, automatic deadlock detection and resolution of thread synchronization issues.

YourKit Java Profiler offers all its advanced options through a rich and intuitive graphical user interface, while providing some basic functionalities to be used from the command line.
5.3 Testing Environment and Settings

As a testing environment we used GSD (Distributed Systems Group) computers’ cluster of the INESC-ID, a research laboratory associated with the Instituto Superior Tecnico, Lisbon. During the measurements, the environment was dedicated to the executing Hadoop distribution which greatly contributed to the objectivity of the results. The GSD cluster consisted of the 10 physical nodes, located in the same private network. The detailed cluster characteristics are shown in the Table 5.1.

For the evaluation, both the original and our Hadoop implementations were started in the distributed, clustered mode, with the HDFS and MapReduce components spread over the GSD cluster nodes. For the majority of the experiments, the following deployment layout was used:

- For the original Hadoop distribution, the JobTracker and the NameNode were started on the same physical node, three DataNodes and TaskTrackers were started on three different nodes, each running a TaskTracker / DataNode pair.

- The fault tolerant and scalable implementation was running Infinispan nodes in addition. In this deployment, two physical nodes running Infinispan Cache instances were employed, forming an Infinispan cluster and one of them was also running a JobTracker. The NameNode was running on a separate node and TaskTrackers and DataNodes were deployed

| # Nodes | 10 |
| Processor | 8x 4-Core Intel Xeon E5506@2.13GHz |
| Memory | 40GB DDR3-1066 RAM |
| Disk | 1TB 7200rpm SATA HDD |
| Operating System | Ubuntu 12.04.2 x86_64 |
| Network Ethernet | 100 Mbit/s |

Table 5.1: Cluster nodes characteristics

A profile feature that was especially attractive for us was making a CPU and memory snapshots, during application execution, which could support doing offline analysis. The primary goal which the profiler was used for was to analyze the size of the memory occupied by the JobTracker state, stored in the Infinispan.
5.4 JobTracker Fault Tolerance Efficiency

**Goal:** Based on the system requirements, the JobTracker fault tolerance supposes the system must continue functioning normally in case the JobTracker fails, i.e. the JobTracker failover must be automatic and very fast (in order to provide high availability of the system) and the system must continue executing already running jobs correctly and produce the correct final results. Bearing this in mind, we performed this experiment for two reasons: first to show that after killing the JobTracker process the failover will really happen on another node and the JobTracker will continue executing all the running jobs correctly to the end; and next, to measure the time the JobTracker requires to perform complete restart and become operative again.

**Settings:** For this experiment we started two Infinispan nodes, one running the JobTracker and the second used as a new destination for JobTracker upon a failover. The NameNode, DataNodes and TaskTrackers deployment was the same as explained in the Testing Environment and Settings section.

The failover time measured was the time passed from the moment the JobTracker process was killed until Task Trackers detected again the restarted JobTracker and continued to communicate with it. The JobClients re-connection to the JobTracker was not important for this experiment since the JobClients only monitor the jobs executions, and eventually they would re-connect to the JobTracker before the jobs execution finished. The type of the executing job was not important for this experiment, since the execution times of different job types were comparable, which would be shown in the following experiments. Thus, we used a single job type which was Sort job and executed five copies in parallel. For this experiment we modified our code not to remove the finished task objects from the Infinispan storage, so the JobTracker would load the whole state from the storage. We manually killed the JobTracker process in different moments in each experiment iteration.

**Results:** In this experiment, each time we killed the JobTracker process it automatically restarted on another available Infinispan node and continued managing already running jobs to the end. In addition, the running jobs continued their execution on the TaskTrackers during the JobTracker downtime which we induced from the JobClients status reports. We examined the
outputs of the jobs and concluded that they were correct. We repeated the experiment 10 times and each time we got the expected behavior.

The total failover time of the JobTracker, as inferred from our implementation, can be divided into several categories:

- Time that Infinispan DEF framework needs to detect the JobTracker failure and initiate a restart on a new node.

- Time required for JobTracker to load all its data structures from the Infinispan. This time is variable and it depends on the size of the data stored in the Infinispan (number and size of the previously running jobs)

- Time that JobTracker needs to start its internal services and become visible again to the other components

- Fixed (bounded) time that consists of the time TaskTrackers need to detect a failure of the JobTracker and establish a new connection after the restart. We say this time is bounded since it can vary depending on different conditions but it has its maximum value. The conditions include a moment the TaskTracker detects the failure, whether the Infinispan node the TaskTracker was connected to also died with the JobTracker or not and the number of retries and sleep time that the TaskTracker is configured with. As the obtained result we present the maximum time for this process.

The times explained above and their values obtained in the experiment are shown in the Table 5.2.

**Table 5.2: JobTracker failover time distribution**

<table>
<thead>
<tr>
<th>Time category</th>
<th>Measured time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF detects a failover</td>
<td>1</td>
</tr>
<tr>
<td>JobTracker loads the state</td>
<td>1</td>
</tr>
<tr>
<td>JobTracker starts internal services</td>
<td>7</td>
</tr>
<tr>
<td>TaskTracker re-connect</td>
<td>43 (expected 23)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>52 (32)</strong></td>
</tr>
</tbody>
</table>

**Discussion:** From the presented table we can see that the DEF master detects the JobTracker very fast and it needs approximately one second for that. This behavior was expected
CHAPTER 5. EVALUATION

since the DEF master holds a remote reference to the running JobTracker and a fault detection is just a matter of throwing and catching a remote exception (assuming the network transport is reliable). For loading the full state from Infinispan into internal data structures, when executing 5 Sort jobs, the JobTracker needs only one second, which is a result of the Infinispan in-memory data storing. For starting its internal services, JobTracker needs 7 seconds, and this time we can also consider as a fixed time, since it does not depend on the size of the stored state or any other variable. The TaskTracker re-connect time shown includes the worst case maximum time that amounts 43 seconds and the expected maximum time which is 23 seconds. We considered the case when the TaskTracker heartbeat interval was 3 seconds, the TaskTracker was configured with the 10 retries when a connection problem with JobTracker occurred and the TaskTracker thread slept 1 second after each retry. Moreover, we assumed that the Infinispan node the TaskTracker was connected to also died with the JobTracker, thus, the TaskTracker had to re-connect to a new Infinispan node first. The retry period for connection to the Infinispan node was also 10 seconds with 1 second sleep between. The worst case scenario included additional 20 seconds in case there was a temporary network problem with the connection to the new Infinispan node and the JobTracker node. We made the assumption that after these retries, TaskTracker would manage to successfully connect first to a new Infinispan node and then to the JobTracker. The presented maximum time for TaskTracker re-connection can be far smaller, since the number of retries and threads sleep time values can be reduced in the external configuration. Moreover, the common time for this process, proved in numerous tests we performed on the JobTracker failover showed much better results in most of the cases, and amounted to around 10 seconds.

Conclusion: In this experiment we showed that in our fault tolerant solution the total JobTracker failover can be performed in less than a minute, in the worst case. The main reason for such a good result is the fact that JobTracker saves all the state into a persistent storage, thus providing a smooth and instant failover without a need to additionally synchronize its internal state with TaskTrackers or to perform a rollback of the state. This represents the first trade off in our solution, the one between the longer total execution time in case of absence of a failure and a small failover time in case of a crash.
5.5 System Performance

Another requirement set for our solution was that the JobTracker fault tolerance must be achieved in an efficient way with an acceptable impact on the system performance. As presented in the Architecture and Implementation chapters, our solution for a fault tolerant JobTracker is based on regular checkpointing of the JobTracker state to Infinispan. To recollect, Infinispan stores all the objects in memory. Thus, we need to evaluate our solution's performance in terms of identifying potential costs and overhead of the additional checkpointing. Regarding this, we identified two potential problems that could affect the system performances. The first is the time overhead that is caused by the additional frequent operations of the JobTracker, i.e. regular checkpointing into Infinispan. The second is a potential memory overload on the Infinispan physical nodes caused by the Java objects that represent JobTracker state.

5.5.1 State saving Overhead

The main overhead in our solution for the JobTracker fault tolerance represents the state that is required to be saved in the Infinispan in order to provide JobTracker smooth failover, without any jobs lost. Each time the JobTracker internal data structure is changed, a write operation is performed to the storage. The goal of this experiment is to show how big this overhead is, by comparing the execution time of the original Hadoop MapReduce distribution with our fault tolerant implementation. We performed two types of experiments, first executing a single job of each of the several job types and the second executing a single job type and varying the number of its job instances executing in parallel. All the experiments were performed in a situation when no JobTracker failure occurred, i.e. we ran the benchmark MapReduce applications to the end, without killing the JobTracker process.

In order to get more accurate results as well as to eliminate corner cases we repeated each particular experiment 10 times and the numbers presented in the graphs are an average of all the executions’ results.

5.5.1.1 Single Job Variations

Goal: For the first experiment we executed four different jobs from the HiBench benchmark suite on both distributions and compared their execution times. All jobs were executed alone
CHAPTER 5. EVALUATION

<table>
<thead>
<tr>
<th>Job name</th>
<th>Input data size</th>
<th>Run settings</th>
</tr>
</thead>
</table>
| Wordcount | data size 3.2 GB | number of maps: 45  
|          |                 | number of reduces: 70 |
| Sort     | 2.4 GB          | number of maps: 45  
|          |                 | number of reduces: 70 |
| Tera sort| 100MB           | number of maps: 180  
|          |                 | number of reduces: 76 |
| K-means  | Num of clusters: 10  
|          | Num of samples: 20000000  
|          | Samples per input file: 40000000  
|          | Dimensions: 10  
|          | Max iterations: 5 | number of maps: 300  
|          |                 | number of reduces: 5 |

Table 5.3: Testing applications settings

in the dedicated environment. The goal was to show the behavior of the solution when varying MapReduce jobs with different characteristics and show the time deviation when comparing it with the official distribution.

**Settings:** For the purpose of this experiment we used the deployment strategy explained in the Testing Environment and Settings section. To utilize the cluster nodes capacities, the worker processes, TaskTracker and DataNodes were granted 7.5 GB of Java heap space each and each TaskTracker child (task) process got 200MB of memory. We set up TaskTrackers to run maximum 30 map and 30 reduce tasks.

Each of the used MapReduce applications consisted of two execution phases: one that prepared the input data for the benchmark, which execution time we did not measure, and the run phase which actually executed the benchmark. The settings for both phases and size of the job inputs for all the executing jobs are shown in the Table 5.3.

**Discussion:** The results of the experiment are shown in the Figure 5.1. From the picture we can see that the original distribution shows better performances then the fault tolerant implementation, in terms of the execution time, and the difference represents the state saving overhead. The overhead was calculated as a percentage of the original time that needed to be added in order to provide fault tolerance. This is shown in the formula 5.1.
5.5. SYSTEM PERFORMANCE

Figure 5.1: Single job execution time comparison

\[ \text{Overhead} = \frac{\text{execution\_time\_fault\_tolerant} - \text{execution\_time\_original}}{\text{execution\_time\_original}} \times 100\% \quad (5.1) \]

The overhead is the lowest in the Sort application and amounts to 1.82 percent. The job itself took about 2.5 minutes for both implementations. The highest overhead was found in the K-means application and amounted to 16.69 percent. The K-means job was the longest job and took about 10 minutes in total for the original solution and 12 minutes for the fault tolerant implementation. The overheads for all the jobs with the respective execution times are summarized in the Table 5.4.

From the previous summary and the experiment settings shown before, we can conclude that the time overhead is directly proportional to the job execution time and the number of tasks processed during the job execution. This can be explained by the growing number of JobTracker task update operations sent to Infinispan and performed on every heartbeat received from the TaskTrackers. For small jobs, the overhead is negligible, as in the case of the Sort and Wordcount.
### 5.5.1.2 Multiple parallel Jobs

**Goal:** For this experiment we chose one specific MapReduce application and varied the number of the copies of the application executing in parallel. We wanted to show how our solution behaved in the situation when dealing with multiple jobs, which represented a real world use case. In particular we present how the execution time changed (converged) over time in both implementations. From the previous experiment we inferred that the sort job created the smallest deviation between the execution times in two implementations, and decided to use it as representative for this experiment to show how its overhead grows when adding more parallel jobs.

**Settings:** We used the same settings as in the previous experiment for the sort job. We executed five tests, varying the number of parallel running sort jobs from 1 to 5.

The JobTracker was using the default job scheduler, JobQueueTaskScheduler, which keeps jobs in a queue in priority order (FIFO by default) (11). JobQueueTaskScheduler holds two separate groups of tasks, for map tasks and for reduce tasks. For each particular group it first schedules all the tasks of the job with the top position in the priority queue and only then goes to the next job. Still, providing enough map and reduce slots in total in the involved TaskTrackers, we obtained a satisfactory level of job parallelization.

**Discussion:** In the Figure 5.2 the parallel execution of the multiple sort jobs is presented. From the figure we can get a clear picture about the trend of the execution time when increasing the number of MapReduce jobs in the system. Although the total execution time increases for both distributions when the number of jobs arises, we can see that the difference between the two

<table>
<thead>
<tr>
<th>Job name</th>
<th>Execution time (s)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>139.2</td>
<td>1.82</td>
</tr>
<tr>
<td>Wordcount</td>
<td>181.8</td>
<td>2.48</td>
</tr>
<tr>
<td>Terasort</td>
<td>307.2</td>
<td>9.83</td>
</tr>
<tr>
<td>K-means</td>
<td>733.62</td>
<td>16.69</td>
</tr>
</tbody>
</table>

*Table 5.4: Summary of the jobs’ executions time and overhead*
times slightly increases when the number of parallel jobs grows. As in the previous experiment, this can be explained by the number of the tasks update operations performed on the storage which is in this case augmented with the updates performed on the running jobs data structures, whose frequency increases when the number of parallel running jobs grows.

5.5.2 Memory Consumption

The memory overhead which results from the MapReduce fault tolerance is defined by the memory size of the state the JobTracker stores in the Infinispan. Since Infinispan stores all the objects in memory, we were able to make a snapshot of the memory and calculate the total size of the objects the JobTracker produces and stores into Infinispan.

Goal: The goal of this experiment is to show how the size of the memory that JobTracker state occupies on Infinispan nodes varies during jobs execution lifetime. This is important in order to identified potential memory overloads that could cause system unavailability and even a loss of the currently executing jobs.
**Settings:** For this experiment we utilized YourKit Java Profiler to measure memory consumption. Since the GSD Linux machines we used for performing the experiments could not support the GUI version of the YourKit profiler and we could not access the clustered Hadoop applications from the remote computers due to security reasons, we used the snapshot function of the profiler and analyzed these snapshots offline. The deployment layout for this experiment was the same as for the state saving overhead experiments except that we started one additional Infinispan Cache instance on a separate physical node which served only for storing the JobTracker state. This was done in order to make it easier to extract objects of interest from all the Java objects located in the JVM heap. We started the Cache instance process with the YourKit Java Profile agent process attached, to be able to make occasional snapshots of the memory. Again, as a representative for this benchmark, we used the Sort job and varied the number of parallel executing copies from 1 to 5. During the execution of the job(s) we performed snapshots every 30 seconds and analyzed these snapshots offline. In addition, after a job finished, we continued making snapshots for 3 more minutes to see how the memory consumption changed.

![Figure 5.3: Memory consumption over the time for varying number of parallel jobs](image)

**Discussion:** In the Figure 5.3 lines 1 to 5 represent the change of the state size in Infinispan through the time, for the respective number of parallel jobs. From the diagram we can infer that the size of the stored state for all the executions varies over the time and reaches its maximum at about half of the jobs execution lifecycle. The memory consumption is bounded between 2 to 16 MBs for all the executions. The leaps of the memory consumption, shown in the graph are caused by the generation of new tasks during the jobs’ lifecycles, when their ID objects are added to the multiple data structures in the Infinispan. The memory falls shows that the JobTracker removes these objects from the storage after the tasks finish. We can observe the steep slope of the memory consumption at the end of each execution, where the state size falls to around
2 MBs. This is because the jobs’ executions are completed in that time and the job and task objects itself are removed from the storage. In the Figure 5.4 we show the maximum memory consumption reached during the jobs execution lifecycle for all the executions showed before.

![Figure 5.4: Maximum memory consumption for varying number of parallel jobs](image)

In the Figure 5.4 we see that the memory consumption maximums go from 8 MBs for one executing job to 16 MBs when executing five jobs in parallel. From the presented logarithmic trend we can induce that the maximum consumption growth is smaller when increasing the number of running jobs in parallel, which makes us confident that we will not get a memory overflow when running additional parallel jobs.

In case we get some degradation when performing more comprehensive tests, we can somewhat rely on the distribution mode of Infinispan, which partitions and distributes the data across the cluster nodes and allows it to scale almost infinitely by adding additional hardware.
5.6 JobTracker Scalability

**Goal:** The last requirement imposed on our solution is that the system must allow JobTracker functionalities to scale, in case when the number of running jobs rapidly grows. As we explained in the Architecture chapter, we achieve JobTracker scalability by allowing multiple JobTrackers to run in parallel in the cluster and distributing jobs and TaskTrackers amongst these JobTrackers. In this test we wanted first to prove the correctness of our design implementation. Since we could not reach the bottleneck of the JobTracker scalability, due to testing environment hardware limitations, the second thing we decided to do in this test was to compare the performance of the scalable solution with the original distribution using a small set of running jobs, in order to see if we could get any performance improvements.

**Settings:** We took a practical scenario where we have limited resources (in this case physical nodes) and measured two different cases: one where we utilized a resource to start a TaskTracker and the second where we utilized the resource to start the second JobTracker. We used four physical nodes for JobTracker(s) and TaskTrackers. In the first case, we started one JobTracker and three TaskTrackers, each of them on a dedicated node and varied the number of parallel running jobs from 2 to 5. We performed this case on both the fault tolerant MapReduce implementation and the original distribution. In the second case, we started two JobTrackers and two TaskTrackers, each of them running on its dedicated node and again varied the number of jobs in the same way. We tested this scenario only on the fault tolerant MapReduce implementation. For both scenarios, we ran NameNode on a separate physical node and one DataNode together with each TaskTracker on the same physical node. We also ran one Infinispan instance per JobTracker on the same physical node. For the experiment we used Sort job and measured the total execution time.

**Results:** We performed the experiment 10 times for each variable number of parallel running jobs. In our scalability solution, each re-run produced the same expected behavior; the submitted jobs were distributed amongst the two running JobTrackers and each of the two TaskTrackers was assigned to one of the running JobTrackers. The results obtained from the experiments are shown in the Figure 5.5.

**Discussion:** In the Figure 5.5, with the blue line we showed our solution running with two parallel JobTrackers while the yellow and the green line represent the execution of the original
5.6. JOBTRACKER SCALABILITY

Figure 5.5: Performance comparison between scalable solution and original implementation

Hadoop and our solution with a single JobTracker, respectively. On the diagram we can observe that the multiple JobTrackers deployment produces noticeably better performances than original distribution when executing two and four jobs in parallel and reduces the execution time for 24\% and 16\% respectively. On the other hand, for the execution of three parallel jobs we obtained worse results and an execution time similar to the single JobTracker fault tolerant deployment. The negative difference is caused by the state saving overhead of our solution.

The optimization obtained in scalable deployment can be explained by the fact that Hadoop MapReduce uses the JobQueueTaskScheduler by default, which schedules the submitted jobs one by one, waiting for previous jobs to complete in order to start the next one. This behavior is not acceptable for many jobs, when executing in parallel with other jobs. Using multiple JobTrackers we achieved a better level of parallelization by instantly scheduling the jobs which would otherwise wait for the previous jobs to complete.

In the first measurement we started two jobs in parallel. The original MapReduce schedules the tasks of the first job to the three running TaskTrackers and waits to schedule the second
job until all the first job tasks are scheduled and until it gets a notification from a TaskTracker that has free slots. Even in a case where the three TaskTrackers have enough free slots in total to execute all the tasks of the first job in a single pass, the second job would have to wait for scheduling all of the tasks of the first job. In our solution, when using two JobTrackers, each of the two jobs can run immediately, since the JobTrackers schedule these jobs independently to their dedicated TaskTrackers, thus providing total parallelization of the running jobs. In the second measurement, we ran three jobs in parallel. In our solution, one job is submitted to one JobTracker and the other two jobs to the second JobTracker. Since the first job is finished quickly, the total execution time depends on the time for executing other two jobs on a single JobTracker using only one TaskTracker. (As we recall from the previous chapter, the first TaskTracker will not join the second JobTracker after finishing its job, since it is the only one assigned to the that JobTracker and thus it is a permanent assignment). As we can observe from the diagram, the performances are similar to the case when we have three jobs running on a single JobTracker using three TaskTrackers (green line on the diagram). The execution of four and five parallel jobs show the similar pattern to the two and three jobs execution, respectively.

**Conclusion:** This experiment uses a specific deployment configuration and a limited number of jobs to show that some optimization is obtained when using multiple JobTrackers in the system. The results depend largely on the concrete deployment layout and the ratio of the total number of running tasks and the configured number of map and reduce slots of the available TaskTrackers. Nevertheless, we believe that the pattern and the scheduling optimization presented above could be achieved in other different cases when appropriate deployment settings are chosen.

### 5.7 Conclusion

In this chapter we showed the experiments we performed in accordance with the main goal and the related requirements imposed on our solution and discussed the obtained results. Each particular experiment resulted from specific system requirements.

We proved that we achieved the JobTracker fault tolerance that provides the correct behaviour of the system after the JobTracker failure happens. The system performs an automatic failover of the JobTracker on some other available node and during the JobTracker downtime the running jobs continue their execution. Moreover, the results of the running jobs are correct
5.7. CONCLUSION

in all experiment iterations. The failover time is variable and mainly depends on the moment of time when TaskTrackers spot the JobTracker failure. Still, even in the worst case scenario, this time is acceptable.

The state saving overhead mainly depends on the total jobs’ execution time and the number of the generated tasks. For simple jobs, the overhead is almost negligible, while it slightly increases when executing more complex jobs. For long running jobs, we can expect this overhead can make a more significant impact on the system performances. The state saving overhead does not depend on the size of the Infinispan cluster, since we use the asynchronous replication of the state and the state is replicated in the background while the framework continues with its operations.

The memory of the Infinispan nodes, consumed by the JobTracker state, is bounded during a particular job execution and shows a logarithmic growth when increasing the number of running jobs in parallel. Moreover, the memory empties all the time during jobs’ execution lifetime and the system totally removes all the job’s objects after they are finished. Thus, we are optimistic about the memory consumption in a case when executing large number of complex jobs and believe that the memory overload will not happen.

We showed some performance improvements we obtained using our scalability solution with a specific deployment configuration which, we believe, can be projected to other different cases. The improvements are based on the fact that our solution provides better scheduling of jobs than the default Hadoop FIFO scheduler, and it depends on the number of generated tasks / number of available slots ratio.

In the following chapters we will explain the improvements we are planning to make on our solution in the future and draw the final conclusions of the thesis.
Conclusion and Future Work

The Hadoop MapReduce framework is an important representative of the large scale processing distributed platforms, which consider the master-slave architectural model. The MapReduce JobTracker represents the master component of the system and it runs on a single physical node. As such, it represents a single point of failure of the system; when it fails, all the currently running user jobs are lost and the system becomes unavailable. In addition, when running a large number of complex jobs, the JobTracker scalability can become an issue. Although plenty of work has been done on these issues, there is still no solution efficient enough for the Hadoop community to include into their official release. In this thesis we proposed, implemented and evaluated a design that solved the JobTracker fault tolerance and scalability problems in a way that did not significantly affect the overall system performances.

The proposed design is based on saving the JobTracker state into the Infinispan data store. Infinispan handles the state replication automatically by copying the state to the other cluster nodes. The JobTracker is started on the Infinispan DEF framework as a distributed task, and in case the node where it runs fails, the framework performs the JobTracker automatic failover on some other, healthy node. Upon a restart, the JobTracker loads the stored state and continues with the normal operation. During the failover, TaskTrackers can continue executing the running jobs normally, and the JobClients and TaskTrackers automatically re-connect to the JobTracker upon the restart.

To provide the JobTracker scalability, the solution supports running multiple JobTrackers in the system in parallel. A simple load balancing algorithm is implemented on the JobClient side, which distributes users’ jobs amongst the JobTrackers currently running in the system. In the same time, TaskTrackers are distributed amongst the JobTrackers and each TaskTracker can be also re-assigned to another JobTracker in case the current JobTrackers load distribution requires that.

In order to fully support a custom failover of the JobTracker, minor improvements to the
Infinispan DEF framework have been made and thus they can be applied to all other DEF distributed tasks.

The implemented solution is fully functional and provides the JobTracker automatic failover in case of a failure, with minimum system downtime and no data/jobs loss. During the JobTracker downtime, all the currently running jobs can continue their execution and users can continue submitting new jobs to the framework. The solution can tolerate multiple JobTracker failures without a need for an administrator’s intervention. JobTracker scalability is provided by simply starting a new JobTracker instance and all the jobs load balancing and TaskTracker re-distribution is handled automatically by the implementation.

The main drawback of the presented solution is the necessity of running one spare Infinispan node in order to provide proper behavior of the system, while the system is fully utilized after the first JobTracker failure happens. The second flaw of the solution is based on the shortcomings of the Infinispan DEF master, which is not fault tolerant.

The evaluation showed that the solution does not make significant a impact on the system performance. Memory consumption on the Infinispan nodes is controlled by the implementation which constantly removes finished parts of a job during its execution. The state saving overhead is negligible when running simple Hadoop jobs and grows slowly when increasing the number and complexity of jobs. However when executing long running complex jobs we expect a more significant performance degradation. The scalability part of the solution showed performance improvements under specific deployment settings when compared to the original distribution, which is mainly the result of the improved jobs scheduling.

### 6.1 Future work

Several improvements on our solution have been planned for a future work.

We want to perform a further optimization of the process of saving the JobTracker state into the Infinispan. That would include the choice of a more efficient Java object serializer instead of the current Java Serializable interface. Since we currently store almost all JobTracker data structures, next, we would focus on optimizing the amount and type of the data we store and try to store as much as possible simple Java data types that do not need additional serialization / de-serialization.
6.1. FUTURE WORK

The second improvement would consider a work on the Infinispan DEF master node fault
tolerance, which is not provided in the current Infinispan implementation. Work on this issue is
both important for Infinispan community and our fault tolerant JobTracker implementation.

Next, in order to fully incorporate our solution into the current Hadoop release, we still need
to update the Hadoop MapReduce administration web interface, so we can track statistics such
as the number of JobTracker restarts, JobTrackers current physical addresses, number of the
JobTrackers running on a particular node and the total number running in the cluster.

Finally, the last improvement includes an automatic replication of MapReduce and Infinispan
configuration files to all the physical nodes running in the cluster. This can be useful, since all
the configuration files have to be identical and currently we need to change them manually on
all the cluster nodes when we want to update a particular setting. Still, this is a minor issue
since these files are not intended to be changed frequently.
References


[2] Douglas Thain, Todd Tannenbaum and Miron Livny, Distributed Computing in Practice: The Condor Experience, Wiley Online Library, 2005


Apache Hadoop 0.23, http://hadoop.apache.org/docs/r0.23.9, [Online; accessed 23-August-2013]


Feng Wang et al., Hadoop High Availability through Metadata Replication, CloudDB’09, November 2, 2009

High Availability: No Single Points of Failure, MapR white paper, 2011

Dhruba Borthakur et al., Apache Hadoop Goes Realtime on Facebook, SIGMOD ’11, June 12-16, 2011
6.1. FUTURE WORK


[28] *Infinispan clustering modes*, [https://docs.jboss.org/author/display/ISPN/Clustering+modes](https://docs.jboss.org/author/display/ISPN/Clustering+modes) [Online; accessed 23-August-2013]

[29] *Infinispan transactions*, [https://docs.jboss.org/author/display/ISPN/Infinispan+transactions](https://docs.jboss.org/author/display/ISPN/Infinispan+transactions) [Online; accessed 23-August-2013]


[31] *Infinispan Marshalling*, [https://docs.jboss.org/author/display/ISPN/Marshalling](https://docs.jboss.org/author/display/ISPN/Marshalling) [Online; accessed 23-August-2013]


[33] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie and Bo Huang, *The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis*, ICDE Workshops, 2010

[34] *HiBench source code*, [https://github.com/hibench/HiBench-2.1](https://github.com/hibench/HiBench-2.1) [Online; accessed 23-August-2013]


[38] *Cassandra - A Decentralized Structured Storage System*, Avinash Lakshman, Prashant Malik, ACM SIGOPS, April 2010

[40] Distributed caching with memcached, Brad Fitzpatrick, Linux Journal, August 2004