Implementing MapReduce Applications in Dynamic Cloud Environments

Fabrizio Marozzo, Domenico Talia and Paolo Trunfio

Abstract MapReduce is one of the most popular programming models for parallel data processing in Cloud environments. Standard MapReduce implementations are based on centralized master-slave architectures that do not cope well with dynamic Cloud environments in which nodes may join and leave the network at high rates. In this chapter we describe P2P-MapReduce, a framework that exploits a Peer-to-Peer (P2P) model to manage intermittent nodes participation, master failures and MapReduce job recovery in a decentralized but effective way. Specifically, the chapter describes the P2P-MapReduce architecture, mechanisms and implementation, and provides an evaluation of its performance. The performance results confirm that P2P-MapReduce ensures a higher level of fault tolerance compared to a centralized implementation of MapReduce.

1 Introduction

Clouds are used as effective computing platforms to face the challenge of extracting knowledge from big data repositories, as well as to provide efficient data analysis environments to both researchers and companies [1]. A key point for the effective implementation of data analysis environments on Cloud platforms is the availability of programming models that support a wide range of applications and system scenarios [2]. One of the most popular programming models adopted for the implementation of data-intensive Cloud applications is MapReduce [3].

Fabrizio Marozzo
DIMES, University of Calabria, Italy, e-mail: fmarozzo@dimes.unical.it

Domenico Talia
DIMES, University of Calabria, Italy, e-mail: talia@dimes.unical.it

Paolo Trunfio
DIMES, University of Calabria, Italy, e-mail: trunfio@dimes.unical.it

Since its introduction by Google, MapReduce has proven to be applicable to many domains, including machine learning and data mining, log file analysis, financial analysis, scientific simulation, image retrieval and processing, blog crawling, machine translation, language modelling, and bioinformatics. It is widely recognized as one of the most important programming models for Cloud environments, being it supported by leading providers such as Amazon, with its Elastic MapReduce service[1], and Google itself, which released a MapReduce API for its App Engine[2].

MapReduce defines a framework for processing large data sets in a highly parallel way by exploiting computing facilities available in a large cluster or through a Cloud system. Users specify the computation in terms of a map function that processes a key/value pair to generate a list of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Standard MapReduce implementations (e.g., Googles MapReduce [4] and Apache Hadoop [5]) are based on a master-slave model. A job is submitted by a user node to a master node that selects idle workers and assigns each one a map or a reduce task. When all map and reduce tasks have been completed, the master node returns the result to the user node. The failure of a worker is managed by re-executing its task on another worker, while standard MapReduce implementations do not handle with master failures as designers consider failures unlikely in large clusters or in reliable Cloud environments.

On the contrary, node failures - including master failures - can occur in large clusters and are likely to happen in dynamic Cloud environments like a Cloud of clouds, which can be formed by a large number of computing nodes that join and leave the network at very high rates. Therefore, providing effective mechanisms to manage master failures is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in large dynamic Cloud environments where standard MapReduce implementations could be unreliable.

P2P-MapReduce [6] exploits a peer-to-peer model to manage node churn, master failures, and job recovery in a decentralized but effective way, so as to provide a more reliable MapReduce middleware that can be effectively exploited in dynamic Cloud infrastructures. This chapter describes the P2P-MapReduce architecture, mechanisms and implementation, and provides an evaluation of its performance. The performance results confirm that P2P-MapReduce ensures a higher level of fault tolerance compared to a centralized implementation of MapReduce.

The remainder of the chapter is organized as follows. Section 2 provides a background on MapReduce. Section 3 describes the P2P-MapReduce architecture. Section 4 discusses the fault tolerance mechanisms used in P2P-MapReduce. Section 5 describes how the system has been implemented. Section 6 presents an evaluation of its performance. Finally, Section 7 concludes the chapter.

[1] http://aws.amazon.com/emr/

[2] https://cloud.google.com/appengine/docs/java/dataprocessing/

2 MapReduce Background

This section describes the operations performed by a generic MapReduce application to transform input data into output data according to the standard master-slave model, and discusses some popular MapReduce frameworks.

MapReduce Users define a *map* and a *reduce* function [3]. The *map* function processes a (key, value) pair and returns a list of intermediate (key, value) pairs:

map (k1,v1) $\rightarrow$ list(k2,v2).

The *reduce* function merges all intermediate values having the same intermediate key:

reduce (k2, list(v2)) $\rightarrow$ list(v2).

The whole transformation process can be described through the following steps (see Fig. 1):

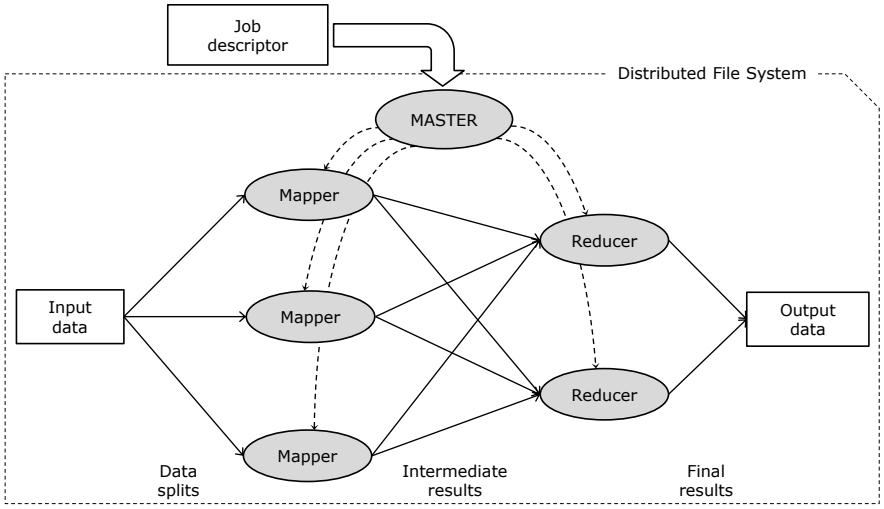

Fig. 1 Execution phases in a generic MapReduce application

1. A master process receives a job descriptor that specifies the MapReduce job to be executed. The job descriptor contains, among other information, the location of the input data, which may be accessed using a distributed file system or an HTTP/FTP server.
2. According to the job descriptor, the master starts a number of mapper and reducer processes on different machines. At the same time, it starts a process that reads the input data from its location, partitions that data into a set of splits, and distributes those splits to the various mappers.

3. After receiving its data partition, each mapper process executes the *map* function (provided as part of the job descriptor) to generate a list of intermediate key/value pairs. Those pairs are then grouped on the basis of their keys.
4. All pairs with the same keys are assigned to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the job descriptor) which merges all the values associated to the same key to generate a possibly smaller set of values.
5. The results generated by each reducer process are then collected and delivered to a location specified by the job descriptor, so as to form the final output data.

Several applications of MapReduce have been demonstrated, including: performing a distributed grep; counting URL access frequency; building a reverse Web-link graph; building a term-vector per host; building inverted indexes, performing a distributed sort. Ref. [5] mentions many significant types of applications implemented exploiting the MapReduce model, including: machine learning and data mining, log file analysis, financial analysis, scientific simulation, image retrieval and processing, blog crawling, machine translation, language modelling, and bioinformatics.

Besides the original MapReduce implementation by Google [4], several other MapReduce implementations have been realized within other systems, including Hadoop [5], GridGain [7], Skynet [8], MapSharp [9] and Disco [10]. Another system sharing most of the design principles of MapReduce is Sector/Sphere [11], which has been designed to support distributed data storage and processing over large Cloud systems. Sector is a high-performance distributed file system; Sphere is a parallel data processing engine used to process Sector data files. Some other works focused on providing more efficient implementations of MapReduce components, such as the scheduler [12] and the I/O system [13], while others focused on adapting the MapReduce model to specific computing environments, like shared-memory systems [14], volunteer computing environments [15], desktop grids [16], and mobile environments [17].

Even though P2P-MapReduce [18] shares some basic ideas with some of the systems discussed above (in particular, [15] and [17]), it also differs from all of them for its use of a peer-to-peer approach both for job and system management. Indeed, the peer-to-peer mechanisms implemented by P2P-MapReduce allow nodes to dynamically join and leave the network, change state over time, manage nodes and job failures in a way that is completely transparent both to users and applications.

3 P2P-MapReduce Architecture

The P2P-MapReduce architecture includes three types of nodes, as shown in Figure 2: *user*, *master* and *slave*. Computing nodes are dynamically assigned the master or the slave role, thus the sets of master and slave nodes change their composition over time, as discussed later. User nodes submit their MapReduce *jobs*, composed by multiple map/reduce *tasks*, through one of the available masters. The choice of the master to which submit the job may be done on the basis of the current workload

of the available masters, i.e., the user may choose the master that is managing the lowest number of jobs.

Master nodes are at the core of the system. They perform three types of operations: management, recovery and coordination. Management operations are those performed by masters that are acting as the *primary master* for one or more jobs. Recovery operations are executed by masters that are acting as *backup master* for one or more jobs. Coordination operations are performed by the master that is acting as the network *coordinator*. The coordinator has the power of changing slaves into masters, and vice versa, so as to keep the desired master/slave ratio.

Each slave executes the tasks that are assigned to it by one or more primary masters. Task assignment may follow various policies, based on current workload, highest reliability, and so on. In our implementation tasks are assigned to the slaves with the lowest workload, i.e., with the lowest number of assigned tasks. Jobs and tasks are managed by processes called *Job Managers* and *Task Managers*, respectively. Each primary master runs one Job Manager thread per managed job, while each slave runs one Task Manager thread per managed task. Moreover, masters use a *Backup Job Manager* for each job they are responsible for as backup masters.

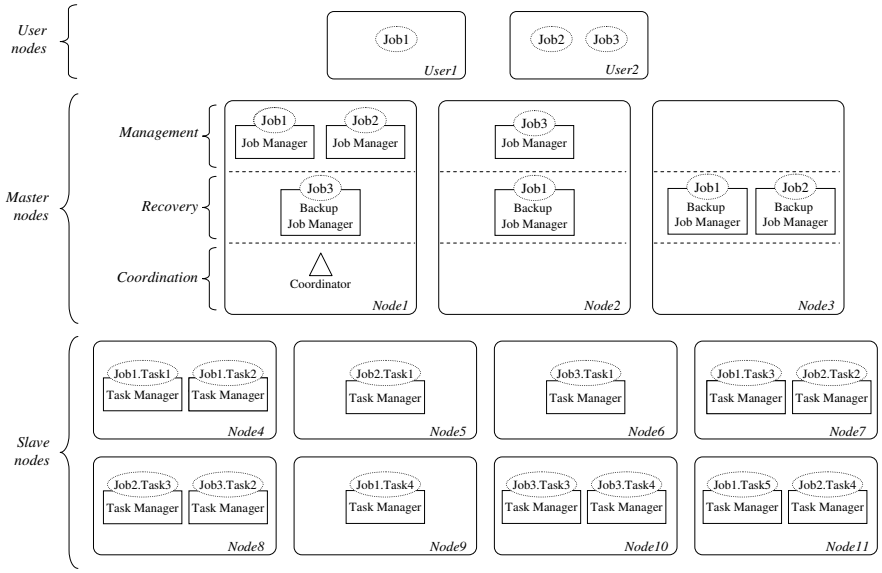

Fig. 2 Architecture of P2P-MapReduce.

Figure 2 shows an example scenario in which three jobs have been submitted: one job by *User1* (*Job1*) and two jobs by *User2* (*Job2* and *Job3*). Focusing on *Job1*, *Node1* is the primary master, and two backup masters are used (*Node2* and *Node3*). *Job1* is composed by five tasks: two of them are assigned to *Node4*, and one each to *Node7*, *Node9* and *Node11*.

If the primary master *Node1* fails before the completion of *Job1*, the following recovery procedure takes place:

- Backup masters *Node2* and *Node3* detect the failure of *Node1* and start a distributed procedure to elect the new primary master among them.
- Assuming that *Node3* is elected as the new primary master, *Node2* continues to play the backup function and, to keep the desired number of backup masters active (two, in this example), another backup node is chosen by *Node3*. Then, *Node3* binds to the connections that were previously associated to *Node1*, and proceeds to manage the job using its local replica of the job state.

As soon as the job is completed, the (new) primary master notifies the result to the user node that submitted the managed job.

4 System Mechanisms

The behavior of a generic node is modeled as a state diagram that defines the different states a node can assume, and all the events that determine the transitions from a state to another state. Figure 3 shows such state diagram modeled using the UML State Diagram formalism.

The state diagram includes two macro-states, `SLAVE` and `MASTER`, which describe the two roles that can be assumed by each node. The `SLAVE` macro-state has three states, `IDLE`, `CHECK_MASTER` and `ACTIVE`, which represent respectively: a slave waiting for task assignment; a slave checking the existence of at least one master in the network; a slave executing one or more tasks. The `MASTER` macro-state is modelled with three parallel macro-states, which represent the different roles a master can perform concurrently: possibly acting as the primary master for one or more jobs (`MANAGEMENT`); possibly acting as a backup master for one or more jobs (`RECOVERY`); coordinating the network for maintenance purposes (`COORDINATION`).

The `MANAGEMENT` macro-state contains two states: `NOT_PRIMARY`, which represents a master node currently not acting as the primary master for any job, and `PRIMARY`, which, in contrast, represents a master node currently managing at least one job as the primary master. Similarly, the `RECOVERY` macro-state includes two states: `NOT_BACKUP` (the node is not managing any job as backup master) and `BACKUP` (at least one job is currently being backed up on this node). Finally, the `COORDINATION` macro-state includes four states: `NOT_COORDINATOR` (the node is not acting as the coordinator), `COORDINATOR` (the node is acting as the co-

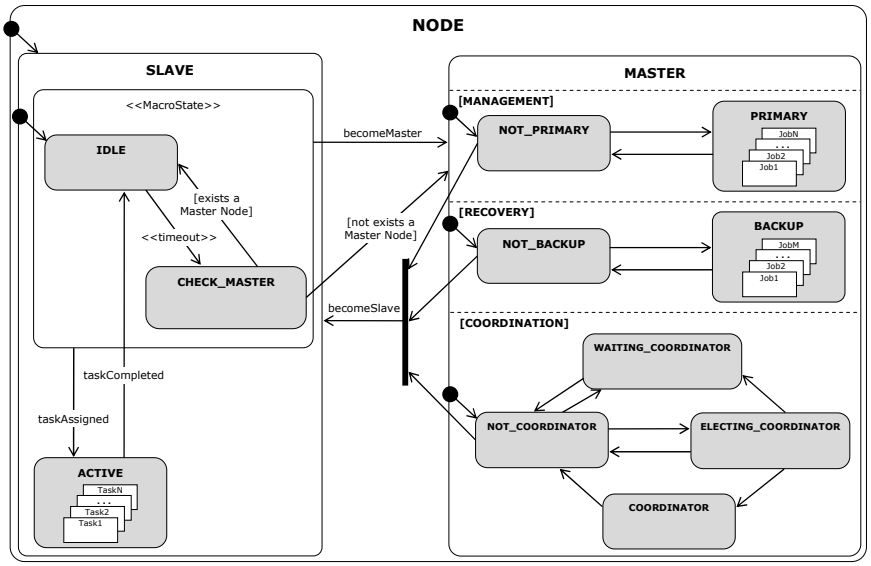

Fig. 3 Behavior of a generic node described by a UML State Diagram.

ordinator), `WAITING_COORDINATOR` and `ELECTING_COORDINATOR` for nodes currently participating to the election of the new coordinator, as specified later.

The combination of the concurrent states [`NOT_PRIMARY`, `NOT_BACKUP`, `NOT_COORDINATOR`] represents the abstract state `MASTER.IDLE`. The transition from master to slave role is allowed only to masters in the `MASTER.IDLE` state. Similarly, the transition from slave to master role is allowed to slaves that are not in `ACTIVE` state.

5 Implementation

We implemented a prototype of the P2P-MapReduce framework using the JXTA framework [19]. JXTA provides a set of XML-based protocols that allow computers and other devices to communicate and collaborate in a peer-to-peer fashion. In JXTA there are two main types of peers: *rendezvous* and *edge*. The rendezvous peers

act as routers in a network, forwarding the discovery requests submitted by edge peers to locate the resources of interest. Peers sharing a common set of interests are organized into a *peer group*. To send messages to each other, JXTA peers use asynchronous communication mechanisms called *pipes*. Pipes can be either point-to-point or multicast, so as to support a wide range of communication schemes. All resources (peers, services, etc.) are described by *advertisements* that are published within the peer group for resource discovery purposes.

All master and slave nodes in the P2P-MapReduce system belong to a single JXTA peer group called *MapReduceGroup*. Most of these nodes are edge peers, but some of them also act as rendezvous peers, in a way that is transparent to the users. Each node exposes its features by publishing an advertisement containing basic information that are useful during the discovery process, such as its role and workload. Each advertisement includes an expiration time; a node must renew its advertisement before expiration; nodes associated with expired advertisements are considered as no longer present in the network.

Each node publishes its advertisement in a local cache and sends some keys identifying that advertisement to a rendezvous peer. The rendezvous peer uses those keys to index the advertisement in a distributed hash table called Shared Resource Distributed Index (SRDI), that is managed by all the rendezvous peers of *MapReduceGroup*. Queries for a given type of resource (e.g., master nodes) are submitted to the JXTA Discovery Service that uses SRDI to locate all the resources of that type without flooding the entire network.

Pipes are the fundamental communication mechanisms of the P2P-MapReduce system, since they allow the asynchronous delivery of event messages among nodes. Different types of pipes are employed within the system: bidirectional pipes are used between users and primary masters to submit jobs and return results, as well as between primary masters and their slaves to submit tasks and receive results notifications, while multicast pipes are used by primary masters to send job updates to their backups.

Figure 4 uses the UML Deployment/Component Diagram formalism to describe the software modules inside each node and how those modules interact with each other in a P2P-MapReduce network.

Each node includes three software modules/layers: *Network*, *Node* and *MapReduce*:

- The Network module is in charge of the interactions with the other nodes by using the pipe communication mechanisms provided by the JXTA framework. When a connection timeout is detected on a pipe associated with a remote node, this module propagates the appropriate failure event to the Node module. Additionally, this module allows the node to interact with the JXTA Discovery Service for publishing its features and for querying the system (e.g., when looking for idle slave nodes).
- The Node module controls the lifecycle of the node in its various aspects, including network maintenance, job management, and so on. Its core is represented by the FSM component which implements the logic of the finite state machine described in Figure 3, steering the behavior of the node in response to inner events

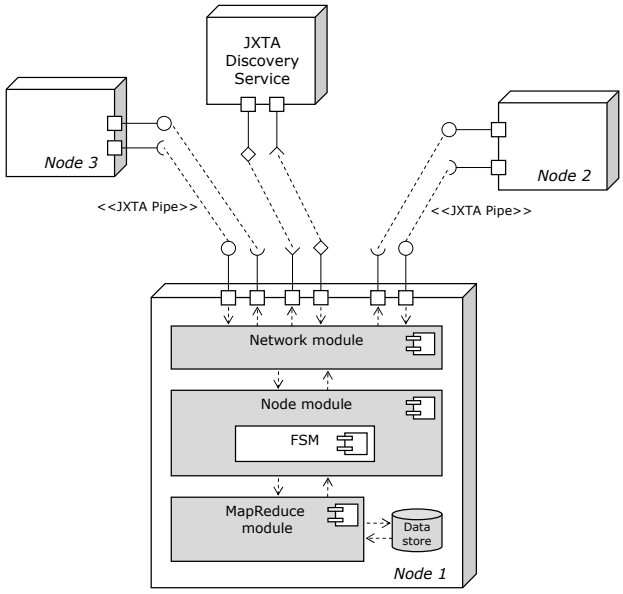

Fig. 4 Software modules inside each node and interactions among nodes.

or messages coming from other nodes (i.e., job assignments, job updates, and so on).

- The MapReduce module manages the local execution of jobs (when the node is acting as a master) or tasks (when the node is acting as a slave). Currently this module is built around the local execution engine of the Hadoop system [5].

6 Evaluation

The evaluation has been carried out by using a custom-made discrete-event simulator that reproduces the behavior of the P2P-MapReduce prototype described in the previous section, as well as the behavior of a centralized MapReduce system that performs the standard operations described in Section 2.

The simulator models joins and leaves of nodes and job submissions as Poisson processes; therefore, the inter-arrival times of all the join, leave and submission events are independent and obey an exponential distribution with a given rate. Table 1 shows the input parameters used during the simulation.

Table 1 Simulation parameters

Symbol	Description	Values
N	Initial number of nodes in the network	10000
NM	Number of masters (% on N)	1 (P2P only)
NB	Number of backup masters per job	1 (P2P only)
LR	Leaving rate: avg. number of nodes that leave the network every minute (% on N)	0.025, 0.05, 0.1, 0.2, 0.4
JR	Joining rate: avg. number of nodes that join the network every minute (% on N)	equal to LR
SR	Submission rate: avg. number of jobs submitted every minute (% on N)	0.01
CT	Avg. computing time of a job (hours)	150
NT	Avg. number of tasks of a job	300

As shown in the table, we simulated MapReduce systems having a size of 10000 nodes, including both slaves and masters. In the centralized implementation, there is one master only and there are not backup nodes. In the P2P implementation, there are 1% masters (out of N) and each job is managed by one master which dynamically replicates the job state on one backup master.

To simulate node churn, a joining rate JR and a leaving rate LR have been defined. On average, every minute JR nodes join the network, while LR nodes abruptly leave the network so as to simulate an event of failure (or a graceless disconnection). In our simulation $JR = LR$ to keep the total number of nodes approximatively constant during the whole simulation. In particular, we used five values for JR and LR: 0.025, 0.05, 0.1, 0.2 and 0.4, so as to evaluate the system under different churn rates. Note that such values are expressed as a percentage of N. For example, if $N = 10000$ and $LR = 0.05$, there are on average 5 nodes leaving the network every minute.

Every minute, SR jobs are submitted on average to the system by user entities. The value of such submission rate is 0.01, expressed, as for JR and LR, as a percentage of N. Each job submitted to the system is characterized by two parameters, total computing time CT and number of tasks NT, whose average values are reported in the table.

For a given submitted job, the system calculates the amount of time that each slave needs to complete the task assigned to it as the ratio between the total computing time and the number of tasks required by that job. Tasks are assigned to the slaves with the lowest workload, i.e., with the lowest number of assigned tasks. Each slave keeps the assigned tasks in a priority queue. After the completion of the current task, the slave selects for execution the task that has failed the highest number of times among those present in the queue.

At the end of the simulation, we collected two main performance indicators:

- The *percentage of failed jobs*, which is the number of jobs failed expressed as a percentage of the total number of jobs submitted.

- The *percentage of lost computing time*, which is the amount of time spent executing tasks that were part of failed jobs, expressed as a percentage of the total computing time.

For the purpose of our evaluation, a "failed" job is a job that does not complete its execution, i.e., does not return a result to the submitting user entity. The failure of a job is always caused by a not-managed failure of the master responsible for that job. The failure of a slave, on the contrary, never causes a failure of the whole job because its task is reassigned to another slave.

Figure 5 compares the P2P and centralized implementations in terms of percentage of failed jobs.

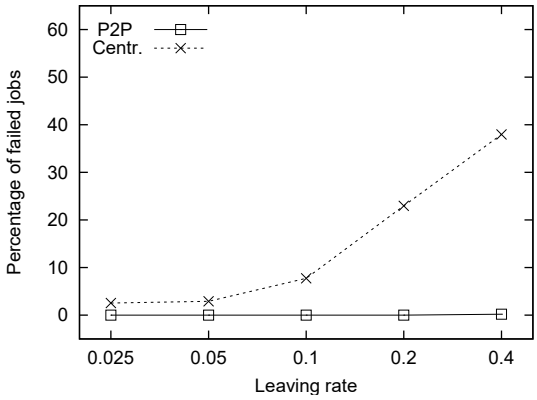

Fig. 5 Percentage of failed jobs

As expected, with the centralized MapReduce implementation the percentage of failed jobs significantly increases with the leaving rate, passing from 2.5% when $LR = 0.025$, to 38.0% when $LR = 0.4$. In contrast to the centralized implementation, the P2P-MapReduce framework is limitedly affected by job failures. In particular, the percentage of failed jobs is 0% for $LR \leq 0.2$, while it is 0.2% for $LR = 0.4$ even if only one backup master per job is used.

Figure 6 reports the percentage of lost computing time in centralized and P2P implementations related to the same experiments of Figure 5, for different leaving rates. The figure also shows the amount of lost computing time, expressed in hours, in correspondence of each graph point for the centralized and P2P cases.

The lost computing time follows a similar trend as the percentage of failed jobs. For example, the percentage of lost computing time for the centralized system passes from 1.9% when $LR = 0.025$ to 24.2% when $LR = 0.4$, while the percentage of time lost by the P2P system is under 0.1% in the same configurations. The difference between centralized and P2P is even clearer if we look at the absolute amount of computing time lost in the various scenarios. In the worst case (LR=0.4), the centralized system loses 29753 hours of computation, while the amount of lost comput-

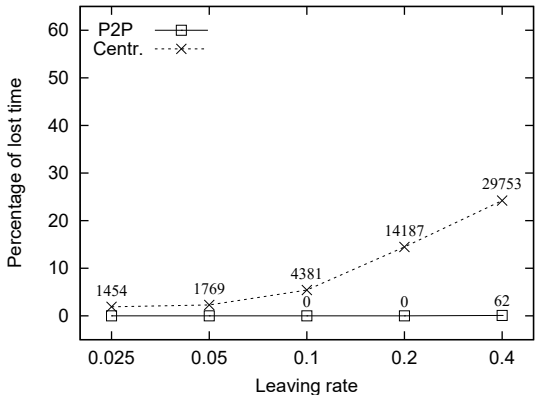

Fig. 6 Percentage of lost time. The numbers in correspondence of each graph point represent the amount of lost computing time expressed in hours.

ing time with the P2P-MapReduce system is only 62 hours. An additional series of simulation results can be found in [6].

7 Conclusions

Providing effective mechanisms to manage master failures, job recovery and intermittent participation of nodes is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in dynamic Cloud environments where current MapReduce implementations may be unreliable.

The P2P-MapReduce model described in this chapter exploits a P2P model to perform job state replication, manage master failures and allow intermittent participation of nodes in a decentralized but effective way. Using a P2P approach, we extended the MapReduce architectural model making it suitable for highly dynamic environments where failure must be managed to avoid a critical loss of computing resources and time.

The performance analysis conducted through simulation confirms that P2P-MapReduce ensures a higher level of fault tolerance compared to a centralized implementation of MapReduce. A prototype of the system is available at the following url: http://gridlab.dimes.unical.it/projects/p2p-mapreduce/.

References

1. D. Talia, P. Trunfio, F. Marozzo, “Data Analysis in the Cloud”. Elsevier, 2015.
2. F. Marozzo, D. Talia, P. Trunfio, “Using clouds for scalable knowledge discovery applications” Lecture Notes in Computer Science, 7640 LNCS, pp. 220-227, 2013.

3. J. Dean, S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. *Communications of the ACM*, 51(1), 107-113, 2008.
4. J. Dean, S. Ghemawat. “MapReduce: Simplified data processing on large clusters”. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI’04), San Francisco, USA, 2004.
5. Hadoop. http://hadoop.apache.org (site visited September 2016).
6. F. Marozzo, D. Talia, P. Trunfio. “P2P-MapReduce: Parallel data processing in dynamic Cloud environments”. *Journal of Computer and System Sciences*, 78(5), 1382-1402, Elsevier Science, 2012.
7. Gridgain. http://www.gridgain.com (site visited September 2016).
8. Skynet. http://skynet.rubyforge.org (site visited September 2016).
9. MapSharp. http://mapsharp.codeplex.com (site visited September 2016).
10. Disco. http://discoproject.org (site visited September 2016).
11. Y. Gu, R. Grossman. “Sector and Sphere: The Design and Implementation of a High Performance Data Cloud”. *Philosophical Transactions, Series A: Mathematical, physical, and engineering sciences*, 367(1897), 2429-2445, 2009.
12. M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, I. Stoica. “Improving MapReduce Performance in Heterogeneous Environments”. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08), San Diego, USA, 2008.
13. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears. “MapReduce Online”. 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI’10), San Jose, USA, 2010.
14. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis. “Evaluating MapReduce for multi-core and multiprocessor systems”. 13th International Symposium on High-Performance Computer Architecture (HPCA’07), Phoenix, USA, 2007.
15. H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, Z. Zhang. “MOON: MapReduce On Opportunistic eNvironments”. 19th International Symposium on High Performance Distributed Computing (HPDC’10), Chicago, USA, 2010.
16. B. Tang, M. Moca, S. Chevalier, H. He, G. Fedak. “Towards MapReduce for Desktop Grid Computing”. 5th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC’10), Fukuoka, Japan, 2010.
17. A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, V. H. Tuulos. “Misco: A MapReduce framework for mobile systems”. 3rd Int. Conference on Pervasive Technologies Related to Assistive Environments (PETRA’10), New York, USA, 2010.
18. F. Marozzo, D. Talia, P. Trunfio, “A Framework for Managing MapReduce Applications in Dynamic Distributed Environments”. Proc. of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011), Ayia Napa, Cyprus, pp. 149–158, 2011.
19. L. Gong. “JXTA: A Network Programming Environment”. *IEEE Internet Computing*, 5(3), 88-95, 2001.