A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments

Fabrizio Marozzo, Domenico Talia and Paolo Trunfio

Abstract MapReduce is a programming model widely used in Cloud computing environments for processing large data sets in a highly parallel way. MapReduce implementations are based on a master-slave model. The failure of a slave is managed by re-assigning its task to another slave, while master failures are not managed by current MapReduce implementations as designers consider failures unlikely in reliable Cloud systems. On the contrary, node failures - including master failures - are likely to happen in dynamic Cloud scenarios, where computing nodes may join and leave the network at an unpredictable rate. Therefore, providing effective mechanisms to manage master failures is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in those dynamic Cloud environments where current MapReduce implementations could be unreliable. The goal of our work is extending the master-slave architecture of current MapReduce implementations to make it more suitable for dynamic Cloud scenarios. In particular, in this chapter we present a P2P-MapReduce 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.

Fabrizio Marozzo
Department of Electronics, Computer Science and Systems (DEIS), University of Calabria, Rende, Italy, e-mail: fmarozzo@deis.unical.it

Domenico Talia
Institute of High Performance Computing and Networking, Italian National Research Council (ICAR-CNR) and Department of Electronics, Computer Science and Systems (DEIS), University of Calabria, Rende, Italy, e-mail: talia@deis.unical.it

Paolo Trunfio
Department of Electronics, Computer Science and Systems (DEIS), University of Calabria, Rende, Italy, e-mail: trunfio@deis.unical.it

1 Introduction

Cloud computing is gaining increasing interest both in science and industry for its promise to deliver service-oriented remote access to hardware and software facilities in a highly reliable and transparent way. A key point for the effective implementation of large-scale Cloud systems is the availability of programming models that support a wide range of applications and system scenarios. One of the most successful programming models currently adopted for the implementation of data-intensive Cloud applications is MapReduce [1].

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. In MapReduce, 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.

MapReduce implementations (e.g., Google's MapReduce [2] and Apache Hadoop [3]) 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 current 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 an Intercloud, a Cloud of clouds, where computing nodes may join and leave the system at an unpredictable rate. 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 current MapReduce implementations could be unreliable. The goal of our work is studying how the master-slave architecture of current MapReduce implementations can be improved to make it more suitable for dynamic Cloud scenarios like Interclouds.

In this chapter we present a P2P-MapReduce 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. An early version of this work, presenting a preliminary architecture of the P2P-MapReduce framework, has been presented in [4]. This chapter extends the previous work by describing an implementation of the P2P-MapReduce framework and a preliminary performance evaluation.

The remainder of this chapter is organized as follows. Section 2 provides a background on the MapReduce programming model. Section 3 describes the P2P-MapReduce architecture, its current implementation, and preliminary evaluation of its performance. Finally, Section 4 concludes the chapter.

2 MapReduce

As mentioned before, MapReduce applications are based on a master-slave model. This section briefly describes the various operations that are performed by a generic application to transform input data into output data according to that model.

Users define a *map* and a *reduce* function [1]. The *map* function processes a (key, value) pair and returns a list of intermediate (key, value) pairs:

map (k1,v1) → list(k2,v2).

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

reduce (k2, list(v2)) → list(v2).

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

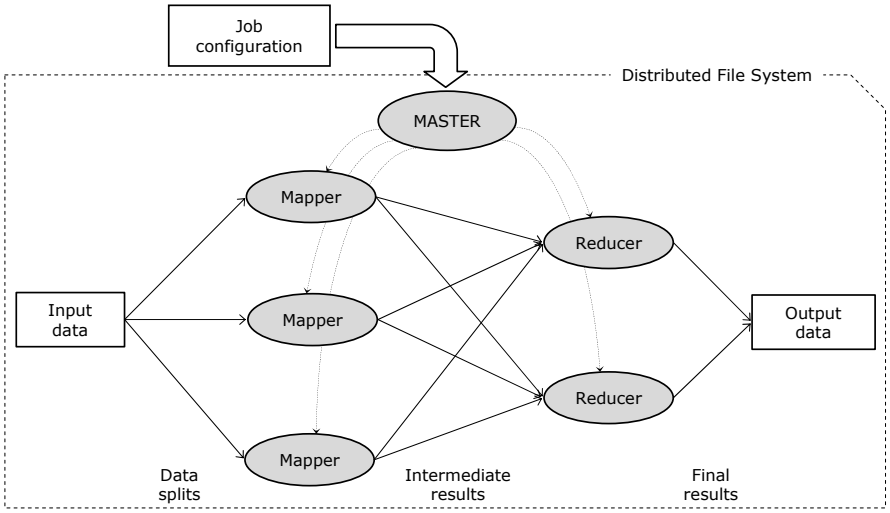

Fig. 1 Execution phases in a generic MapReduce application

1. A master process receives a "job configuration" describing the MapReduce job to be executed. The job configuration specifies, among other information, the location of the input data, which normally is a directory in a distributed file system.
2. According to the job configuration, 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 piece of data, each mapper process executes the *map* function (provided as part of the job configuration) 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 sent to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the job configuration) which, starting from all the intermediate pairs with the same key, generates a single pair containing that key and the aggregate values.
5. The results generated by each reducer process are then collected and delivered to a location specified by the job configuration, so as to form the final output data.

Besides the original MapReduce implementation by Google [2], several other MapReduce implementations have been realized within other systems, including Hadoop [3], GridGain [5], Skynet [6], MapSharp [7] and Disco [8]. Another system sharing most of the design principles of MapReduce is Sector/Sphere [9], 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. Ref. [10] describes a distributed data mining application developed using such system.

Several applications of the MapReduce paradigm have been demonstrated. Ref. [11] discusses some examples of interesting applications that can be expressed as MapReduce computations, 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. [3] mentions many significant types of applications that have been (or are being) implemented by 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.

3 P2P-MapReduce

The objective of the P2P-MapReduce framework is two-fold: *i*) handling master failures by dynamically replicating the job state on a set of backup masters; *ii*) supporting MapReduce applications over dynamic networks composed by nodes that join and leave the system at unpredictable rates.

To achieve these goals, P2P-MapReduce exploits the P2P paradigm by defining an architecture in which each node can act either as master or slave. The role assigned to a given node depends on the current characteristics of that node, and so it can change dynamically over time. Thus, at each time, a limited set of nodes is assigned the master role, while the others are assigned the slave role.

Moreover, each master node can act as backup node for other master nodes. A user node can submit the job to one of the master nodes, which will manage it as usual in MapReduce. That master will dynamically replicate the entire job state (i.e., the assignments of tasks to nodes, the locations of intermediate results, etc.) on its backup nodes. In case those backup nodes detect the failure of the master, they will

elect a new master among them that will manage the job computation using its local replica of the job state.

The remainder of this section describes the architecture of the P2P-MapReduce framework, its current implementation, and a preliminary evaluation of its performance.

3.1 Architecture

The P2P-MapReduce architecture includes three basic roles, shown in Fig. 2: user (*U*), master (*M*) and slave (*S*). Master nodes and slave nodes form two logical P2P networks called *M-net* and *S-net*, respectively. As mentioned above, computing nodes are dynamically assigned the master or slave role, hence *M-net* and *S-Net* change their composition over time. The mechanisms used for maintaining this infrastructure are discussed in Section 3.2.

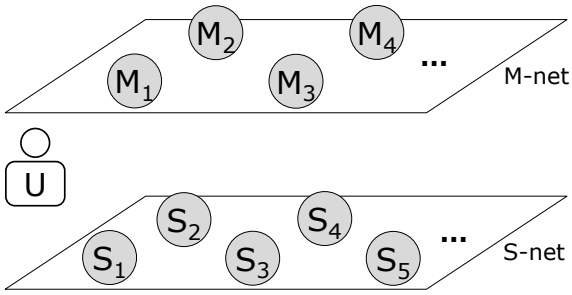

Fig. 2 Basic architecture of a P2P-MapReduce network

In the following we describe, through an example, how a master failure is handled in the P2P-MapReduce architecture. We assume the initial configuration represented in Fig. 2, where *U* is the user node that submits a MapReduce job, nodes *M* are the masters and nodes *S* are the slaves.

The following steps are performed to submit the job and to recover from a master failure (see Fig. 3):

1. *U* queries *M-net* to get the list of the available masters, each one characterized by a workload index that measures how busy the node is. *U* orders the list by ascending values of workload index and takes the first element as primary master. In this example, the chosen primary master is M_1; thus, *U* submits the MapReduce job to M_1.
2. M_1 chooses k masters for the backup role. In this example, assuming that $k = 2$, M_1 chooses M_2 and M_3 for this role. Thus, M_1 notifies M_2 and M_3 that they will act as backup nodes for the current job (in Fig. 3, the apex "*B*" to nodes M_2 and M_3 indicates the backup function). This implies that whenever the job state

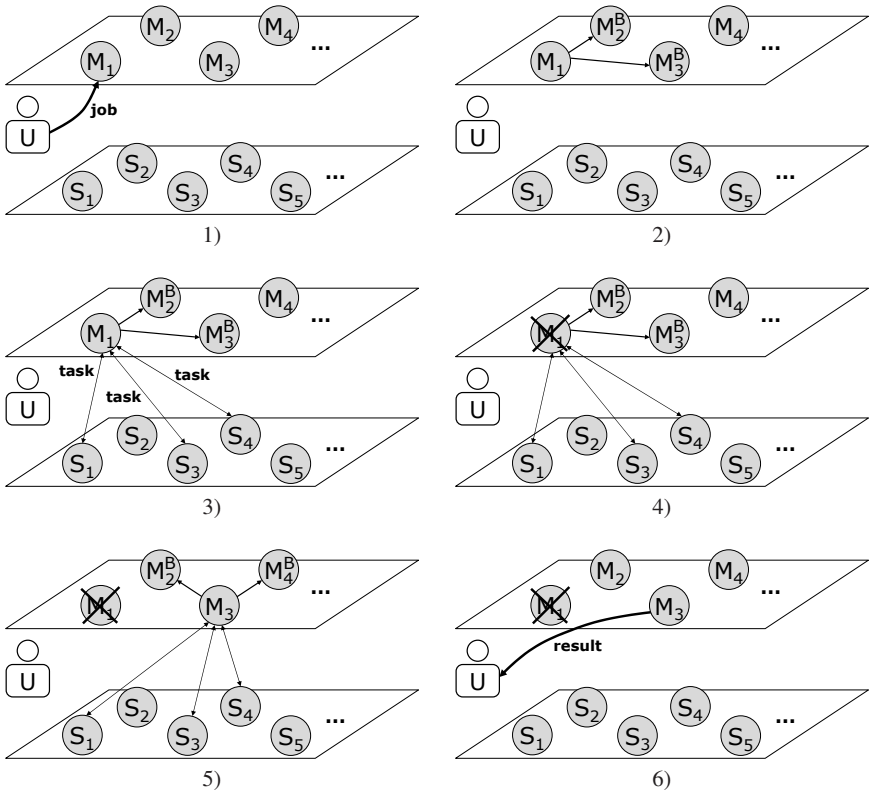

Fig. 3 Steps performed to submit a job and to recover from a master failure

changes, M_1 backups it on M_2 and M_3, which in turn will periodically check whether M_1 is alive.

3. M_1 queries *S-net* to get the list of the available slaves, choosing (part of) them to execute a map or a reduce task. As for the masters, the choice of the slave nodes to use is done on the basis of a workload index. In this example, nodes S_1, S_3 and S_4 are selected as slaves. The tasks are started on the slave nodes and managed as usual in MapReduce.
4. The primary master M_1 fails. Backup masters M_2 and M_3 detect the failure of M_1 and start a distributed procedure to elect a new primary master among them.
5. The new primary master (M_3) is elected by choosing the backup node with the lowest workload index. M_2 continues to play the backup function and, to keep k backup masters active, another backup node (M_4, in this example) is chosen by M_3. Then, M_3 proceeds to manage the MapReduce job using its local replica of the job state.
6. As soon as the MapReduce job is completed, M_3 returns the result to U.

It is worth noticing that the master failure and the subsequent recovery procedure are transparent to the user. It should also be noted that a master node may play at the same time the role of primary master for one job and that of backup master for another job.

3.2 Implementation

We implemented a prototype of the P2P-MapReduce framework using the JXTA framework [12]. JXTA provides a set of XML-based protocols that allow computers and other devices to communicate and collaborate in a P2P 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*. All resources (peers, services, etc.) are described by *advertisements* that are published within the peer group for resource discovery purposes.

In the following we briefly describe how the JXTA components are used in the P2P-MapReduce system to implement basic mechanisms for resource discovery, network maintenance, job submission and failure recovery. Then we describe the state diagram that steers the behavior of a generic node and the software modules provided by each node in a P2P-MapReduce network.

3.2.1 Basic mechanisms

Resource discovery

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 such as its `Role` and `WorkloadIndex`.

An edge peer 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 Services that uses SRDI to locate all the resources of that type without flooding the entire network.

Note that *M-net* and *S-net*, represented in Fig. 2, are "logical" networks in the sense that queries to *M-net* (or *S-net*) are actually submitted to the whole *MapRe-*

duceGroup but restricted to nodes having the attribute `Role` set to `"Master"` (or `"Slave"`) using the SRDI mechanisms.

Network maintenance

Network maintenance is carried out cooperatively by all nodes on the basis of their role. The maintenance task of each slave node is to check periodically the existence of at least one master in the network. In case no masters are found, the slave promotes itself to the master role. In this way, the first node joining the network always assumes the master role. The same happens to the last node remaining into the network.

The maintenance task of master nodes is to ensure the existence of a given percentage p of masters on the total number of nodes. This task is performed periodically by one master only (referred to as *coordinator*), which is elected for this purpose among all masters using a variation of the "bully" election algorithm. The coordinator has the power of changing slaves into masters, and viceversa. During a maintenance operation, the coordinator queries all nodes and orders them by ascending values of workload index: the first p percent of nodes must assume (or maintain) the master role, while the others will become or remain slaves. Nodes that have to change their role are notified by the coordinator in order to update their status.

Job submission and failure recovery

To describe the JXTA mechanisms used for job submission and master failure recovery, we take the six-point example presented in Section 3.1 as reference:

1. The user node invokes the Discovery Service to obtain the advertisements of the master nodes published in *MapReduceGroup*. Based on the `WorkloadIndex`, it chooses the primary master for its job. Then, it opens a bidirectional pipe (called *PrimaryPipe*) to the primary master and submits the job configuration.
2. The primary master invokes the Discovery Service to choose its backup masters and opens a multicast pipe (*BackupPipe*) to the backup masters. The *BackupPipe* has two goals: replicating job state information to the backup nodes and allowing backup nodes to detect a primary master failure in case the *BackupPipe* connection times out.
3. The primary master invokes the Discovery Service to select the slave nodes to use for the job. Slave nodes are filtered on the basis of `WorkloadIndex` attribute. The primary master opens a bidirectional pipe (*SlavePipe*) to each slave and starts a map or a reduce task on it.
4. The backup masters detect a primary master failure (i.e., a timeout on the *BackupPipe* connection) and start a procedure to elect the new primary master (to this end, they connect each other with a temporary pipe and exchange information about their current `WorkloadIndex`).
5. The backup master with the lowest `WorkloadIndex` is elected as the new primary master. This new primary master binds the pipes previously associated

to the old primary master (*PrimaryPipe*, *BackupPipe* and *SlavePipe*s), chooses (and connect to) a substitute backup master, and then continues to manage the MapReduce job using its replica of the job state.
6. The primary master returns the result of the MapReduce job to the user node through the *PrimaryPipe*.

The primary master detects the failure of a slave by getting a timeout to the associated *SlavePipe* connection. If this event occurs, a new slave is selected and the failed map or reduce task is assigned to it.

3.2.2 State diagram and software modules

The behavior of a generic node is modelled as a state diagram that defines the different states that a node can assume, and all the events that determine the transitions from a state to another one. Fig. 4 shows such state diagram modelled using the UML State Diagram formalism.

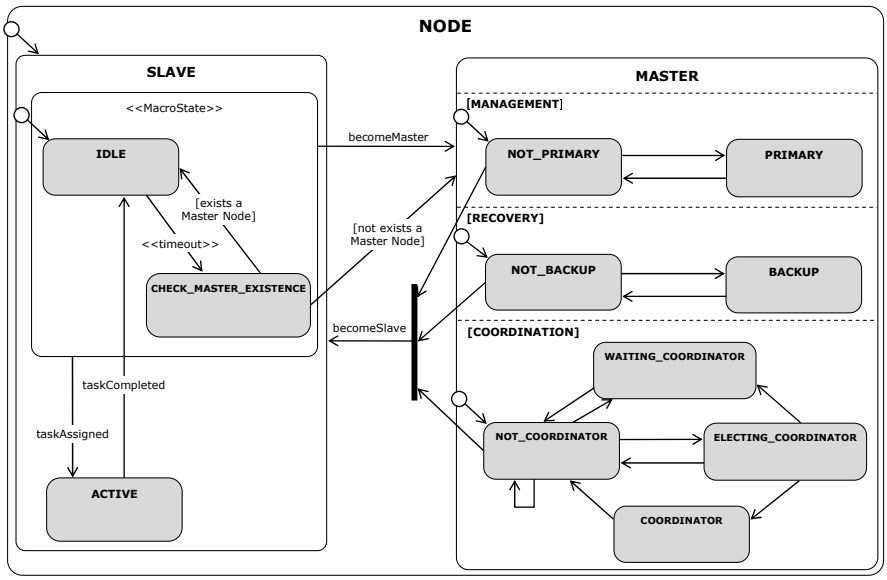

Fig. 4 UML State Diagram describing the behavior of a generic node in the P2P-MapReduce framework

The state diagram includes two macro-states, `SLAVE` and `MASTER`, which describes the two roles that can be assumed by each node. The `SLAVE` state has three states, `IDLE`, `CHECK_MASTER_EXISTENCE` and `ACTIVE`, which represent respectively: a slave waiting for task assignment; a slave checking the existence of a master; a slave executing a given task.

The MASTER state is modelled with three parallel macro-states, which represent the different roles a master can perform concurrently: possibly acting as a 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 a primary master for any job, and PRIMARY, which, in contrast, represents a master node currently managing at least one job as a 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 coordinator), COORDINATOR (the node is acting as coordinator), WAITING_COORDINATOR and ELECTING_COORDINATOR for nodes currently participating to the election of the new coordinator, as mentioned in Section 3.2.1.

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 that receive a *becomeSlave* message from the coordinator. Similarly, the transition from slave to master role is allowed to slaves that receive a *becomeMaster* and are not in ACTIVE state.

Finally, we briefly describe the software modules inside each node and how those modules interact each other in a P2P-MapReduce network (see Fig. 5).

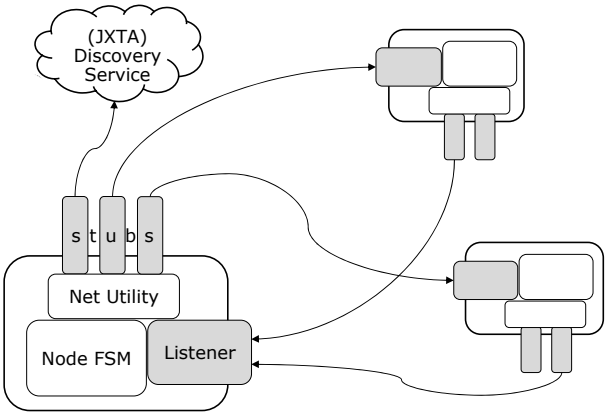

Fig. 5 Software modules inside each node and interactions among nodes in P2P-MapReduce

The *Node FSM* implements the logic of the finite state machine described in Fig. 4, steering the behavior of the node in response to local events or messages

coming from remote nodes (e.g., tasks assignments, discovery requests, etc.). The *Listener* is a component that receives messages from other nodes and passes them to the *Node FSM* for processing. The *Net Utility* allows outgoing messages to be sent to remote nodes; a specific *stub* is used for each remote service to be invoked. Additionally, each node interacts with the JXTA Discovery Service for publishing its features and for querying the system (e.g., when looking for idle slave nodes).

3.3 Evaluation

We carried out a preliminary set of experiments to evaluate the behavior of the P2P-MapReduce framework compared to a centralized implementation of MapReduce, in presence of dynamic nodes participation. The experimental results demonstrate that using a P2P approach it is possible to extend the MapReduce architectural model making it suitable for highly dynamic Cloud environments where failure must be managed to avoid a critical loss of computing resources and time.

The evaluation has been carried out by implementing a simulator of the system in which each node is represented by an independent thread. Each thread executes the algorithms specified by the state diagram in Fig.4, and communicates with the other threads by invoking local routines having the same interface of the JXTA pipes. Our simulation analyzes the system in steady state, that is when *M-net* and *S-net* are formed and the desired ratio between number of masters and slaves is reached.

The network includes 1000 nodes. To simulate a dynamic nodes participation a joining rate R_J and a leaving rate R_L are defined. On average, every $1/R_J$ seconds one node joins the network, while every $1/R_L$ another node abruptly leaves the network so as to simulate an event of failure (or a disconnection). In our simulation $R_J = R_L$ in order to keep the total number of nodes and the master/slave ratio approximatively constant during the whole simulation. In particular, we considered the following values for R_J and R_L: 0.05, 0.1 and 0.2, which correspond to the join/failure of one node (out of 1000 nodes) every 20, 10 and 5 seconds, respectively.

Every 120 seconds (mean value) a user entity submits one job to the system. The average sequential duration of a job is 20 hours that are distributed, on average, to 100 nodes. On the basis of the actual number of slaves, the system determines the amount of time each slave will be busy to complete its task. Every node, other than managing a job or a task, executes the network maintenance operations described above (election of the coordinator, choice of backup masters, and so on).

The main task performed by the simulator is evaluating the number of jobs failed versus the total number of nodes submitted to the system. For the purpose of our simulations, a "failed" job is a job that does not complete its execution, i.e., does not return a result to the submitting user. 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 re-assigned to another slave.

The system has been evaluated in two scenarios: *i*) *centralized*, where there is only one primary master and there are not backup masters; *ii*) *P2P*, where there are 10 masters and each job is managed by one master which periodically replicates the job state on one backup master. Fig. 6 presents the percentage of completed jobs in centralized and P2P scenarios after the submission of 100 jobs.

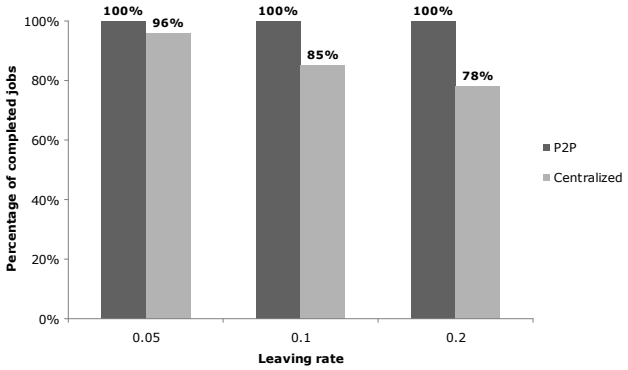

Fig. 6 Percentage of completed jobs in centralized and P2P scenarios for a leaving rate ranging from 0.05 to 0.2

As expected, in the centralized scenario the number of failed jobs increases as the leaving rate increases. In contrast, The P2P-MapReduce scenario is able to complete all the jobs for all the considered leaving rates, even if we used just one backup per job. It is worth noticing that when a backup master becomes primary master as a consequence of a failure, it chooses another backup in its place to maintain the desired level of reliability.

The percentages in Fig. 6 can be translated into lost CPU hours, by multiplying the average job duration to the average number of failed jobs. In the centralized scenario, the absolute number of failed jobs is 4, 15 and 22 for leaving rates equal to 0.05, 0.1 and 0.2, respectively. Hence, with an average sequential duration of 20 hours per job, the total number of lost computing hours equals, in the worst case, to 80, 300 and 440 hours.

We can further estimate the amount of resources involved in a typical MapReduce job by taking the statistics about a large set of MapReduce jobs run at Google, presented in [1]. On March 2006, the average completion time per job has been 874 seconds, using 268 slaves on average. Assuming that each machine is fully assigned to one job, the overall machine time is 874×268 seconds (about 65 hours). On September 2007, the average job completion time has been 395 seconds using 394 machines, with an overall machine time of 43 hours.

From the statistics reported above, and from the results generated by our experiments, we see that a master failure causes loss of dozens CPU hours for a typical MapReduce job. Moreover, when the number of available machines per user is limited (as in a typical Cloud systems where resources are shared among thousands

of users), a master failure produces also a significant loss of time, since the job completion time increases as the number of machines decreases.

4 Conclusions

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

The P2P-MapReduce model presented in this chapter exploits a P2P model to perform job state replication, manage master failures and allow intermittent nodes participation 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.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, vol. 51 n. 1 (2008) 107-113.
2. Google's Map Reduce. http://labs.google.com/papers/mapreduce.html (Visited: September 2009).
3. Hadoop. http://hadoop.apache.org (Visited: September 2009).
4. Marozzo, F., Talia, D., Trunfio, P.: Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model. Workshop on Cloud Computing and its Applications, Chicago, USA, 2008.
5. Gridgain. http://www.gridgain.com (Visited: September 2009).
6. Skynet. http://skynet.rubyforge.org (Visited: September 2009).
7. MapSharp. http://mapsharp.codeplex.com (Visited: September 2009).
8. Disco. http://discoproject.org (Visited: September 2009).
9. Gu, Y., Grossman, R.: Sector and Sphere: The Design and Implementation of a High Performance Data Cloud, Philosophical Transactions. Series A: Mathematical, physical, and engineering sciences, vol. 367 n. 1897 (2009) 2429-2445.
10. Grossman, R., Gu, Y.: Data Mining Using High Performance Data Clouds: Experimental Studies Using Sector and Sphere. SIGKDD 2008, Las Vegas, USA, 2008.
11. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Symposium on Operating Systems Design and Implementation (OSDI), San Francisco, USA, 2004.
12. Gong, L.: JXTA: A Network Programming Environment. IEEE Internet Computing, vol. 5 n. 3 (2001) 88-95.