P2P-MapReduce: Parallel data processing in dynamic Cloud environments

Fabrizio Marozzo[a], Domenico Talia[a,b], Paolo Trunfio[a,*]

[a] *DEIS, University of Calabria, Via P. Bucci 41C, 87036 Rende (CS), Italy*
[b] *ICAR-CNR, Via P. Bucci 41C, 87036 Rende (CS), Italy*

Abstract

MapReduce is a programming model for parallel data processing widely used in Cloud computing environments. Current MapReduce implementations are based on centralized master-slave architectures that do not cope well with dynamic Cloud infrastructures, like a Cloud of clouds, in which nodes may join and leave the network at high rates. We have designed an adaptive MapReduce framework, called *P2P-MapReduce*, which 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 paper describes the P2P-MapReduce system providing a detailed description of its basic mechanisms, a prototype implementation, and an extensive performance evaluation in different network scenarios. The performance results confirm the good fault tolerance level provided by the P2P-MapReduce framework compared to a centralized implementation of MapReduce, as well as its limited impact in terms of network overhead.

Keywords: MapReduce; Parallel Processing; Cloud computing; Peer-to-peer computing; P2P-MapReduce.

1. Introduction

MapReduce is a system and method for efficient large-scale data processing presented by Google in 2004 [1] to cope with the challenge of processing very large input data generated by Internet-based applications. Since its introduction, MapReduce has proven to be applicable to a wide range of 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 [2]. Today, MapReduce is widely recognized as one of the most important programming models for Cloud computing environments, being it supported by leading Cloud providers such as Amazon, with its Elastic MapReduce service [3], and Google itself, which recently released a Mapper API for its App Engine [4].

The MapReduce abstraction is inspired by the *map* and *reduce* primitives present in Lisp and other functional languages [5]. A user defines a MapReduce application 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. Current MapReduce implementations, like Google's MapReduce [6] and Hadoop [2], are based on a master-slave architecture. A job is submitted by a user node to a master node that selects idle workers and assigns a map or reduce task to each one. When all the 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 master failures are not explicitly managed as designers consider failures unlikely in reliable computing environments, such as a data center or a dedicated Cloud.

*Corresponding author.
Email addresses: fmarozzo@deis.unical.it (Fabrizio Marozzo), talia@deis.unical.it (Domenico Talia), trunfio@deis.unical.it (Paolo Trunfio)

On the contrary, node churn and failures - including master failures - are likely in large 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 such problems is fundamental to enable reliable MapReduce applications in dynamic Cloud infrastructures, where current MapReduce middleware could be unreliable. We have designed an adaptive MapReduce framework, called *P2P-MapReduce*, which 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 paper describes the P2P-MapReduce system providing a detailed description of its basic mechanisms, a prototype implementation, and an extensive performance evaluation in different network scenarios. The experimental results show that, differently from centralized master-server implementations, the P2P-MapReduce framework does not suffer from job failures even in presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in very dynamic Cloud infrastructures. In an early version of this work [7] we presented a preliminary architecture of the P2P-MapReduce framework, while in a more recent paper [8] we introduced its main software modules and a preliminary evaluation. This paper significantly extends our previous work by providing a detailed description of the mechanisms at the base of the P2P-MapReduce system, as well as an extensive evaluation of its performance in different scenarios.

The remainder of this paper is organized as follows. Section 2 provides a background on the MapReduce programming model and discusses related work. Section 3 introduces the system model and presents the general architecture of the P2P-MapReduce framework. System 4 describes the system mechanisms, while Section 5 discusses its implementation. Section 6 evaluates the performance of P2P-MapReduce compared to a centralized implementation of MapReduce. Finally, Section 7 concludes the paper.

2. Background and Related Work

This section provides a background on the MapReduce programming model and discusses related work.

2.1. The MapReduce Programming Model

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 [5]. 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(v3).

As an example, let's consider the creation of an inverted index for a large set of Web documents [1]. In its basic form, an inverted index contains a set of words (index terms), and for each word it specifies the IDs of all the documents that contain that word. Using a MapReduce approach, the *map* function parses each document and emits a sequence of (word, documentID) pairs. The *reduce* function takes all pairs for a given word, sorts the corresponding document IDs, and emits a (word, list(documentID)) pair. The set of all output pairs generated by the *reduce* function forms the inverted index for the input documents.

In general, the whole transformation process performed in a MapReduce application can be described through the following steps (see Figure 1):

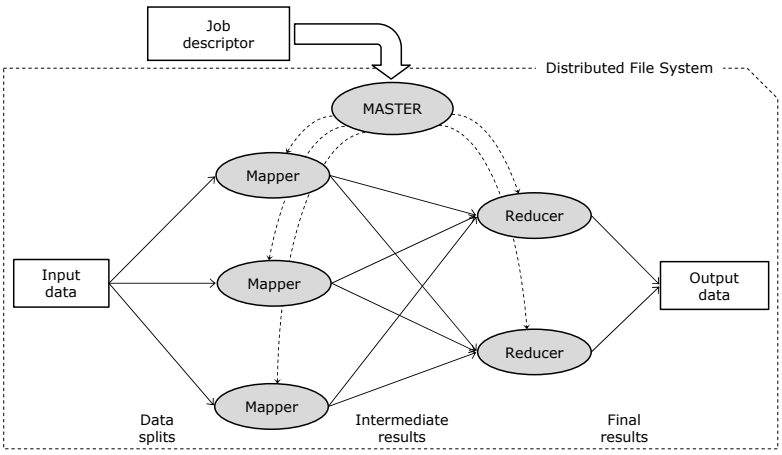


Figure 1: Execution phases of a generic MapReduce application.

1. A master process receives a job descriptor which 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 with 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.

Distributed file systems are the most popular solution for accessing input/output data in MapReduce systems, particularly for standard computing environments like a data center or a cluster of computers. On the other hand, distributed file systems may be ineffective in large-scale dynamic Cloud environments characterized by high levels of churn. Therefore, we assume that data are moved across nodes using a file transfer protocol like FTP or HTTP as done, for example, by the MISCO MapReduce Framework [9].

2.2. Related Work

Besides the original MapReduce implementation by Google [6], several other MapReduce implementations have been realized within other systems, including Hadoop [2], GridGain [10], Skynet [11], MapSharp [12], Twister [13], and Disco [14]. Another system sharing most of the design principles of MapReduce is Sector/Sphere [15], 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 [16] and the I/O system [17], while others focused on adapting the MapReduce model to specific computing environments, like shared-memory systems [18], volunteer computing environments [19], Desktop Grids [20], and mobile environments [9].

Zaharia et al. [16] studied how to improve the Hadoop's scheduler in heterogeneous environments, by designing a new scheduling algorithm, called LATE, which significantly improves response times in heterogeneous settings. The LATE algorithm uses estimated finish times to efficiently schedule speculative copies

of tasks (also called "backup" tasks in MapReduce terminology) to finish the computation faster. The main policy adopted by LATE is to speculatively execute the task that is thought to finish farthest into the future.

The Hadoop Online Prototype (HOP) [17] modifies the Hadoop MapReduce framework to support online aggregation, allowing users to see early returns from a job as it is being computed. HOP also supports continuous queries, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. HOP extends the applicability of the MapReduce model to pipelining behaviors, which is useful for batch processing. In fact, by pipelining both within and across jobs, HOP can reduce the time to job completion.

Phoenix [18] is an implementation of MapReduce for shared-memory systems that includes a programming API and a runtime system. Phoenix uses threads to spawn parallel map or reduce tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors in order to achieve load balance and maximize task throughput. Overall, Phoenix proves that MapReduce is a useful programming and concurrency management approach also for multi-core and multi-processor systems.

MOON [19] is a system designed to support MapReduce jobs on opportunistic environments. It extends Hadoop with adaptive task and data scheduling algorithms to offer reliable MapReduce services on a hybrid resource architecture, where volunteer computing systems are supplemented by a small set of dedicated nodes. The adaptive task and data scheduling algorithms in MOON distinguish between different types of MapReduce data and different types of node outages in order to place tasks and data on both volatile and dedicated nodes.

Another system that shares some of the key ideas behind MOON is that proposed by Tang et al. [20]. The system is specifically designed to support MapReduce applications in Desktop Grids, and exploits the BitDew middleware [21], which is a programmable environment for automatic and transparent data management on Desktop Grids. BitDew relies on a specific set of metadata to drive key data management operations, namely life cycle, distribution, placement, replication and fault-tolerance with a high level of abstraction.

Finally, Misco [9] is a framework for supporting MapReduce applications on mobile systems. Although Misco follows the general design of MapReduce, it does vary in two main aspects: task assignment and data transfer. The first aspect is managed through the use of a polling strategy. Each slave polls the master each time it becomes available. If there are no tasks to execute, the slave will idle for a period of time before requesting a task again. For data transfer, instead of a distributed file system that is not practical in a mobile scenario, Misco uses HTTP to communicate requests, task information and transfer data.

Even though P2P-MapReduce shares some basic ideas with some of the systems discussed above (in particular, [19, 20, 9]), 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 described in Section 4 allows 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. System Model and Architecture

As mentioned before, the goal of P2P-MapReduce is to enable a reliable execution of MapReduce applications in Cloud environments characterized by high levels of churn. To achieve this goal, P2P-MapReduce adopts a peer-to-peer model in which a wide set of autonomous nodes (peers) can act either as a master or as a slave. At each time, a limited set of nodes is assigned the master role, while the others are assigned the slave role. The role assigned to a given node can change dynamically over time, so as to ensure the presence of the desired master/slave ratio for reliability and load balancing purposes.

To prevent loss of work in the case of a master failure, each master can act as a backup for other masters. The master responsible for a job J, referred to as the *primary master* for J, dynamically updates the job state (e.g., the assignments of tasks to nodes, the status of each task, etc.) on its backup nodes, which are referred to as the *backup masters* for J. To prevent excessive overhead, the update does not contain whole job information, but only that part of information that has changed. If a primary master fails (or,

equivalently, it abruptly leaves the network), its place is taken by one of its backup masters in a way that is transparent both to the user who submitted the job, and to the job itself.

The overall system behavior, as well as its features (resilience to failures, load balancing), are the result of the behavior of each single node in the system. The node behavior will be described in detail in Section 4 as a state diagram that defines the different states a node can assume, and all the events that determine transitions from state to state. The remainder of this section describes the system model and the general architecture of the P2P-MapReduce framework.

3.1. System Model

The model introduced here provides abstractions for describing the characteristics of jobs, tasks, users, and nodes. For the reader's convenience, Figure 2 illustrates the system model entities and their interrelationships using the UML Class Diagram formalism.

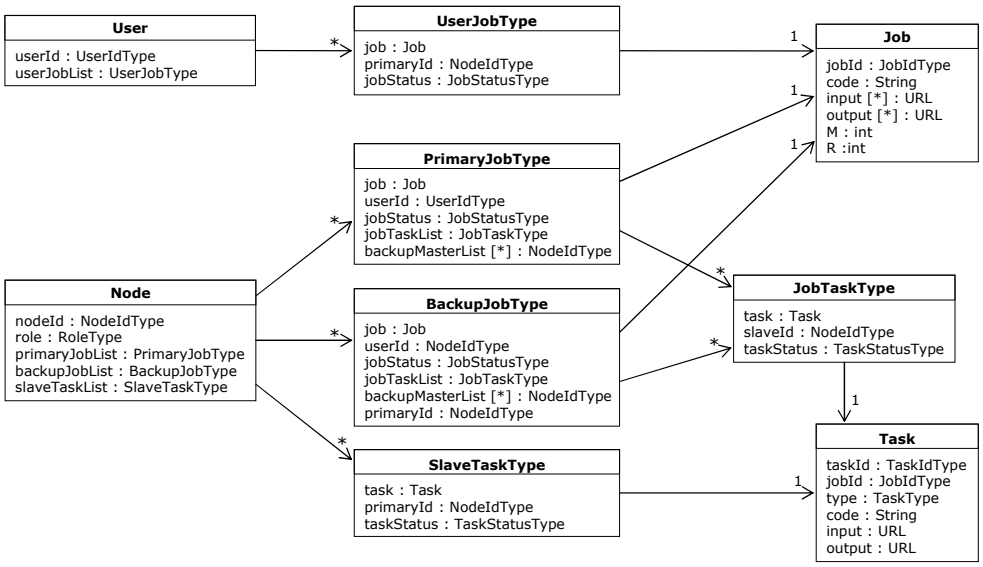


Figure 2: System model described through the UML Class Diagram formalism.

3.1.1. Jobs and tasks

A *job* is modelled as a tuple of the form:

$job = \langle jobId,\ code,\ input,\ output,\ M,\ R \rangle$

where *jobId* is a job identifier, *code* includes the map and reduce functions, *input* (resp., *output*) is the location of the input (output) data of the job, M is the number of map tasks, and R is the number of reduce tasks.

A *task* is modelled as a tuple:

$task = \langle taskId,\ jobId,\ type,\ code,\ input,\ output \rangle$

where *taskId* is a task identifier, *jobId* is the identifier of the job the task belongs to, *type* can be either `MAP` or `REDUCE`, *code* includes the map or reduce function (depending on the task type), and *input* (*output*) is the location of the input (output) data of the task.

3.1.2. Users and nodes

A *user* is modelled as a pair of the form:

$user = \langle userId,\ userJobList \rangle$

which contains the user identifier (*userId*) and the list of jobs submitted by the user (*userJobList*). The *userJobList* contains tuples of a *userJobType* defined as:

$$userJobType = \langle job, primaryId, jobStatus \rangle$$

where *job* is a job descriptor, *primaryId* is the identifier of the node that is managing the job as the primary master, and *jobStatus* represents the current job status.

A *node* is modelled as a tuple:

$$node = \langle nodeId, role, primaryJobList, backupJobList, slaveTaskList \rangle$$

which contains the node identifier (*nodeId*), its *role* (`MASTER` or `SLAVE`), the list of jobs managed by this node as the primary master (*primaryJobList*), the list of jobs managed by this node as a backup master (*backupJobList*), and the list of tasks managed by this node as a slave (*slaveTaskList*). Note that *primaryJobList* and *backupJobList* are empty if the node is currently a slave, while *slaveTaskList* is empty if the node is acting as a master.

The *primaryJobList* contains tuples of a *primaryJobType* defined as:

$$primaryJobType = \langle job, userId, jobStatus, jobTaskList, backupMasterList \rangle$$

where *job* is a job descriptor, *userId* is the identifier of the user that has submitted the job, *jobStatus* is the current status of the job, *jobTaskList* is a list containing dynamic information about the tasks that compose the job, and *backupMasterList* is a list containing the identifiers (*backupId*) of the backup masters assigned to the job. The *jobTaskList* contains tuples of a *jobTaskType*, which is defined as follows:

$$jobTaskType = \langle task, slaveId, taskStatus \rangle$$

where *task* is a task descriptor, *slaveId* is the identifier of the slave responsible for the task, and *taskStatus* represents the current task status.

The *backupJobList* contains tuples of a *backupJobType* defined as:

$$backupJobType = \langle job, userId, jobStatus, jobTaskList, backupMasterList, primaryId \rangle$$

that differs from *primaryJobType* for the presence of an additional field, *primaryId*, which represents the identifier of the primary master associated with the job.

Finally, the *slaveTaskList* contains tuples of a *slaveTaskType*, which is defined as:

$$slaveTaskType = \langle task, primaryId, taskStatus \rangle$$

where *task* is a task descriptor, *primaryId* is the identifier of the primary master associated with the task, and *taskStatus* contains its status.

3.2. Architecture

The P2P-MapReduce architecture includes three types of nodes, as shown in Figure 3: *user*, *master* and *slave*. Master nodes and slave nodes form two logical peer-to-peer networks referred to as *M-net* and *S-net*, respectively. As mentioned earlier in this section, computing nodes are dynamically assigned the master or the slave role, thus *M-net* and *S-Net* change their composition over time. The mechanisms used for maintaining the infrastructure will be described in Section 4.

User nodes submit their MapReduce jobs to the system through one of the available masters. The choice of the master to which to 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 viceversa, 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.

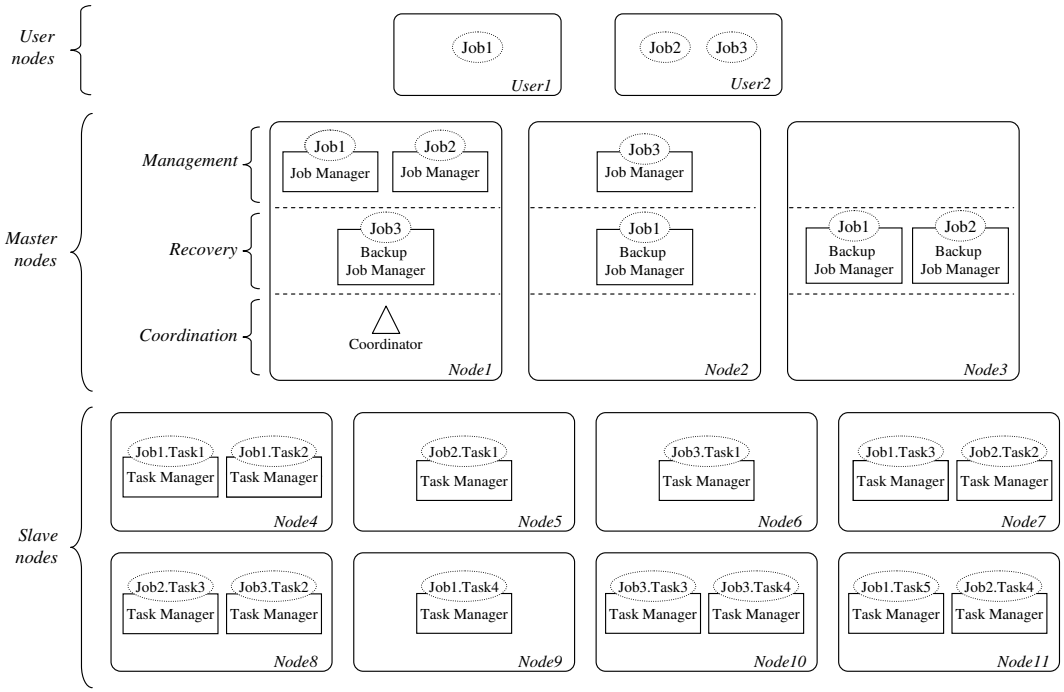


Figure 3: General architecture of P2P-MapReduce.

Figure 3 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 with *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.

The system mechanisms sketched above are described in detail in Section 4, while Section 5 will provide a description of the system implementation.

4. System Mechanisms

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 state. Figure 4 shows such state diagram modelled using the UML State Diagram formalism.

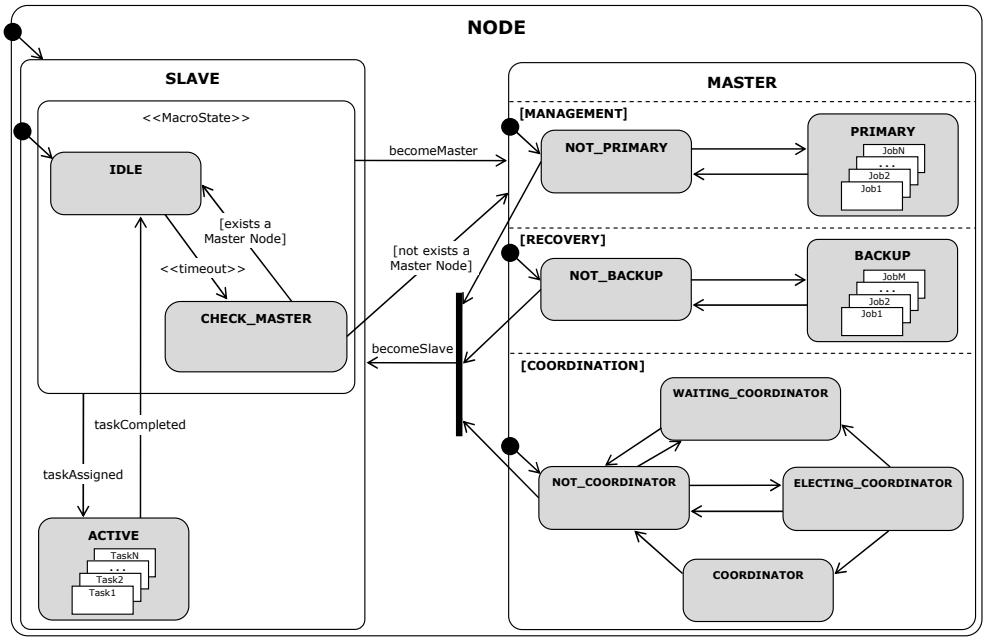


Figure 4: Behavior of a generic node described by an UML State Diagram.

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 coordinator), `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.

The events that determine state transitions are shown in Table 1. For each event message the table shows: the event parameters; whether it is an inner event; the sender's state; the receiver's state.

In the following we describe in detail the algorithmic behavior of each node, using Table 1 as a reference for the events that are exchanged among system entities. We proceed as follows. Section 4.1 describes

data structures and high-level behaviors of slave and master nodes. Section 4.2 focuses on job and tasks management, by describing the algorithms that steer the behavior of Job Managers, Tasks Managers, and Backup Job Managers. Finally, Section 4.3 describes the behavior of user nodes.

Table 1: Description of the event messages that can be exchanged by P2P-MapReduce nodes

Event	Parameters	Inner event	Sender's state	Receiver's state
becomeMaster		no	COORDINATION	SLAVE
becomeSlave		no	COORDINATION	MANAGEMENT
jobAssigned	Job, UserIdType	no	USER	MANAGEMENT
jobReassigned	BackupJobType	yes	BACKUP_JOB_MANAGER	MANAGEMENT
jobCompleted	JobIdType, JobStatusType	yes no	JOB_MANAGER MANAGEMENT	MANAGEMENT USER
backupJobAssigned	PrimaryJobType, NodeIdType	no	JOB_MANAGER	RECOVERY
jobUpdate	JobIdType, List<NodeIdType>, List<JobTaskType>	no	JOB_MANAGER	RECOVERY
backupJobCompleted	JobIdType	yes no	BACKUP_JOB_MANAGER MANAGEMENT	RECOVERY RECOVERY
taskAssigned	List<Task>, NodeIdType	no	JOB_MANAGER	SLAVE
taskCompleted	TaskIdType, TaskStatusType	yes no	TASK_MANAGER SLAVE	SLAVE MANAGEMENT
primaryUpdate	JobIdType, NodeIdType	no no yes no yes	MANAGEMENT MANAGEMENT RECOVERY MANAGEMENT SLAVE	USER RECOVERY BACKUP_JOB_MANAGER SLAVE TASK_MANAGER

4.1. Node Behavior

Figure 5 describes the behavior of a generic node, by specifying the algorithmic details behind the finite state machine depicted in Figure 4. Each node includes five fields, already introduced in Section 3.1: *nodeId*, *role*, *primaryJobList*, *backupJobList* and *slaveTaskList*.

As soon as a node joins the network, it transits to the SLAVE macro-state (*line 8*). Then, it sets its role accordingly (*line 12*) and passes to the IDLE state (*line 13*). When a slave is in IDLE state, three events can happen:

- An internal timeout elapses, causing a transition to the CHECK_MASTER state (*line 17*).
- A *taskAssigned* message is received from the Job Manager of a primary master (*line 19*). The message includes a list with the tasks to be executed (*assignedTaskList*), and the identifier of the primary master (*primaryId*). For each assigned task, a new *slaveTask* is created, it is added to the *slaveTaskList*, and associated with a *TaskManager* (lines 20-24). Then, the slave passes to the ACTIVE state (*line 25*).
- A *becomeMaster* message is received from the coordinator, causing a transition to the MASTER state (*lines 27-28*).

When a slave is in the CHECK_MASTER state, it queries the discovery service to check the existence of at least one master in the network. In case no masters are found, the slave promotes itself to the master role; otherwise, it returns to the IDLE state (*lines 33-36*). This state allows the first node joining (and the last node remaining into) the network to assume the master role. A node in CHECK_MASTER state can also receive *taskAssigned* and *becomeMaster* events, which are managed as discussed earlier.

```
1  StateDiagram NODE {
2   NodeIdType nodeId;
3   RoleType role;
4   List<PrimaryJobType> primaryJobList;
5   List<BackupJobType> backupJobList;
6   List<SlaveTaskType> slaveTaskList;
7   entry() {
8    transition to SLAVE;
9   }
10  Sequential Macro-State SLAVE {
11   entry() {
12    role = SLAVE;
13    transition to IDLE;
14   }
15   State IDLE {
16    do() {
17     transition to CHECK_MASTER after a timeout;
18    }
19    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
20     foreach(Task t in assignedTaskList){
21      SlaveTaskType st(task, primaryId);
22      add st to slaveTaskList;
23      start TaskManager for st;
24     }
25     transition to ACTIVE;
26    }
27    on event becomeMaster() {
28     transition to MASTER;
29    }
30   }
31   State CHECK_MASTER {
32    do() {
33     if(exists a master)
34      transition to IDLE;
35     else
36      transition to MASTER;
37    }
38    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
39     //same as lines 20 to 25
40    }
41    on event becomeMaster() {
42     transition to MASTER;
43    }
44   }
45   State ACTIVE {
46    on event taskAssigned(List<Task>
          assignedTaskList, NodeIdType primaryId) {
47     //same as lines 20 to 24
48    }
49    on event primaryUpdate(JobIdType
          updatedJobId, NodeIdType
          updatedPrimaryId) {
50     foreach(SlaveTaskType t in slaveTaskList) {
51      if(t.task.jobId = updatedJobId){
52       t.primaryId = updatedPrimaryId;
53       propagate event to TaskManager for t;
54      }
55     }
56    }
57    on inner event taskCompleted(TaskIdType
          completedTaskId, TaskStatusType
          completedTaskStatus) {
58     SlaveTaskType t = tuple in slaveTaskList
           with task.taskId=completedTaskId;
59     stop taskManager for t;
60     propagate event to t.primaryId;
61     if(slaveTaskList does not contain tasks to
          execute)
62      transition to IDLE;
63    }
64   }
65  }
66  Concurrent Macro-State MASTER {
67   entry() {
68    role = MASTER;
69    concurrent transition to MANAGEMENT, RECOVERY
          and COORDINATION;
70   }
71   on event becomeSlave() {
72    if(MANAGEMENT in NOT_PRIMARY and RECOVERY in
          NOT_BACKUP and COORDINATION in
          NOT_COORDINATOR)
73     transition to SLAVE;
74   }
75   Sequential Macro-State MANAGEMENT {
76    //see Figure 6
77   }
78   Sequential Macro-State RECOVERY {
79    //see Figure 7
80   }
81   Sequential Macro-State COORDINATION {
82    //see Figure 8
83   }
84  }
85 }
```

Figure 5: Pseudo-code describing the behavior of a generic node.

Slaves in `ACTIVE` state can receive three events:

- *taskAssigned*, which adds other tasks to those already being executed (*line 46*).
- *primaryUpdate*, which informs the slave that the primary master for a given job has changed (*line 49*). This message, sent by the new primary master, includes the identifier of the job whose primary master has changed (*updatedJobId*), and the identifier of the new primary master (*updatedPrimaryId*). Each task in the *slaveTaskList* whose job identifier equals *updatedJobId* is updated accordingly, and the associated Task Manager is notified by propagating the event to it (lines 50-55).
- *taskCompleted*, an inner event sent by the Task Manager of the same node, on completion of a given task (*line 57*). The event includes the identifier of the completed task (*completedTaskId*) and its status (*completedTaskStatus*). The slave identifies the corresponding tasks in the *slaveTaskList*, stops the corresponding Task Manager, and propagates the event to the primary master (*lines 58-60*). If there are no more tasks being executed, the slave transits to the `IDLE` state.

A node that is promoted to the `MASTER` macro-state sets its role accordingly (*line 68*), and concurrently transits to three parallel states (*line 69*), as depicted in Figure 4 and mentioned earlier in this section: `MANAGEMENT`, `COORDINATION` and `RECOVERY`. If a master receives a *becomeSlave* event from the coordinator (*line 71*), and the internal states of the `MANAGEMENT`, `COORDINATION` and `RECOVERY` concurrent macro-states

are respectively equal to NOT_PRIMARY, NOT_BACKUP and NOT_COORDINATOR, it transits to the SLAVE state (*lines 72-73*).

```
1  Sequential Macro-State MANAGEMENT {
2   entry() {
3    transition to NOT_PRIMARY;
4   }
5   State NOT_PRIMARY {
6    on event jobAssigned(Job job, UserIdType
         userId) {
7     PrimaryJobType t(job, userId);
8     add t to primaryJobList;
9     start jobManager for t;
10    transition to PRIMARY;
11   }
12   on inner event jobReassigned(BackupJobType bt)
         {
13    PrimaryJobType pt(bt.job, bt.userId,
          bt.jobStatus, bt.jobTaskList,
          bt.backupMasterList);
14    add pt to primaryJobList;
15    foreach(NodeIdType b in pt.backupMasterList)
16     send event primaryUpdate(pt.job.jobId,
           nodeId) to b;
17    foreach(JobTaskType t in pt.jobTaskList)
18     send event primaryUpdate(pt.job.jobId,
           nodeId) to t.slaveId;
19    send event primaryUpdate(pt.job.jobId,
          nodeId) to pt.userId;
20    start jobManager for pt;
21    transition to PRIMARY;
22   }
23  }
24  State PRIMARY {
25   on event jobAssigned(Job job, UserIdType
         userId) {
26    //same as lines 7 to 9
27   }
28   on inner event jobReassigned(BackupJobType bt)
         {
29    //same as lines 13 to 20
30   }
31   on inner event jobCompleted(JobIdType
         completedJobId, JobStatusType
         completedJobStatus) {
32    PrimaryJobType t = tuple in PrimaryJobList
          with job.jobId=completedJobId;
33    stop jobManager for t;
34    foreach(NodeIdType b in t.backupMasterList)
35     send event
           backupJobCompleted(completedJobId) to b;
36    propagate event to t.userId;
37    if(primaryJobList does not contain jobs to
          execute)
38     transition to NOT_PRIMARY;
39   }
40  }
41 }
```

Figure 6: Pseudo-code describing the behavior of a master node performing Management activities.

The MANAGEMENT macro-state is described in Figure 6. Initially, the master transits to the NOT_PRIMARY state, where two events can be received:

- *jobAssigned*, through which a user submits a job to the master (*line 6*). The message includes the *job* to be managed and the identifier of the submitting user (*userId*). In response to this event, a new *primaryJob* is created, it is added to the *primaryJobList*, and associated with a *JobManager* (*lines 7-9*). Then, the master transits to the PRIMARY state (*line 10*).
- *jobReassigned*, an inner event sent by the Backup Job Manager of the same node to notify this master that it has been elected as the new primary for a *backupJob* received as parameter (*line 12*). The following operations are performed: a new *primaryJob*, created from *backupJob*, is added to the *primaryJobList* (*lines 13-14*); all the other backup masters associated with this job are notified about the primary master change, by sending a *primaryUpdated* message to them (*lines 15-16*); a *primaryUpdate* message is also sent to all the slaves that are managing tasks part of this job (*lines 17-18*), as well as to the user that submitted the job (*line 19*); finally, a new Job Manager is started for the *primaryJob* and the state is changed to PRIMARY.

When a master is in the PRIMARY state, besides the *jobAssigned* and *jobReassigned* events, which are identical to those present in the NOT_PRIMARY state, it can receive a *jobCompleted* inner event from a Job Manager, on completion of a given job (*line 31*). The event includes the identifier of the completed job (*completedJobId*) and its status (*completedJobStatus*). The master identifies the job in the *primaryJobList*, stops the corresponding Job Manager, and sends a *backupJobCompleted* event to all the backup masters associated with this job (*lines 32-35*). Then, the event is propagated to the user that submitted the job (*line 36*) and, if there are no more jobs being managed, the master transits to the NOT_PRIMARY state.

```
1  Sequential Macro-State RECOVERY {
2   entry() {
3    transition to NOT_BACKUP;
4   }
5   State NOT_BACKUP {
6    on event backupJobAssigned(PrimaryJobType
         primaryJob, NodeIdType primaryId) {
7     BackupJobType t(primaryJob, primaryId);
8     add t to backupJobList;
9     start backupJobManager for t;
10    transition to BACKUP;
11   }
12  }
13  State BACKUP {
14   on event backupJobAssigned(PrimaryJobType
         primaryJob, NodeIdType primaryId) {
15    //same as lines 7 to 9
16   }
17   on event primaryUpdate(JobIdType updatedJobId,
         NodeIdType updatedPrimaryId) {
18    BackupJobType t = tuple in BackupJobList with
          job.jobId=updatedJobId;
19    t.primaryId = updatedPrimaryId;
20    propagate event to backupJobManager for t;
21   }
22   on event jobUpdate(JobIdType updatedJobId,
         List<NodeIdType> updatedBackupMasterList,
         List<JobTaskType> updatedJobTaskList) {
23    BackupJobType bt = tuple in backupJobList
          with job.jobId=updatedJobId;
24    bt.backupMasterList = updatedBackupMasterList;
25    if(updatedJobTaskList is not empty)
26     foreach(JobTaskType t in updatedJobTaskList)
27      update t in bt.jobTaskList;
28   }
29   on [inner] event backupJobCompleted(JobIdType
         completedJobId) {
30    BackupJobType t = tuple in BackupJobList with
          job.jobId=completedJobId;
31    stop backupJobManager for t;
32    if(backupJobList does not contain jobs to
          execute)
33     transition to NOT_BACKUP;
34   }
35  }
36 }
```

Figure 7: Pseudo-code describing the behavior of a master node performing Recovery activities.

Figure 7 describes the `RECOVERY` macro-state. Masters in the `NOT_BACKUP` state can receive a *backupJobAssigned* event from the Job Manager of another node that is acting as the primary master for a given job (*line 6*). In this case, the node adds a *backupJob* to the *backupJobList*, starts a Backup Job Manager, and transits to the `BACKUP` state (line 7-10).

In the `BACKUP` state four events can be received:

- *backupJobAssigned*, which is identical to that received in the `NOT_BACKUP` state, except for the absence of the state transition (*lines 14-15*).
- *primaryUpdate*, which informs this backup master that the primary master for a given job has changed (*line 17*). The *backupJob* in the *backupJobList* is updated accordingly and the same event is propagated to the corresponding Backup Job Manager (*lines 18-20*).
- *jobUpdate*, which is a message sent by the primary master each time a change happens to the information associated with a given job (*line 22*). The message includes three fields: the identifier of the job whose information has changed (*updatedJobId*); the (possibly updated) list of backup masters associated with *updatedJobId* (*updatedBackupMasterList*); the tasks associated with *updatedJobId* that have been updated (*updatedJobTaskList*). Note that the *updatedJobTaskList* does not contain whole tasks information, but only the information that has to be updated. In response to this event, the *backupMasterList* and all the relevant tasks in *jobTaskList* are updated (*lines 23-27*).
- *backupJobCompleted*, an event that notifies the completion of a given *backupJob* (*line 29*). In most cases this messages is sent by the primary master on job completion. In the other cases, it is an inner event sent by the Backup Job Manager of the same node, because the node has been elected as the new primary master. In both cases, the Backup Job Manager associated with the job is stopped and, if the *backupJobList* does not contain other jobs to execute, the node transits to the `NOT_BACKUP` state (*lines 30-33*).

```
1  Sequential Macro-State COORDINATION {
2   entry(){
3    transition to NOT_COORDINATOR;
4   }
5   State NOT_COORDINATOR {
6    on inner event coordinatorFailure(NodeId
         coordinatorId) {
7     transition to ELECTING_COORDINATOR;
8    }
9    //specific events based on the election
         algorithm
10  }
11  State ELECTING_COORDINATOR {
12   //specific events based on the election
         algorithm
13  }
14  State WAITING_COORDINATOR {
15   //specific events based on the election
         algorithm
16  }
17  Sequential Macro-State COORDINATOR {
18   entry() {
19    notify other master nodes that this node is
          the coordinator;
20    transition to ACTIVE;
21   }
22   State ACTIVE {
23    entry() {
24     int N = desired #master nodes - current
           #master nodes;
25     if(N>0) {
26      List<NodeIdType> idleSlaveList = search for
            N idle slave nodes;
27      foreach(NodeIdType s in idleSlaveList)
28       send event becomeMaster() to s;
29     }
30     else if(N<0) {
31      List<NodeIdType> idleMasterList = search
            for N idle master nodes;
32      foreach(NodeIdType m in idleMasterList)
33       send event becomeSlave() to m;
34     }
35     transition to IDLE;
36    }
37   }
38   State IDLE {
39    do() {
40     transition to ACTIVE after a timeout;
41    }
42   }
43  }
44 }
```

Figure 8: Pseudo-code describing the behavior of a master node performing Coordination activities.

Finally, the pseudo-code associated with the COORDINATION macro-state is described in Figure 8. Initially, a master transits to the NOT_COORDINATOR state; if a coordinator failure is detected by the network module, the master transits to the ELECTING_COORDINATOR state. Here, a distributed election algorithm starts. In particular, assuming to use the Bully algorithm [22], the following procedure takes place:

- A node in the ELECTING_COORDINATOR state sends an election message to all masters with higher identifier; if it does not receive a response from a master with a higher identifier within a time limit, it wins the election and passes to the COORDINATOR macro-state; otherwise, it transits to the WAITING_COORDINATOR state.
- A node in the WAITING_COORDINATOR waits until it receives a message from the new coordinator, then it transits to the NOT_COORDINATOR state. If it does not receive a message from the new coordinator before the expiration of a timeout, the node returns to the ELECTING_COORDINATOR state.

When a master enters the COORDINATOR macro-state, it notifies all the other masters that it became the new coordinator, and transits to the ACTIVE state (*lines 19-20*). Whenever the coordinator is ACTIVE, it performs its periodical network maintenance operations: if there is a lack of masters (i.e., the number of desired masters is greater than the current number of masters), the coordinator identifies a set of slaves that can be promoted to the master role, and sends a *becomeMaster* message to each of them (*lines 25-29*); if there is an excess of masters, the coordinator transforms a set of idle masters to slaves, by sending a *becomeSlave* message to each of them (*lines 30-34*).

4.2. Job and Task Management

As described in Section 3.2, MapReduce jobs and tasks are managed by processes called Job Managers and Task Managers, respectively. In particular, Job Managers are primary masters' processes, while Task Managers are slaves' processes. Moreover, masters run Backup Job Managers to manage those jobs they are responsible for as backup masters. In the following we describe the algorithmic behavior of Job Managers, Task Managers and Backup Job Managers.

Figure 9 describes the behavior of a Job Manager. The Job Manager includes only two states: ASSIGNMENT and MONITORING. As soon as it is started, the Job Managers transits to the ASSIGNMENT state (*line 4*).

```
1  StateDiagram JOB_MANAGER {
2   PrimaryJobType managedJob;
3   entry() {
4    transition to ASSIGNMENT;
5   }
6   State ASSIGNMENT {
7    entry() {
8     int B = desired #backup nodes - current
          #backup nodes;
9     int S = desired #slave nodes - current #slave
          nodes;
10    if(B>0) {
11     List<NodeIdType> backupList = search for B
           backup nodes;
12     foreach(NodeIdType b in backupList) {
13      send event backupJobAssigned(managedJob,
            nodeId) to b;
14      add b to managedJob.backupMasterList;
15     }
16    }
17    List<JobTaskType> assignedTaskList = empty
          list;
18    if(S>0) {
19     List<NodeIdType> slaveList = search for S
           slave nodes;
20     foreach(NodeIdType s in slaveList) {
21      List<Task> taskList = task assignment for s;
22      foreach (Task t in taskList){
23       JobTaskType jt(t, s, ASSIGNED);
24       update jt in managedJob.jobTaskList;
25       add jt to assignedTaskList;
26      }
27      send event taskAssigned(taskList, nodeId)
            to s;
28     }
29    }
30    if(B>0 or S>0) {
31     foreach(NodeIdType b in
           managedJob.backupMasterList)
32      send event jobUpdate(managedJob.job.jobId,
            managedJob.backupMasterList,
            assignedTaskList) to b;
33    }
34    transition to MONITORING;
35   }
36  }
37  State MONITORING {
38   do() {
39    transition to ASSIGNMENT after a timeout;
40   }
41   on inner event backupMasterFailure(NodeId
         backupId) {
42    remove backupId from
          managedJob.backupMasterList;
43    transition to ASSIGNMENT;
44   }
45   on inner event slaveFailure(NodeId
         failedSlaveId) {
46    foreach(JobTaskType t in
          managedJob.jobTaskList)
47     if(t.slaveId = failedSlaveId){
48      t.slaveId = null id;
49      t.taskStatus = NOT ASSIGNED;
50      update t in managedJob.jobTaskList;
51     }
52    transition to ASSIGNMENT;
53   }
54   on event taskCompleted(TaskIdType
         completedTaskId, TaskStatusType
         completedTaskStatus) {
55    JobTaskType t = tuple in
          managedJob.jobTaskList with
          taskId=completedTaskId;
56    t.taskStatus = completedTaskStatus;
57    List<JobTaskType> updatedTaskList = empty
          list;
58    update t in updatedTaskList;
59    foreach(NodeIdType b in
          managedJob.backupMasterList)
60     send event jobUpdate(managedJob.job.jobId,
           managedJob.backupMasterList,
           updatedTaskList) to b;
61    if(completedTaskStatus != SUCCESS)
62     transition to ASSIGNMENT;
63    else if(all tasks in managedJob.jobTaskList
          have completed with success)
64     send inner event
           jobCompleted(managedJob.job.jobId,
           managedJob.jobStatus) to
           NODE.MASTER.MANAGEMENT;
65   }
66  }
67 }
```

Figure 9: Pseudo-code describing the behavior of a Job Manager.

Briefly, the operations performed by a Job Manager in the `ASSIGNMENT` state are the following: *i*) it calculates the number of backup nodes (B) and slave nodes (S) needed to manage the job (*lines 8-9*); if $B > 0$, it identifies (up to) B masters that can be assigned the backup role, and sends a *backupJobAssigned* message to them (*lines 10-16*); similarly, if $S > 0$, it identifies S slaves, and assigns a subset of the tasks to each of them through a *taskAssigned* message (*lines 18-29*); finally, if $B > 0$ or $S > 0$, a *jobUpdate* event is sent to each backup master associated with the managed job (*lines 30-35*). The search for backup and slave nodes is performed by querying the discovery service.

In the `MONITORING` state four events can happen:

- An internal timeout elapses, causing a transition to the `ASSIGNMENT` state (*line 39*). This enforces the Job Manager to periodically check whether there are backup or slave nodes to be assigned.
- The failure of a backup master is notified by the network module (*backupMasterFailure*) (*line 41*). The failed backup master is removed from the *backupMasterList* and a transition to the `ASSIGNMENT` state is performed to find a suitable replacement (*lines 42-43*).
- The failure of a slave is detected (*slaveFailure*) (*line 45*), and is managed similarly to the *backupMasterFailure* event.
- A *taskCompleted* event is received from a slave (*line 54*). The event includes the identifier of the completed task (*completedTaskId*) and its status (*completedTaskStatus*). The Job Manager identifies the corresponding task from the *jobTaskList*, changes its status, and notifies all the backup masters

about this update through a *jobUpdate* message (*lines 55-60*). If the task has not completed with success, the Job Manager returns to the ASSIGNMENT state to reassign it (*lines 61-62*). Finally, if all the tasks have completed with success, it sends a *jobCompleted* message to the MANAGEMENT state.

Figure 10 describes the behavior of a Task Manager. When started, the Task Manager transits to two concurrent states: EXECUTION and PRIMARY_MONITORING (*line 4*). The EXECUTION state executes the assigned task; on task completion, it sends a *taskCompleted* message to the SLAVE state (*lines 8-10*).

```
1  StateDiagram TASK_MANAGER {
2   SlaveTaskType managedTask;
3   entry() {
4    concurrent transition to EXECUTION,
          PRIMARY_MONITORING;
5   }
6   State EXECUTION{
7    do(){
8     executes managedTask.task;
9     managedTask.task.taskStatus = task status;
10    send inner event
           taskCompleted(managedTask.task.taskId,
           managedTask.task.taskStatus) to
           NODE.SLAVE;
11   }
12  }
13  Sequential Macro-State PRIMARY_MONITORING{
14   entry(){
15    transition to CHECK_PRIMARY;
16   }
17   State CHECK_PRIMARY{
18    on inner event primaryMasterFailure(NodeId
           failedPrimaryId) {
19     transition to WAITING_PRIMARY;
20    }
21   }
22   State WAITING_PRIMARY{
23    do() {
24     transition to PRIMARY_UNAVAILABLE after a
            timeout;
25    }
26    on inner event primaryUpdate(JobIdType
           updatedJobId, NodeIdType
           updatedPrimaryId) {
27    transition to CHECK_PRIMARY;
28   }
29   }
30   State PRIMARY_UNAVAILABLE{
31    entry(){
32     send inner event
       taskCompleted(managedTask.task.taskId,
       FAILED) to NODE.SLAVE;
33   }
34  }
35 }
36 }
```

Figure 10: Pseudo-code describing the behavior of a Task Manager.

The PRIMARY_MONITORING macro-state allows to monitor the primary master responsible for the job of the managed task. In the case a failure of the primary master is detected (*line 18*), the Task Manager waits for the election of a new primary (*line 26*); if the election is not completed within a time limit, the Task Manager enters a PRIMARY_UNAVAILABLE state and sends a *taskCompleted* event to the SLAVE state (*line 32*).

Finally, the Backup Job Manager pseudo-code is shown in Figure 11. Initially, the Backup Job Manager enters the CHECK_PRIMARY state (*line 4*). If a primary master failure is detected, the backup master passes to the ELECTING_PRIMARY state and a procedure to elect the new primary master begins. This election algorithm is the same to elect the network coordinator. The new primary transits to the NEW_PRIMARY state.

```
1  StateDiagram BACKUP_JOB_MANAGER {
2   BackupJobType managedBackupJob;
3   entry() {
4    transition to CHECK_PRIMARY;
5   }
6   State CHECK_PRIMARY {
7    on inner event primaryMasterFailure(NodeId
          primaryId) {
8     transition to ELECTING_PRIMARY;
9    }
10   //specific events based on the election
          algorithm
11  }
12  State ELECTING_PRIMARY {
13   //specific events based on the election
          algorithm
14  }
15  State WAITING_PRIMARY {
16   on inner event primaryUpdate(JobIdType
          updatedJobId, NodeIdType
          updatedPrimaryId) {
17    transition to CHECK_PRIMARY;
18   }
19   //specific events based on the election
          algorithm
20  }
21  State NEW_PRIMARY {
22   entry() {
23    notify other backup masters that this node is
           the new primary master;
24    remove nodeId from
           managedBackupJob.backupMasterList;
25    send inner event
           backupJobCompleted(managedBackupJob.
           job.jobId) to NODE.MASTER.RECOVERY;
26    send inner event
           jobReassigned(managedBackupJob) to
           NODE.MASTER.MANAGEMENT;
27   }
28  }
29 }
```

Figure 11: Pseudo-code describing the behavior of a Backup Job Manager.

The Backup Job Manager performs the following operations in the NEW_PRIMARY state (*lines 23-26*): informs the other backup masters that this node became the new primary master; removes itself from the *backupMasterList* of the managed *backupJob*; sends a *backupJobCompleted* message to the RECOVERY state; sends a *jobReassigned* message to the MANAGEMENT state.

4.3. User Behavior

```
1  StateDiagram USER {
2   UserIdType userId;
3   List<UserJobType> userJobList;
4   on user event submitJob(Job submittedJob) {
5    UserJobType t(submittedJob);
6    add t to userJobList;
7    NodeIdType masterId = search for a master node;
8    send event jobAssigned(submittedJob, userId)
          to masterId;
9   }
10  on event primaryUpdate(JobId updatedJobId,
         NodeIdType updatedPrimaryId) {
11    UserJobType t = tuple in userJobList with
           job.jobId=updatedJobId;
12    t.primaryId = updatedPrimaryId;
13   }
14   on event jobCompleted(JobIdType completedJobId,
          JobStatusType completedJobStatus) {
15    UserJobType t = tuple in userJobList with
           job.jobId=completedJobId;
16    t.jobStatus = completedJobStatus;
17   }
18  }
```

Figure 12: Pseudo-code describing the behavior of a user node.

We conclude this section by describing the behavior of user nodes (see Figure 12). Each user includes an identifier (*userId*) and a list of jobs submitted (*userJobList*). Three events are possible:

- *submitJob*, a user-generated event (for this reason not listed in Table 1) which requires the submission of a new job (*line 4*). The job is added to the *userJobList*; then, a master is searched and the job is assigned to it using a *jobAssigned* message (*lines 5-8*). The search for a master is performed by querying the discovery service for nodes whose *role* is MASTER.
- *primaryUpdate*, which informs the user that the primary master has changed (*line 10*). The message includes the identifier of the job whose primary master has changed, and the identifier of the new primary master (*updatedPrimaryId*). The user identifies the job in the *userJobList*, and changes its *primaryId* to *updatedPrimaryId* (*lines 11-12*).
- *jobCompleted*, which notifies the user that a job has completed its execution (*line 14*). The message includes the identifier of the job and its status (*completedJobStatus*). The job is identified in the *userJobList*, and its status is changed to *completedJobStatus* (*lines 15-16*).

5. Implementation

We implemented a prototype of the P2P-MapReduce framework using the JXTA framework [23]. JXTA provides a set of XML-based protocols that allow computers and other devices to communicate and collaborate in a peer-to-peer fashion. Each peer provides a set of services made available to other peers in the network. Services are any type of programs that can be networked by a single or a group of peers.

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. For example, if a user node wants to search for all the available masters, it submits a query to the JXTA Discovery Service asking for all the advertisements whose field *role* is equal to `MASTER`. Note that *M-net* and *S-net*, introduced in Section 3, are "logical" networks in the sense that queries to *M-net* (or *S-net*) are actually submitted to the whole *MapReduceGroup* but restricted to nodes having their field *role* equal to `MASTER` (or `SLAVE`) using the SRDI mechanisms.

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.

In JXTA pipes it is possible to rebind one endpoint without affecting the other endpoint. We use this feature when a failure occurs: in fact, the new primary master can bind the pipes that were previously used by the old primary master, without affecting the entities connected at the other endpoint (i.e., the user node and the slave nodes).

We conclude this section briefly describing the software modules inside each node and how those modules interact each other in a P2P-MapReduce network. Figure 13 shows such modules and interactions using the UML Deployment/Component Diagram formalism.

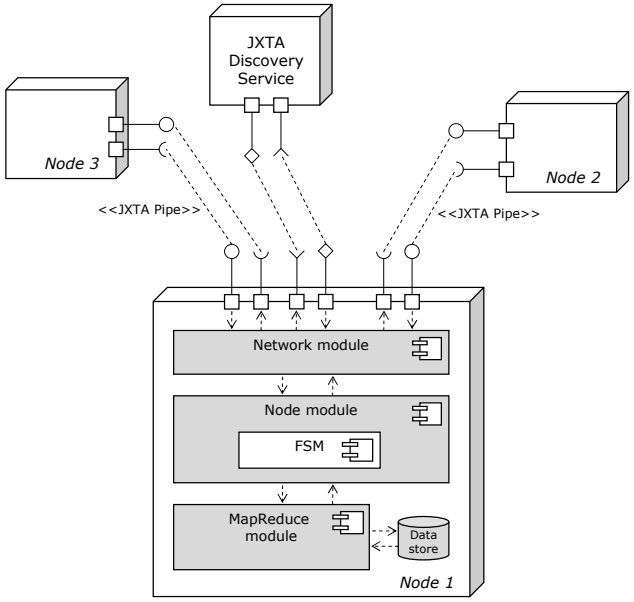


Figure 13: UML Deployment/Component Diagram describing the software modules inside each node and the interactions among nodes.

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 4, steering the behavior of the node in response to inner events 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 [2].

While the current implementation is based on JXTA for the Network layer and on Hadoop for the MapReduce layer, the layered approach described in Figure 13 is thought to be independent from a specific implementation of the Network and MapReduce modules. In other terms, it may be possible to adopt alternative technologies for the Network and MapReduce layers without affecting the core implementation of the Node module.

6. System Evaluation

A set of experiments has been carried out to evaluate the behavior of the P2P-MapReduce framework compared to a centralized implementation of MapReduce in the presence of different levels of churn. In particular, we focused on comparing P2P and centralized implementations in terms of fault tolerance, network traffic, and scalability.

The remainder of this section is organized as follows. Section 6.1 describes the experimental setup and methodology. Section 6.2 compares fault tolerance capability of P2P and centralized MapReduce implementations. Section 6.3 analyzes the systems in terms of network traffic. Section 6.4 concentrates on the scalability of the systems. Finally, 6.5 summarizes the main results of the present evaluation.

6.1. Experimental Setup and Methodology

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 like that introduced in Section 2.1. While the P2P-MapReduce prototype allowed us to perform functional testing of the system mechanisms on a small scale, the simulator allowed us to perform non-functional testing (i.e., performance evaluation) on large networks (thousands of nodes), which represent our reference scenario.

The simulator models joins and leaves of nodes and job submissions as Poisson processes; therefore, the interarrival times of all the join, leave and submission events are independent and obey an exponential distribution with a given rate. This model has been adopted in literature to evaluate several P2P systems (see, for example, [24] and [25]), for its ability to approximate real network dynamics reasonably well.

Table 2 shows the input parameters used during the simulation.

Table 2: Simulation parameters

Symbol	Description	Values
N	Initial number of nodes in the network	5000, 7500, 10000, 15000, 20000, 30000, 40000
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
JT	Job type	A, B, C (see Table 3)

As shown in the table, we simulated MapReduce systems having an initial size ranging from 5000 to 40000 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 and number of tasks. To evaluate the behavior of the system under different loads, we defined three job types (JT), A, B and C, as detailed in Table 3.

Table 3: Job types and associated parameters (average values)

Type	Total computing time (hours)	Number of tasks
A	100	200
B	150	300
C	200	400

Hence, the jobs submitted to the system simulator belong either to type A, B or C. 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.

In order to compare the P2P-MapReduce system with a centralized MapReduce implementation, we analyzed several scenarios characterized by different combinations of simulation parameters (i.e., network size, leaving rate, job type). For each scenario under investigation, the simulation ends after the completion of 500 jobs. At the end of the simulation, we collect four 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.
- The *number of messages* exchanged through the network during the whole simulation process.
- The *amount of data* associated with all the messages exchanged through the network.

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 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.

6.2. Fault Tolerance

As mentioned earlier, one of the goals of our simulations is to compare the P2P and centralized implementations in terms of fault tolerance, i.e., the percentage of failed jobs and the corresponding percentage of lost computing time. The results discussed in this section have been obtained considering the following scenario: $N = 10000$ and LR ranging from 0.025 to 0.4. Figure 14 compares the percentage of failed jobs in such scenario, for each of the job types defined above: (a) JT=A; (b) JT=B; (c) JT=C.

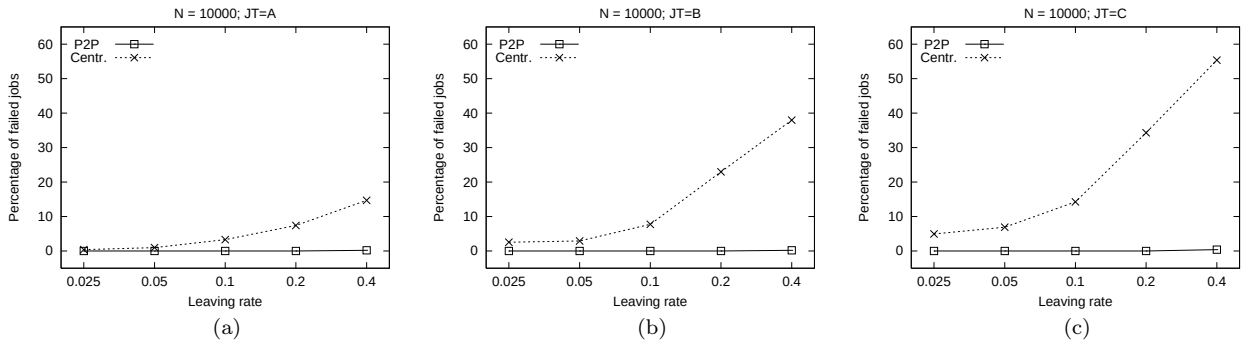
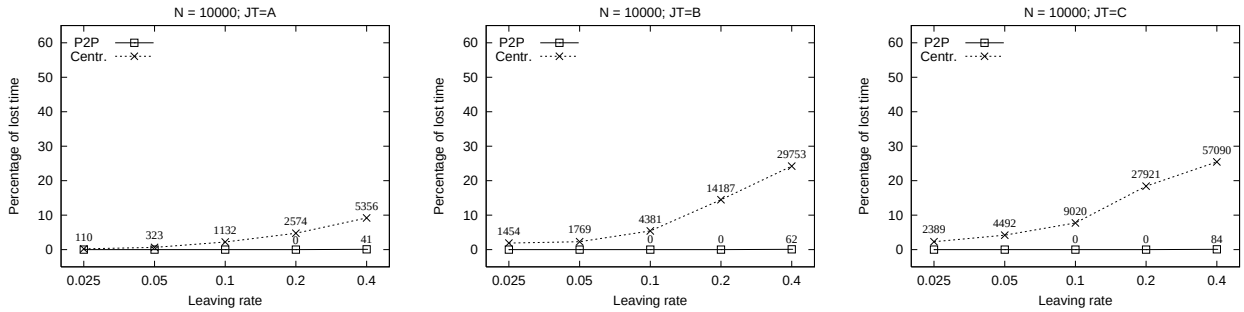


Figure 14: Percentage of failed jobs in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

As expected, with the centralized MapReduce implementation the percentage of failed jobs significantly increases with the leaving rate, for each job type. For example, when $JT = B$, the percentage of failed jobs passes from 2.5 when $LR = 0.025$, to 38.0 when $LR = 0.4$. Moreover, we can observe that, fixed the value of LR, the percentage of failed jobs increases from JT=A to JT=B, and from JT=B to JT=C. For example, with LR=0.1, the percentage of failed jobs is 3.3 for JT=A, 7.8 for JT=B, and 14.2 for JT=C. This is motivated by the fact that longer jobs (as jobs of type C are compared to jobs of type B and A) are statistically more subject to be affected by a failure of the associated master.

In contrast to the centralized implementation, the P2P-MapReduce framework is limitedly affected by job failures. In particular, for any job type, the percentage of failed jobs is 0% for $LR \leq 0.2$, while it is ranges from 0.2% to 0.4% for $LR = 0.4$, even if only one backup master per job is used. It is worth recalling here 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, as discussed in Section 4.

Figure 15 reports the percentage of lost computing time in centralized and P2P implementations related to the same experiments of Figure 14, for different combinations of network sizes, leaving rates and job types. 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.



(a) (b) (c)

Figure 15: Percentage of lost time in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C. The numbers in correspondence of each graph point represent the amount of lost computing time expressed in hours (some zero values are omitted for readability).

The lost computing time follows a similar trend as the percentage of failed jobs, and it results affected by the same dependence from the job type. For example, when LR=0.4, the percentage of lost computing time for the centralized system passes from 9.2 for JT=A to 25.5 for JT=C, 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 and JT=C), the centralized system loses 57090 hours of computation, while the amount of lost computing time with the P2P-MapReduce system is only 84 hours.

6.3. Network Traffic

This section compares the P2P and centralized implementations of MapReduce in terms of network traffic, i.e., the number of messages exchanged through the network during the whole simulation, and the corresponding amount of data expressed in MBytes. The amount of data is obtained by summing the sizes of all the messages that are exchanged through the network.

In order to calculate the size of each messages, Table 4 lists the sizes of all the basic components that may be found in a message.

Table 4: Sizes of message components

Message components	Size (Bytes)
Header	260
Identifier (e.g., jobId, taskId)	4
Code (job.code, task.code)	4000
URL (e.g., job.input, job.output)	150
Integer (e.g., job.M, job.R)	4
Status (e.g., task.type, jobStatus)	1

Each message includes a *header* that represents the fixed amount of traffic each message generates independently from the specific payload. Its size has been determined experimentally by measuring the average amount of traffic generated to transfer an empty message from a host to another host using a TCP socket. The sizes for *identifier*, *integer* and *status* variables are those used in common system implementations. The size of the *code* component is the average code size observed on a set of MapReduce applications; the size of the *URL* component has been calculated similarly.

For example, let's calculate the size of a *jobAssigned* message. From Table 1, we know that a *jobAssigned* message includes three parts: 1) one *Job* tuple; 2) one *UserIdType* variable; 3) one *header* (implicitly present in each message). While the size of the second and third parts are known (respectively 4 and 260 Bytes),

the size of the first part must be calculated as the sum of each of its fields. From Section 3.1, a *Job* tuple includes the following fields: *jobId* (4 Bytes), *code* (4000 Bytes), *input* (150 Bytes), *output* (150 Bytes), M (4 Bytes) and R (4 Bytes), for a total of 4312 Bytes. Therefore, the size of a *jobAssigned* message is equal to 4576 Bytes.

The size of messages that include lists, like *taskAssigned*, is calculated taking into account the actual number of elements in the list, and the size of each such elements. For the messages generated by the discovery service and by the election algorithms, we proceeded in the same way. We just mention that most of such messages are very small since they include only a few fields.

For the purpose of the evaluation presented below, we distinguish four categories of messages:

- *Management*: messages exchanged among nodes to manage jobs and tasks execution. Referring to Table 1, the management messages are those associated with the following (not inner) events: *jobAssigned*, *jobCompleted*, *taskAssigned* and *taskCompleted*.
- *Recovery*: messages exchanged among primary masters and their backups to dynamically replicate job information (*backupJobAssigned*, *backupJobCompleted*, *jobUpdate* and *primaryUpdate*), as well as to elect a new primary in the case of a master failure (messages specific to the election algorithm used).
- *Coordination*: messages generated by the coordinator to perform network maintenance operations (*becomeMaster* and *becomeSlave*), as well as to elect the new coordinator (specific to the election algorithm).
- *Discovery*: messages generated to publish and search information about nodes using the JXTA Discovery Service.

Management messages are present both in the P2P and in the centralized case, since they are generated by the standard execution mechanisms of MapReduce. In contrast, recovery, coordination and discovery operations are performed only by the P2P-MapReduce system, therefore the corresponding messages are not present in the centralized case.

We start focusing on the total traffic generated, without distinguishing the contribution of the different categories of messages, in order to obtain an aggregate indicator of the overhead generated by the two systems. As for the previous section, the results presented here are obtained considering a network with $N = 10000$ and LR ranging from 0.025 to 0.4. In particular, Figure 16 compares the total number of messages exchanged for three job types: (a) JT=A; (b) JT=B; (c) JT=C.

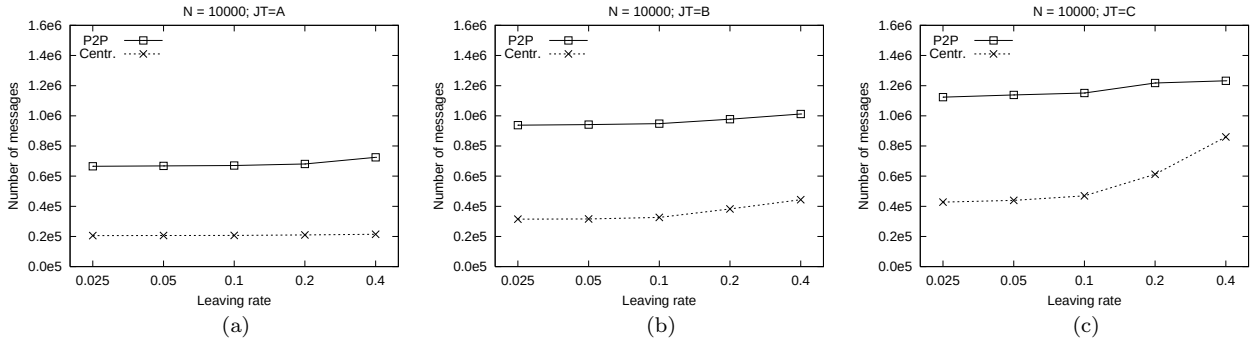


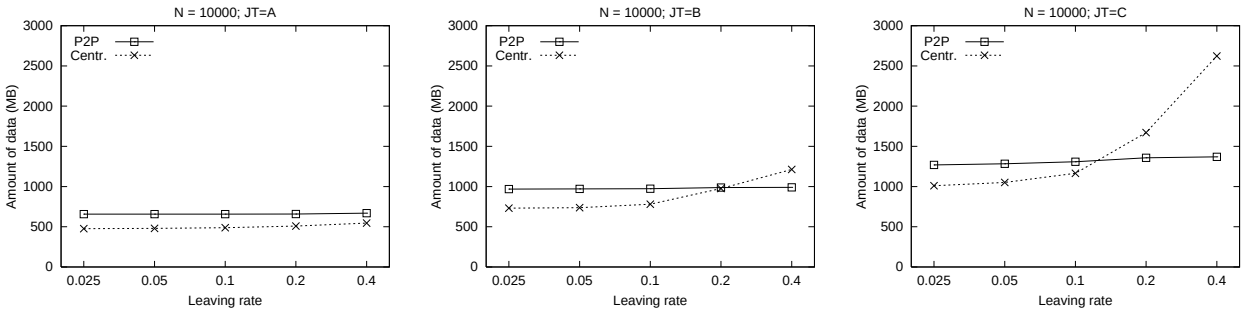
Figure 16: Number of messages exchanged in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

As shown by the graphs, the total number of messages generated by the P2P system is higher than that generated by the centralized system in all the considered scenarios. This is mainly due to the presence in the P2P system of discovery messages, which are not present in the centralized system. We will discuss later in this section the impact of the different types of messages also in terms of amounts of data exchanged.

We observe that in both cases - P2P and centralized - the number of messages increases with the leaving rate. This is due to the fact that by increasing the leaving rate also the number of failed jobs increases; since failed jobs are resubmitted, a corresponding increase in the number of management messages is produced. As shown in the figure, such increase is higher with the centralized MapReduce implementation, being higher the number of failed jobs compared to P2P-MapReduce.

The increase in the number messages is higher for heavy jobs (i.e., JT=B or JT=C), since their failure requires the reassignment of a greater number of tasks, thus producing a higher number of management messages. For example, Figure 16c shows that with JT=C, the number of messages for the P2P case passes from 1.12 millions when $LR = 0.025$, to 1.23 millions when $LR = 0.04$, which corresponds to an increase of about 10%. In contrast, the number of messages for the centralized case passes from 0.43 to 0.86 millions, which corresponds to an increase of 100%.

Figure 17 shows the amount of data associated with all the messages exchanged through the network.



(a) (b) (c)

Figure 17: Data exchanged (MBytes) in a network with 10000 nodes: (a) JT=A; (b) JT=B; (c) JT=C.

For the P2P case, the amount of data for a given job type increases very little with the leaving rate. In fact, the few jobs that fail even with the higher leaving rate, produce a relatively little number of additional management messages and so they have a limited impact in terms of amount of data exchanged. For the centralized case, the amount of data for a given job type increases significantly with the leaving rate, since the percentage of failed jobs grows faster than the P2P case.

It is interesting to observe that, in some scenarios, the amount of data exchanged in the centralized implementation is greater than the amount of data exchanged in P2P-MapReduce. In our simulations this happens when $LR > 0.2$ for JT=B (see Figure 17b), and when $LR > 0.1$ for JT=C (see Figure 17c). In particular, with $LR = 0.4$ and JT=C, the amount of data exchanged is equal to 1369 MB for the P2P system and 2623 MB for the centralized implementation.

We conclude the traffic analysis by showing what is the contribution of the different types of messages (management, recovery, coordination and discovery) in terms of number of messages and corresponding amount of data exchanged through the network. Figure 18 presents the results of such analysis for a network with $N = 10000$ and JT=C.

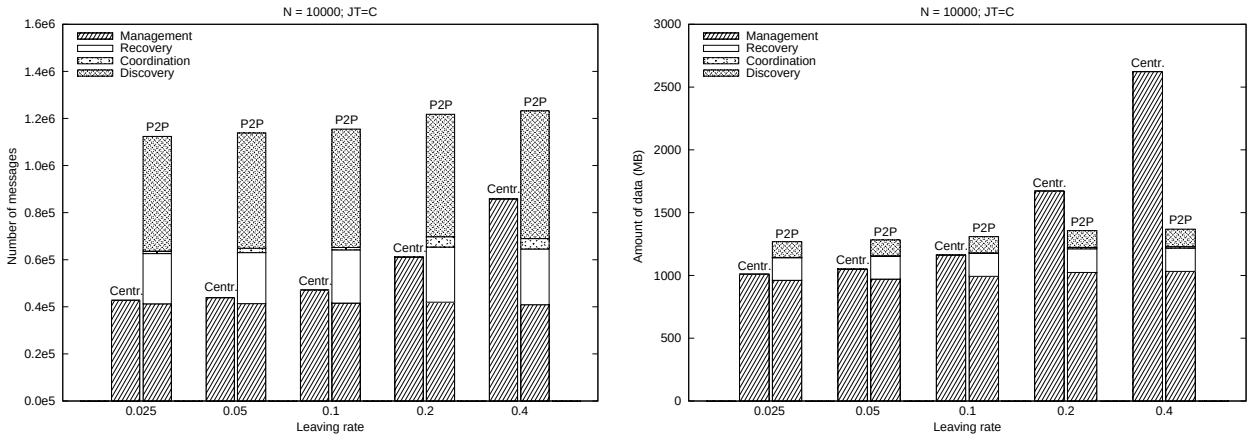


Figure 18: Detailed traffic in a network with $N = 10000$ and JT=C: (a) number of messages; (b) amount of data.

As stated earlier, in the centralized case only management messages are generated. Therefore, their number and corresponding amount of data are the same already shown in Figures 16c and 17c. We just highlight that, for high values of leaving rate, the number of messages and the amount of data grows significantly.

For the P2P case, we observe that the management messages represent only one third of the total number of messages. Discovery messages represent 40% in terms of number of messages, but only 10% in terms of amount of data. This is due to the fact that the size of discovery messages is very small, as mentioned earlier, and so they do not produce a significant network overhead. Also recovery and coordination messages have limited impact on the total network traffic, both in terms of number of messages and amount of data.

6.4. Scalability

We finally conducted a set of simulations to evaluate the behaviors of the P2P and centralized MapReduce implementations by varying the network size. In particular, Figure 19 compares P2P and centralized systems with $LR = 0.1$, JT=C, and N ranging from 5000 to 40000, in terms of: (a) percentage of failed jobs; (b) percentage (and absolute amount) of lost computing time; (c) number of messages; (d) amount of data exchanged.

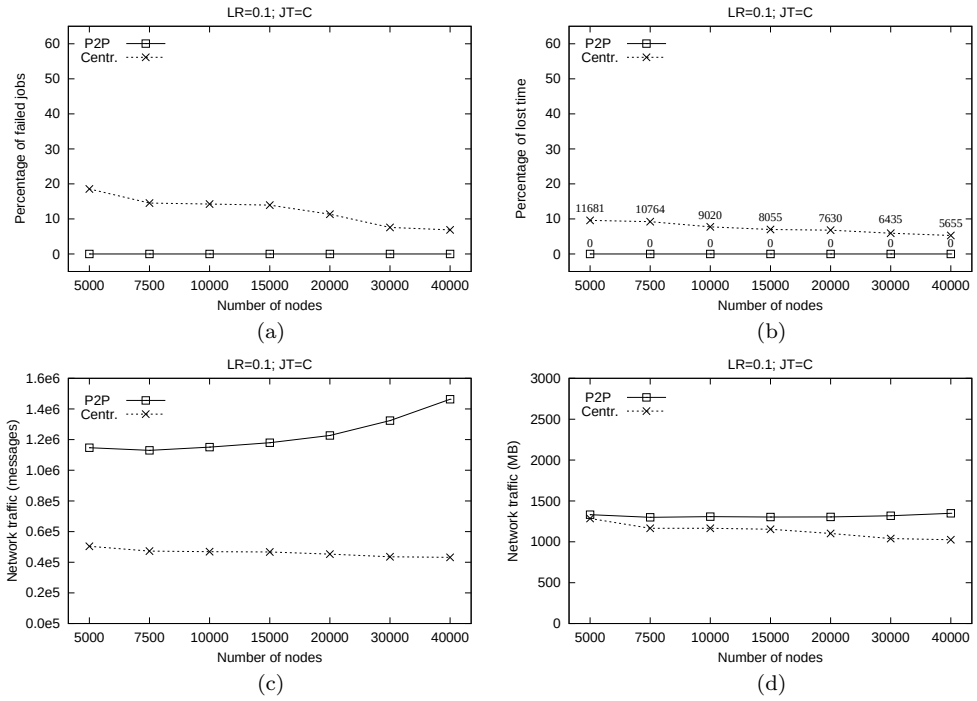


Figure 19: Comparison of P2P and centralized systems with $LR = 0.1$, JT=C, and N ranging from 5000 to 40000: (a) percentage of failed jobs; (b) percentage (and absolute amount) of lost computing time; (c) number of messages; (d) amount of data exchanged.

As shown in Figure 19a, the percentage of failed jobs for the centralized case slightly decreases when the network size increases. This is due to the fact that jobs complete faster in larger networks, since the number of slaves increases and the job type is fixed (JT=C in our case), and so they are less affected by failure events. On the other hand, in the P2P case the percentage is always zero, independently from the network size. For the percentage of lost computing time (see Figure 19b) a similar trend can be noted.

Regarding network traffic (see Figures 19c and 19d), we observe that, in the P2P case, the number of messages slightly increases with the number of nodes. This is due to the higher number of discovery and coordination messages that are generated in larger networks. However, in terms of amount of data this increment is negligible. Also for the centralized system the variation is not significant passing from 5000 to 40000 nodes.

6.5. Remarks

The results discussed above confirm the fault tolerance level provided by the P2P-MapReduce framework compared to a centralized implementation of MapReduce, since in all the scenarios analyzed the amount of failed jobs and the corresponding lost computing time was negligible. The centralized system, on the contrary, was significantly affected by high churn rates, producing critical levels of failed jobs and lost computing time.

The experiments have also shown that the P2P-MapReduce system generates more messages than a centralize MapReduce implementation. However, the difference between the two implementations reduces as the leaving rate increases, particularly in the presence of heavy jobs. Moreover, if we compare the two systems in terms of amount of data exchanged, we see that in many cases the P2P-MapReduce system is more efficient than the centralized implementation.

We have finally assessed the behavior of P2P-MapReduce with different network sizes. The experimental results showed that the overhead generated by the system is not significantly affected by an increase of the network size, thus confirming the good scalability of our system.

In summary, the experimental results show that even if the P2P-MapReduce system consumes in most cases more network resources than a centralized implementation of MapReduce, it is far more efficient in job management since it minimizes the lost computing time due to jobs failures.

7. Conclusion

The P2P-MapReduce framework 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 paper provided a detailed description of the mechanisms that are at the base of the P2P-MapReduce system, presented a prototype implementation based on the JXTA peer-to-peer framework, and an extensive performance evaluation of the system in different network scenarios.

The experimental results showed that, differently from centralized master-server implementations, the P2P-MapReduce framework does not suffer from job failures even in the presence of very high churn rates, thus enabling the execution of reliable MapReduce applications in dynamic Cloud infrastructures.

The P2P-MapReduce prototype is available as open-source software from `http://grid.deis.unical.it/p2p-mapreduce`.

Bibliography

[1] 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.

[2] Hadoop. `http://hadoop.apache.org` (site visited December 2010).

[3] Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce` (site visited December 2010).

[4] Mapper API for Google App Engine. `http://googleappengine.blogspot.com/2010/07/introducing-mapper-api.html` (site visited December 2010).

[5] J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". *Communications of the ACM*, 51(1), 107-113, 2008.

[6] Google's Map Reduce. `http://labs.google.com/papers/mapreduce.html` (site visited December 2010).

[7] F. Marozzo, D. Talia, P. Trunfio. "Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model". 1st Workshop on Cloud Computing and its Applications (CCA'08), Chicago, USA, 2008.

[8] F. Marozzo, D. Talia, P. Trunfio. "A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments". In: N. Antonopoulos, L. Gillam (eds.), Cloud Computing: Principles, Systems and Applications, Springer, Chapter 7, 113-125, 2010.

[9] 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.

[10] Gridgain. `http://www.gridgain.com` (site visited December 2010).

[11] Skynet. `http://skynet.rubyforge.org` (site visited December 2010).

[12] MapSharp. `http://mapsharp.codeplex.com` (site visited December 2010).

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, G. Fox. "Twister: A Runtime for Iterative MapReduce". 1st International Workshop on MapReduce and its Applications (MAPREDUCE'10), Chicago, USA, 2010.

[14] Disco. `http://discoproject.org` (site visited December 2010).

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] G. Fedak, H. He, F. Cappello. "BitDew: A Data Management and Distribution Service with Multi-Protocol and Reliable File Transfer". *Journal of Network and Computer Applications*, 32(5), 961975, 2009.

[22] H. Garcia-Molina. "Election in a Distributed Computing System". *IEEE Transactions on Computers*, 31(1), 48-59, 1982.

[23] L. Gong. "JXTA: A Network Programming Environment". *IEEE Internet Computing*, 5(3), 88-95, 2001.

[24] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. “Chord: a scalable peer-to-peer lookup protocol for internet applications”. IEEE/ACM Transactions on Networking, 11(1), 17-32, 2003.
[25] K. Albrecht, R. Arnold, M. Gahwiler, R. Wattenhofer. “Join and Leave in Peer-to-Peer Systems: The Steady State Statistics Service Approach”. Technical Report 411, ETH Zurich, 2003.