A Framework for Managing MapReduce Applications in Dynamic Distributed Environments

Fabrizio Marozzo
DEIS, University of Calabria
Rende (CS), Italy
Email: fmarozzo@deis.unical.it

Domenico Talia
ICAR-CNR
DEIS, University of Calabria
Rende (CS), Italy
Email: talia@deis.unical.it

Paolo Trunfio
DEIS, University of Calabria
Rende (CS), Italy
Email: trunfio@deis.unical.it

***Abstract*—MapReduce is a programming model widely used in data centers for processing large data sets in a highly parallel way. Current MapReduce systems are based on master-slave architectures that do not cope well with dynamic node participation, since they are mostly designed for conventional parallel computing platforms. On the contrary, in Internet-based computing environments, node churn and failures - including master failures - are likely to happen since nodes join and leave the network at an unpredictable rate. The goal of this work is enabling the use of MapReduce in dynamic distributed environments so as to combine the effectiveness of a well-established programming model with the scalability of a large-scale computing infrastructure. This paper presents an adaptive MapReduce framework, called P2P-MapReduce, which exploits a peer-to-peer model to manage intermittent node participation, master failures and job recovery in a decentralized but effective way, so as to provide a more robust MapReduce middleware that can be effectively exploited in Internet-scale dynamic distributed environments.**

***Keywords*-Internet computing; Peer-to-peer computing; MapReduce**

I. Introduction

Internet-based computing approaches have proven effective over the years as demonstrated by projects like SETI@home [1], Folding@home [2], and Boinc [3]. The key principle of such projects is the use of large collections of unreliable hosts, available through the Internet, to run applications that need huge amounts of resources (processing and/or storage) to be executed. As compared to conventional high-performance computing systems, Internet-wide decentralized systems are more scalable, motivating their use as efficient platforms for parallel and distributed computing applications.

Despite many successful experiences, Internet-based computing systems are still far from being generally accepted as a viable alternative to more conventional computing platforms, like clusters and massively parallel machines. One of the main reasons for this is the lack of programming models for large-scale distributed computing systems that are both effective and flexible (in terms of supported applications) as those adopted in conventional systems. One of the most successful programming models currently used in clusters and data centers is MapReduce [4]. 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. The success of this model derives from its simplicity: indeed, it is relatively simple to implement parallel versions of a wide range of data-intensive applications using MapReduce.

The goal of our work is enabling the use of MapReduce in dynamic Internet-based environments, so as to combine the effectiveness of a well-established programming model with the scalability of a large-scale computing infrastructure. Current MapReduce implementations (e.g., Google's MapReduce [5] and Apache Hadoop [6]) 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 to 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 master failures are not explicitly managed as designers consider failures unlikely in reliable centralized environments.

On the contrary, node churn and failures - including master failures - are likely to happen in dynamic distributed environments where computing nodes may join and leave the system at an unpredictable rate. Therefore, providing effective mechanisms to manage master churn and failures is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in large-scale dynamic environments where current MapReduce implementations could be unreliable. This paper presents an adaptive MapReduce framework, called P2P-MapReduce, which exploits a peer-to-peer model to manage intermittent node participation, master failures and job recovery in a decentralized but effective way, so as to provide a more robust MapReduce middleware that can be effectively exploited in Internet-scale dynamic distributed environments.

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. This paper extends our previous work by providing a more detailed description of the P2P-MapReduce implementation, as well as an extensive evaluation of its performance in different scenarios.

The remainder of this paper is organized as follows. Section II provides a background on the MapReduce programming model. Section III describes the P2P-MapReduce architecture and its implementation. Section IV evaluates the performance of P2P-MapReduce compared to a centralized implementation of MapReduce. Finally, Section V concludes the paper.

II. 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 [4]. 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).

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

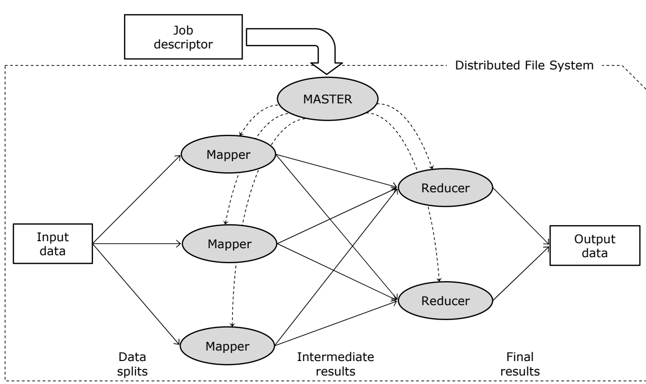

Figure 1. Execution phases in a generic MapReduce application

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 normally is a directory in a distributed file system.
2) According to the job descriptor, the master starts a number of mapper and reducer processes on different machines. At the same time, it starts a process that reads the input data from its location, partitions that data into a set of splits, and distributes those splits to the various mappers.
3) After receiving its data partition, each mapper process executes the *map* function (provided as part of the job descriptor) to generate a list of intermediate key/value pairs. Those pairs are then grouped on the basis of their keys.
4) All pairs with the same keys are assigned to the same reducer process. Hence, each reducer process executes the *reduce* function (defined by the job descriptor) which merges all the values associated to the same key to generate a possibly smaller set of values.
5) The results generated by each reducer process are then collected and delivered to a location specified by the job descriptor, so as to form the final output data.

Besides the original MapReduce implementation by Google [5], several other MapReduce implementations have been realized within other systems, including Hadoop [6], GridGain [9], Skynet [10], MapSharp [11] and Disco [12]. Another system sharing most of the design principles of MapReduce is Sector/Sphere [13], 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 and flexible implementations of MapReduce. As an example, Zaharia et al. [14] studied how to improve the Hadoop's scheduler in heterogeneous environments, by designing a new scheduling algorithm that significantly improves response times in heterogeneous settings. Another example is the Hadoop Online Prototype (HOP) proposed by Condie et al. [15], which modifies the Hadoop MapReduce framework to supports online aggregation, allowing users to see early returns from a job as it is being computed.

Several applications of the MapReduce paradigm have been demonstrated. Ref. [16] 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. [6] 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.

III. P2P-MapReduce: Architecture and Implementation

The objective of the P2P-MapReduce framework is twofold: i) handling master failures by dynamically replicating the job state on a set of backup masters; ii) supporting

MapReduce applications over dynamic distributed environments composed by nodes that join and leave the system at unpredictable rates.

To achieve these goals, P2P-MapReduce exploits the peer-to-peer paradigm by defining an adaptive architecture in which each node can act either as a master or a 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 and its current implementation.

A. Architecture

The P2P-MapReduce architecture includes three basic roles, shown in Figure 2: user (U), master (M) and slave (S). Master nodes and slave nodes form two logical peer-to-peer 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 III-B.

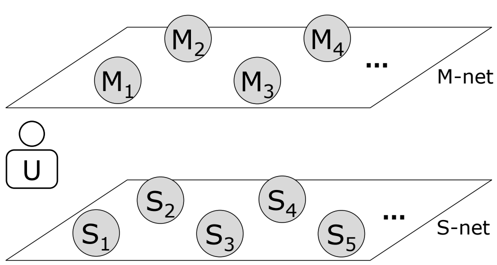

Figure 2. Basic architecture of a P2P-MapReduce network

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 Figure 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 Figure 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 Figure 3, the apex "B" to nodes M_2 and M_3 indicates the backup function). This implies that whenever the job state changes, M_1 backs up 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 binds to the connections that were previously associated to M_1, and 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.

B. Implementation

We implemented a prototype of the P2P-MapReduce framework using the JXTA framework [18]. 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

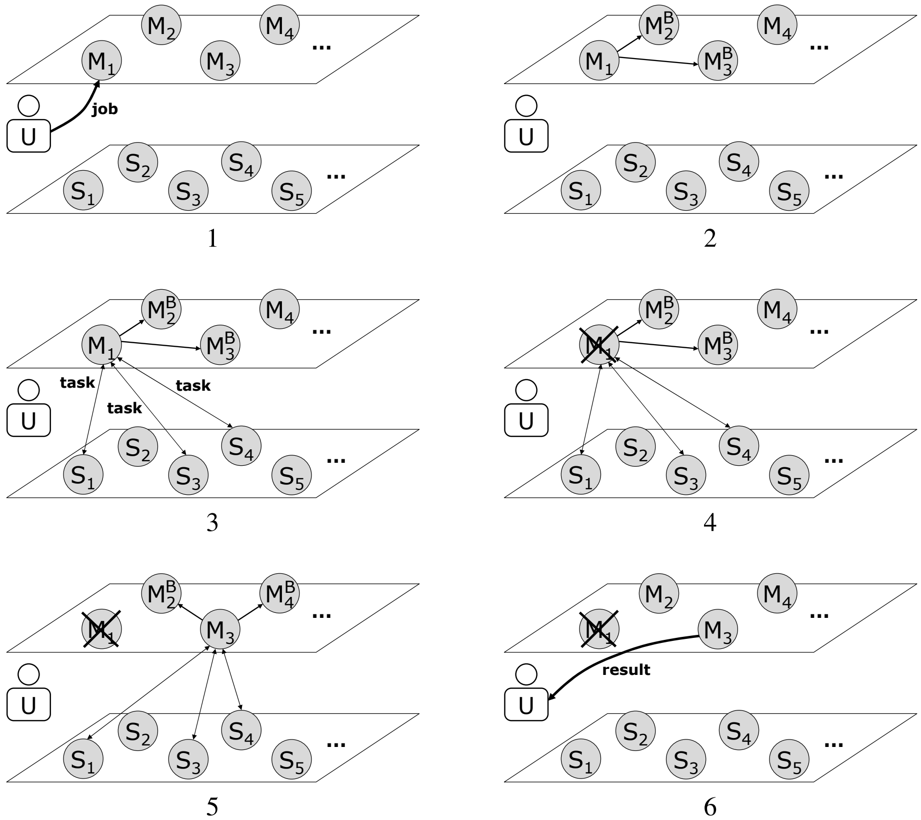

Figure 3. Steps performed to submit a job and to recover from a master failure

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.

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.

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 Figure 2, 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 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.

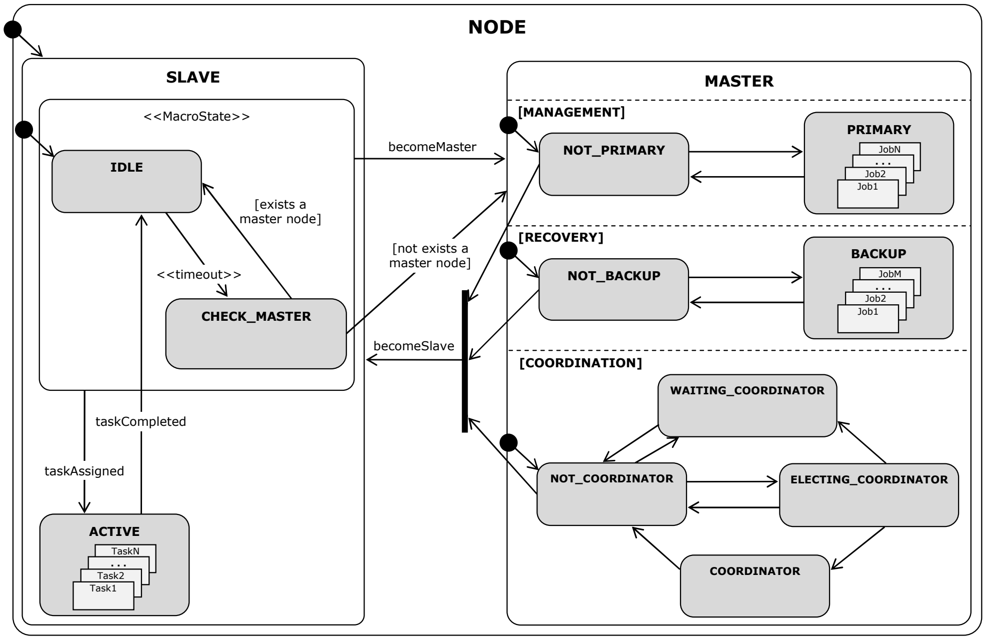

Figure 4. UML State Diagram describing the behavior of a generic node

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 the *coordinator*), which is elected for this purpose among all masters using the "bully" election algorithm [17]. 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-step example presented in Section III-A 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 descriptor.
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 *SlavePipes*), chooses (and connects 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.

2) State diagram:

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

The state diagram includes two macro-states, `SLAVE` and `MASTER`, which describe the two roles that can be assumed by each node. The `SLAVE` macro-state has three states, `IDLE`, `CHECK_MASTER` and `ACTIVE`, which represent respectively: a slave waiting for task assignment; a slave checking the existence of a master; a slave executing 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 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 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 mentioned in Section III-B1.

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.

3) Software modules:

This section briefly describes the software modules inside each node and how those modules interact each other in a P2P-MapReduce network. Figure 5 shows such modules and interactions using the UML Deployment/Component Diagram formalism.

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

- The Network module is in charge of the interactions with the other nodes using the pipe communication mechanisms provided by the JXTA framework. 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 local events or messages coming from remote nodes (e.g., tasks assignments, discovery requests, etc.).
- 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 [6].

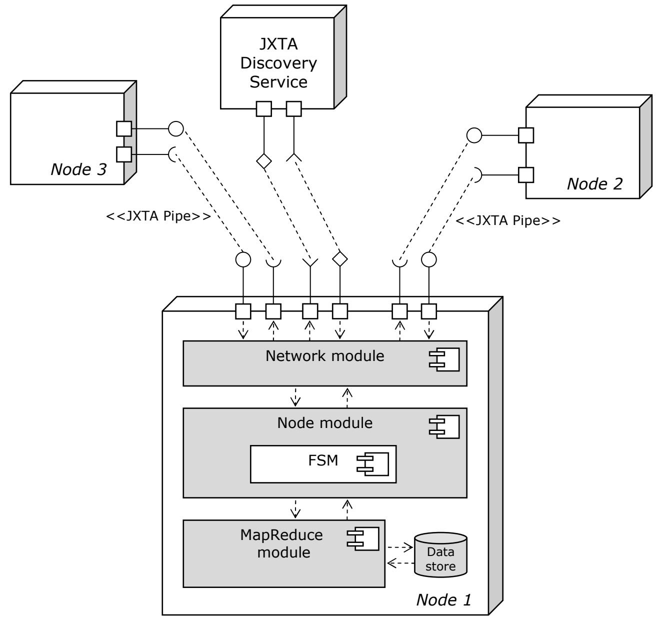

Figure 5. UML Deployment/Component Diagram describing the software modules inside each node and the interactions among nodes

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 5 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.

IV. Evaluation

We carried out a set of experiments to evaluate the behavior of the P2P-MapReduce framework compared to a centralized implementation of MapReduce, considering dynamic node participation.

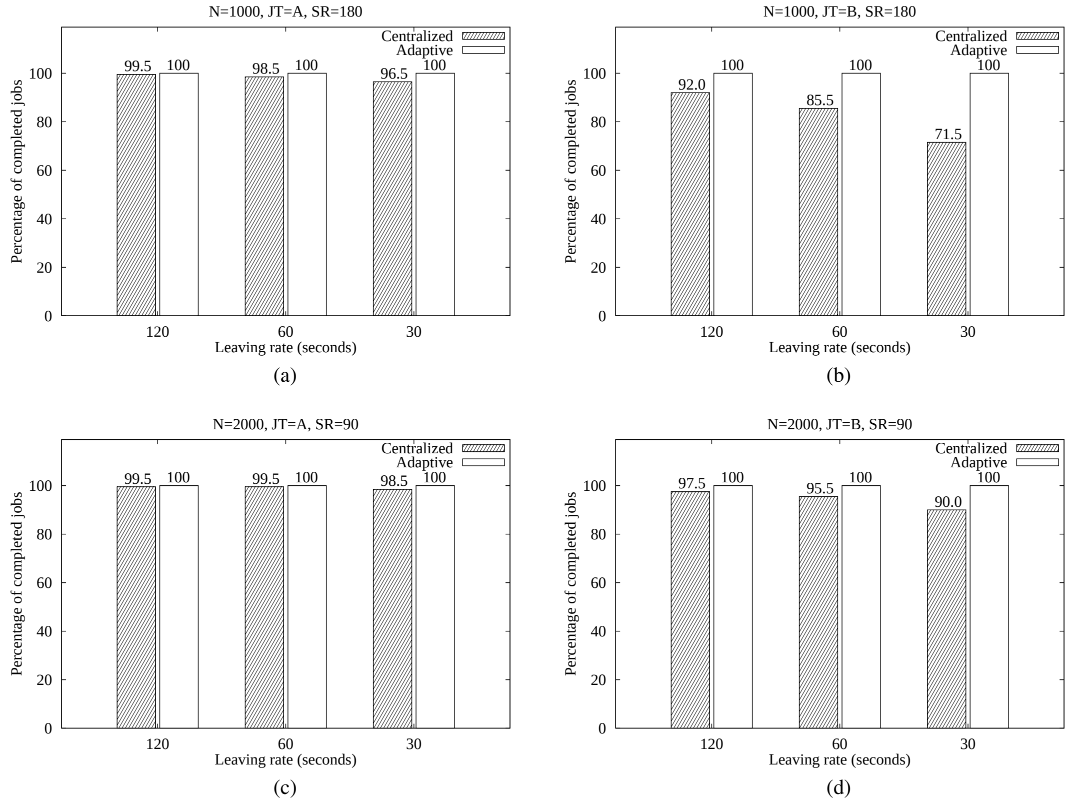

Figure 6. Percentage of completed jobs in four configurations: (a) N=1000, JT=A; (b) N=1000, JT=B; (c) N=2000, JT=A; (d) N=2000, JT=B

The evaluation has been carried out by implementing a simulator of the system in which each node is simulated by an independent thread. Each thread executes the algorithms specified by the state diagram in Figure 4, and communicates with the other threads by invoking local routines having the same interface of the JXTA pipes. Table I shows the main parameters used during the simulation.

Table I
SIMULATION PARAMETERS

Symbol	Description	Values
`N`	Total number of nodes in the network	1000, 2000
`LR`	Leaving rate: avg. amount of time (secs.) between two "node leaving" events	30, 60, 120
`JR`	Joining rate: avg. amount of time (secs.) between two "node joining" events	equal to `LR`
`SR`	Submission rate: avg. amount of time (secs.) between two job submissions	180 (`N`=1000) 90 (`N`=2000)
`JT`	Job type	A, B (see Table II)

As shown in the table, we simulated MapReduce networks of two sizes: `N`=1000 and `N`=2000 nodes (including both slaves and masters). To simulate dynamic node participation, a joining rate `JR` and a leaving rate `LR` have been defined. On average, every `JR` seconds one node joins the network, while every `LR` seconds another node (out of all `N` nodes) abruptly leaves the network so as to simulate an event of failure (or a graceless disconnection). In our simulation `JR`=`LR` in order to keep the total number of nodes and the master/slave ratio approximatively constant during the whole simulation. In particular, we used three values for `JR` and `LR`: 30, 60 and 120, so as to evaluate the system under different churn rates. Every `SR` seconds (average value) a user entity submits one job to the system. The value of such submission rate is 180 seconds for networks with `N`=1000 nodes, while it is reduced to 90 seconds for networks with `N`=2000 nodes in order to keep approximatively the same level of load per node in both scenarios.

Each job submitted to the system is characterized by two parameters: number of slaves and total computing time. In order to simulate jobs characterized by realistic combinations of these parameters, we referred to the statistics presented in Ref. [4] about a large set of MapReduce jobs run at Google during some observation periods. 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 total computing time is 874×268 seconds (65.06 hours). On September 2007, the average job completion time has been 395 seconds using 394 machines, with a total computing time of 43.23 hours. We used these statistics to define two job types (`JT`), A and B, as detailed in Table II.

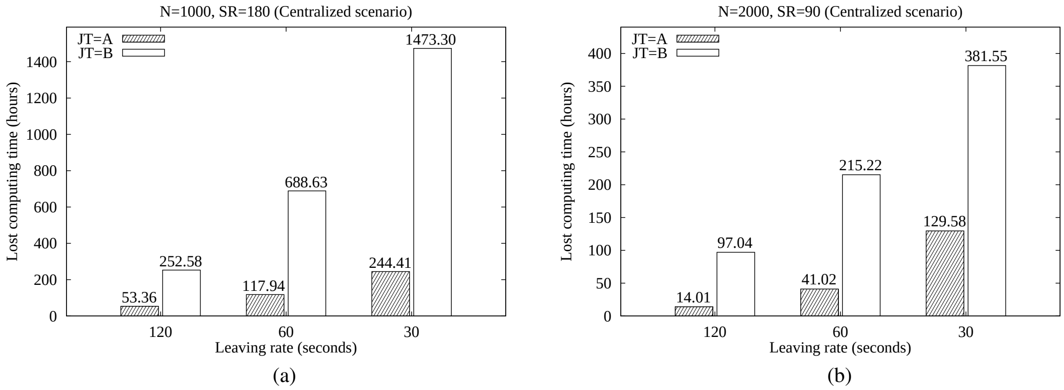

Figure 7. Amount of lost computing time caused by master failures in a centralized scenario for: (a) N=1000; (b) N=2000

Table II
JOB TYPES AND ASSOCIATED PARAMETERS (AVERAGE VALUES)

Type	Number of slaves	Total computing time (hours)
A	394	43.23
B	268	65.06

Hence, the jobs submitted to the system simulator belong either to type A or B. For a given submitted job, the system calculates the amount of time that each slave (assigned to that job) needs to complete its task as the ratio between the total computing time and the number of slaves required by that job. Note that every node, other than managing a job (as a master) or a task (as a slave), executes the network maintenance operations described above, i.e., election of the coordinator, choice of backup masters, and so on.

The main goal of our simulations is evaluating the number of jobs failed versus the total number of jobs submitted to the system, and the corresponding amount of computing time lost as a consequence of these failures. 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 re-assigned to another slave.

This evaluation is performed when the system is in steady state, that is when *M-net* and *S-net* are formed and the desired value of N has been reached. The system has been evaluated in two scenarios: i) *centralized*, where there is only one primary master and there are not backup masters; ii) *adaptive*, where there are $0.01\times$N masters and each job is managed by one master which dynamically replicates the job state on one backup master.

Figure 6 compares the percentage of completed jobs in the centralized and adaptive scenarios after the execution of 200 jobs, considering four combinations of network sizes and job types: (a) N=1000, JT=A; (b) N=1000, JT=B; (c) N=2000, JT=A; (d) N=2000, JT=B.

As expected, in the centralized scenario the number of completed jobs decreases as the leaving rate increases, for each of the considered configurations. We can observe that, fixed the values of N and LR, the percentage of completed jobs is lower when we submit jobs having JT=B. For example, when N=1000 and LR=60, the percentage of completed jobs passes from 98.5% for JT=A to 85.5% for JT=B. This is motivated by the fact that longer jobs (as jobs of type B are compared to jobs of type A) are statistically more subject to be affected by a failure of the associated master.

In contrast to the centralized scenario, the adaptive 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 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 III-B1.

We also evaluated the impact of job failures in a centralized scenario in terms of lost computing time, defined as the total amount of time spent by slaves working on tasks that were part of failed jobs. Figure 7 reports the lost computing time caused by master failures in a centralized scenario related to the same experiments of Figure 6, for different combinations of network sizes, job types, and leaving rates. 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 N=2000 and LR=60, the amount of lost computing time jobs passes from 41.02 hours for JT=A to 215.22 hours for JT=B.

From the results discussed above, we see that a master failure causes loss of dozens or hundreds CPU hours for a typical MapReduce job. Moreover, when the number of available machines per user is limited (as in a typical P2P systems where resources are shared among thousands of users), a master failure produces also a significant loss of user time, since the job completion time increases as the number of machines decreases.

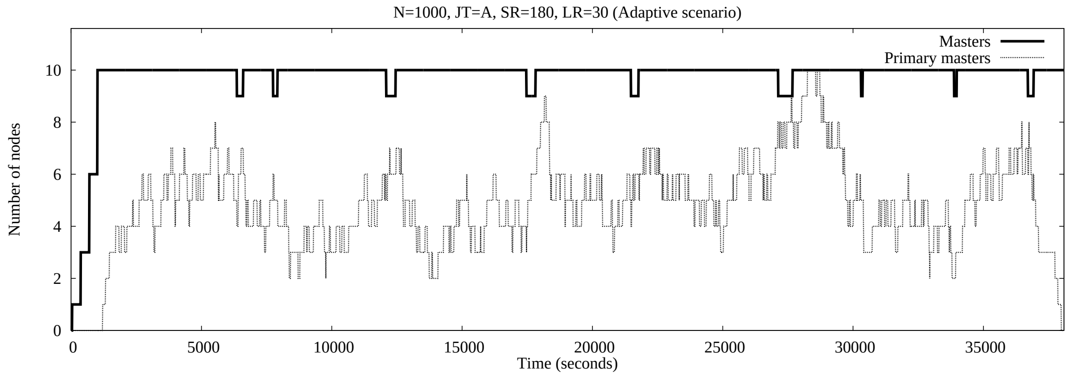

Figure 8. Simulation trace of an adaptive P2P-MapReduce network with N=1000, LR=30, JT=A: Number of masters and primary masters

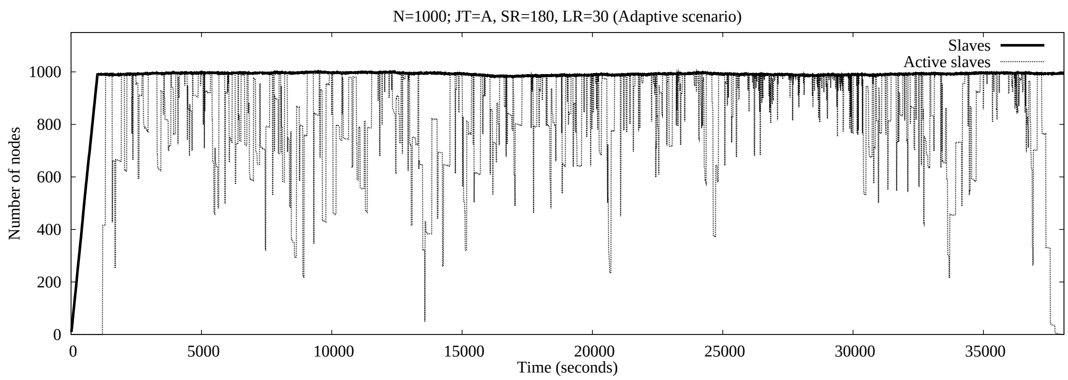

Figure 9. Simulation trace of an adaptive P2P-MapReduce network with N=1000, LR=30, JT=A: Number of slaves and active slaves

After evaluating the capability of the P2P-MapReduce system to manage job submission and recovery under different levels of churn, we evaluated the system behavior when performing the network management tasks described in Section III-B1. In particular, we assessed that the system ensures a desired master/slave ratio and balances the load among the available masters. Figs. 8 and 9 report the number of masters and slaves observed during the simulation of a P2P-MapReduce network with `N`=1000, `LR`=30 and `JT`=A. The simulation trace shows a period of around 38000 seconds, including an initial phase (of 1000 seconds) which is only used to build the network, and a second phase (of around 37000 seconds) in which 200 jobs are executed while nodes join and fail at the desired rate.

Figure 8 shows the number of masters and primary masters observed during the simulation. The system was always able to maintain the desired number of masters, which is set to $0.01 \times$`N`. This corresponds to an average value of 10 masters when the system is in steady state (i.e., after the first 1000 seconds of simulation), while during the initial phase masters increase from 1 to 10 proportionally to the current number of nodes. Indeed, it can be observed that whenever the number of masters decreases as a consequence of a failure, the coordinator replaces it by promoting a slave to that role. Moreover, the figure shows that the number of primary masters ranges around an average of 4.95, which is equal to the measured average number of jobs concurrently running in the system. This result shows that the system balances the master load by assigning each new job to the master with lowest workload. Hence, in the simulated scenario, each master manages either 0 or 1 job at one time. In other simulation scenarios - not discussed here for the sake of space - in which the number of running jobs was greater than the number of masters, the system was always able to evenly distribute the load (i.e., the number of jobs) among the available masters.

Finally, Figure 9 shows the number of slaves and active slaves observed during the same simulation. A slave is considered active when it is currently executing a task. We see that the number of slaves always ranges around an average of 990 according to the fact that `LR` is equal to `JR` (and so the number of nodes remains approximatively the same along the time) and that the system enforces 10 nodes (out of 1000) to act as masters. The number of active slaves varied significantly during the simulation depending on the dynamics of the job submission, hovering around an average of 841.

As a concluding remark, the experimental results discussed throughout this section demonstrate that using an adaptive approach it is possible to extend the MapReduce architectural model making it suitable for highly-dynamic distributed environments where nodes participate intermittently to the system, causing failures that must be effectively managed to avoid a critical loss of computing resources and user time.

V. Conclusion

Providing effective mechanisms to manage node churn and failures, job recovery and intermittent node participation is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in Internet-based computing environments where current MapReduce implementations could be unreliable.

The P2P-MapReduce model presented in this paper exploits an adaptive model to perform job state replication, manage master failures and allow intermittent node participation in a decentralized but effective way. Using a P2P approach, we extended the MapReduce architectural model making it suitable for highly dynamic large-scale environments where failures must be managed to prevent a critical waste of computing resources and time.

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11), 56-61, 2002.

[2] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, V. S. Pande. "Folding@home: Lessons from eight years of volunteer distributed computing". 8th IEEE Int. Workshop on High Performance Computational Biology (HiCOMB'09), Rome, Italy, 2009.

[3] D. P. Anderson. "Boinc: A system for public-resource computing and storage". 5th IEEE/ACM Int. Workshop on Grid Computing (Grid'04), Washington, USA, 2004.

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

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

[6] Hadoop. http://hadoop.apache.org (site visited November 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] Gridgain. http://www.gridgain.com (site visited November 2010).

[10] Skynet. http://skynet.rubyforge.org (site visited November 2010).

[11] MapSharp. http://mapsharp.codeplex.com (site visited November 2010).

[12] Disco. http://discoproject.org (site visited November 2010).

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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