Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model

Fabrizio Marozzo, Domenico Talia, Paolo Trunfio

DEIS, University of Calabria, Via P. Bucci 41C, 87036 Rende, Italy
fmarozzo@unical.it, {talia,trunfio}@deis.unical.it

Extended Abstract

1 Introduction

MapReduce is a programming model used for processing large data sets in a highly-parallel way [1]. Users specify the computation in terms of a "map" function that processes a key/value pair to generate a set of intermediate key/value pairs, and a "reduce" function that merges all intermediate values associated with the same intermediate key.

MapReduce implementations (e.g., Google's MapReduce [2] and Apache Hadoop [3]) are based on a master-slave model. A job is submitted by a user node to a master node that selects idle workers and assigns each one a map or a reduce task. When all map and reduce tasks have been completed, the master node returns the result to the user node. The failure of a worker is managed by re-executing its task on another worker, while master failures are not managed by current MapReduce implementations as designers consider failures unlikely in large clusters or in reliable Cloud environments.

On the contrary, node failures (including master failures) can occur in large clusters and Clouds and are likely to happen in dynamic environments, like computational Grids and volunteer computing systems, where nodes join and leave the network at an unpredictable rate. Therefore, providing effective mechanisms to manage master failures is fundamental to exploit the MapReduce model in the implementation of data-intensive applications in those dynamic environments where current MapReduce implementations could be unreliable. The goal of this work is investigating how to improve the master-slave architecture of current MapReduce implementations to make it more suitable for Grid-like and P2P dynamic scenarios. The extended model we introduce here exploits a P2P model to dynamically assign the master role and to manage master failures in a decentralized but simple way.

In our P2P-MapReduce architecture, each node can act either as master or slave. The role assigned to a given node depends on the current characteristics of that node, and so it can change dynamically over time. Thus, at each time, a limited set of nodes is assigned the master role, while the others are assigned the slave role. Moreover, each master node can act as backup node for other master nodes. A user node can submit the job to one of the master nodes, which will manage it as usual in MapReduce. That master will periodically checkpoint the status of the job on its backup nodes. In case those backup nodes detect the failure of the master, they will elect a new master among them and will restart the job from the latest available checkpoint.

In the following we describe the designed P2P-MapReduce architecture and outline its implementation using the Sun's JXTA P2P framework.

2 Architecture

The P2P-MapReduce architecture includes three basic roles, shown in Fig. 1: user (U), master (M) and slave (S). Master nodes and slave nodes form two logical P2P networks called *M-net* and *S-net*, respectively. As mentioned above, computing nodes are dynamically assigned the master or slave role, hence *M-net* and *S-Net* change their composition over time. The mechanisms used for maintaining this infrastructure are discussed in Section 3.

In the following we describe, through an example, how a master failure is handled in the P2P-MapReduce architecture. We assume the starting situation represented in Fig. 1, where U is the user node that submits a MapReduce job, nodes M are the masters and nodes S are the slaves.

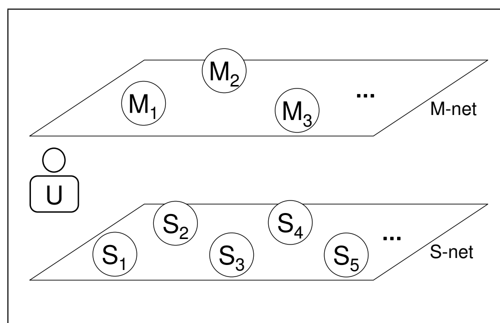

Fig. 1. P2P-MapReduce architecture

The following steps are performed to submit the job and recover from a master failure (see Fig. 2):

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, takes the first element as primary master and the next k elements as backup masters. In this example, M_1 is chosen as primary master and M_2 and M_3 as backup masters ($k = 2$). Then, U submits the MapReduce job to M_1 along with the names of its backup nodes (M_2 and M_3).

2) M_1 notifies M_2 and M_3 that they will act as backup nodes for the current job (in Fig. 2, the apex "B" to node M_2 and M_3 indicates the backup function). This implies that M_1 will periodically backup the entire job state (e.g., the assignments of tasks to nodes, the locations of intermediate results, etc.) 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 slave, choosing (part of) them to execute a map or a reduce task. As for the masters, the choice of the slave

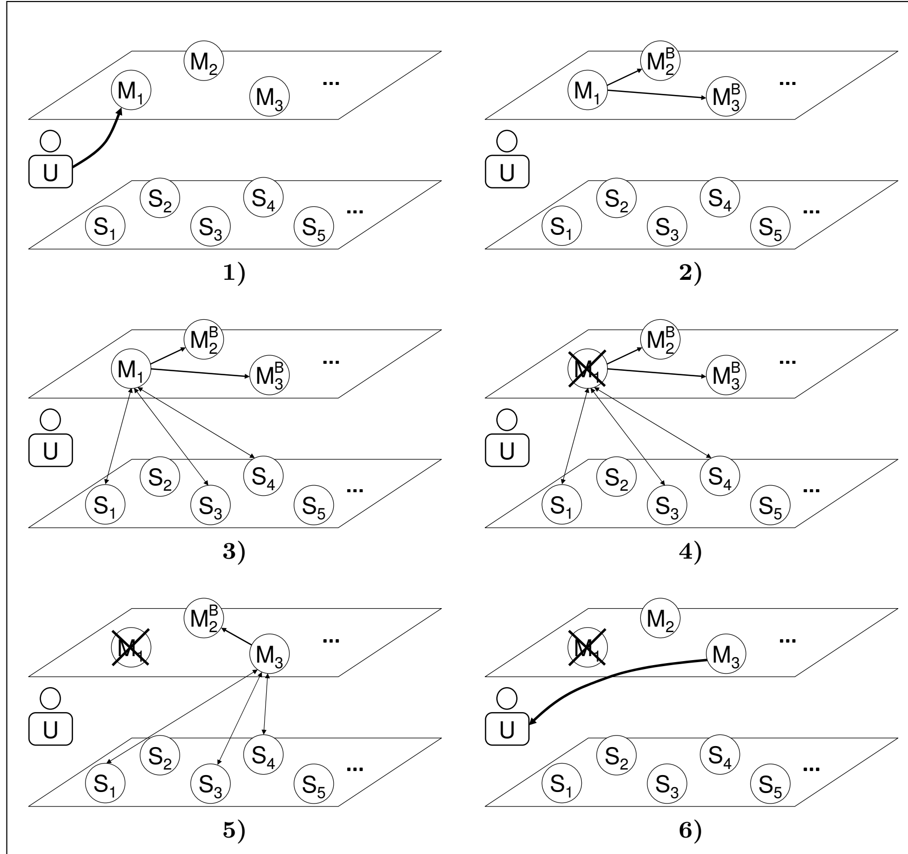

Fig. 2. Steps performed to submit a job and manage a master failure

nodes to use can be done on the basis of a workload index and other performance metrics (e.g. CPU speed). In this example, nodes S_1, S_3 and S_4 are selected as slave nodes. 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 among them the new primary master.

5) The new primary master (M_2) is elected by choosing the backup node with the lowest workload index. The remaining $k-1$ backup nodes (only M_3, in this example) continue to play the backup master role. Then, the MapReduce job restarts from the latest checkpoint available on M_2.

6) As soon as the MapReduce job is completed M_2 returns the result to U.

It is worth noticing that the master failure and the 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. Fig. 3 shows an example of such a situation, in which:

– user nodes U_1 and U_2 have submitted their MapReduce jobs (respectively *Job 1* and *Job 2*);
– M_1 is the primary master of *Job 1* and its backup masters are M_2 and M_3;
– M_3 is the primary master of *Job 2* and its backup master is M_4 (hence, M_3 is at the same time backup master for *Job 1* and primary master for *Job 2*).

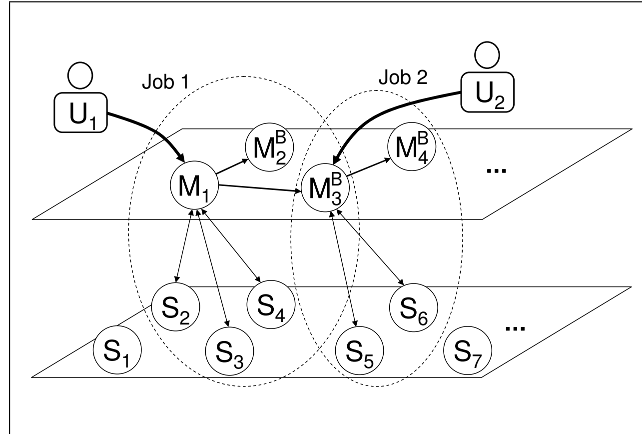

Fig. 3. Master nodes playing multiple roles for different jobs

3 Implementation

After designing the model, we are now implementing the P2P-MapReduce system using the JXTA framework [4]. JXTA provides a set of XML-based protocols that allow computers and other devices to communicate and collaborate in a P2P fashion. In JXTA there are two main types of peers: *rendezvous* and *edge*. The rendezvous peers act as routers in a network, forwarding the discovery requests submitted by edge peers to locate the resources of interest. Peers sharing a common set of interests are organized into a *peer group*. To send messages to each other, JXTA peers use asynchronous communication mechanisms called *pipes*. All resources (peers, services, etc.) are described by *advertisements* that are published within the peer group for resource discovery purposes.

In the following we briefly describe how the JXTA components are used in the P2P-MapReduce system to implement resource discovery, network maintenance, job submission and failure recovery mechanisms.

3.1 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", "WorkloadIndex", and "CPUSpeed".

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

Note that *M-Net* and *S-Net*, represented in Fig. 1, 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.

3.2 Network Maintenance

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

The maintenance task of master nodes is to ensure the existence of a given percentage p of masters on the total number of nodes. This task is performed periodically by one master only (referred to as *coordinator*), which is elected for this purpose among all masters on a turn basis. 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.

3.3 Job Submission and Failure Recovery

To describe the JXTA mechanisms used for job submission and master failure recovery, we take the six-point example presented in Section 2 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 and the backup masters. Then, it opens a bidirectional pipe (called *PrimaryPipe*) to the primary master and submits the job along with the identifiers of the backup nodes.

2) The primary master opens a multicast pipe (*BackupPipe*) to the backup masters. The *BackupPipe* has two goals: copying checkpoint information to the backup nodes and allowing backup nodes to detect a primary master failure when 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" and "CPUSpeed" attributes. 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*) 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 new primary master. This new primary master binds the pipes previously associated to the old primary master (*PrimaryPipe*, *BackupPipe* and *SlavePipes*), and restarts from the latest available checkpoint.
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.

4 Concluding remarks

The master node in current MapReduce implementations is a single point of failure for jobs submitted to it. This implies that, in case of master failure, the whole job submitted to it must be restarted. This results in the waste of great amounts of resources, as the machine time taken by a single MapReduce job is the sum of the machine times of the various slaves that participated to its execution.

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

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

The P2P-MapReduce model presented in this paper exploits a P2P model to perform job status checkpointing and manage master failures in a decentralized but simple way. Using a P2P approach, we extended the MapReduce architectural model making it suitable for highly dynamic environments where failure must be managed to avoid a critical waste of computing resources and time.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, vol. 51 n. 1 (2008) 107-113.
2. Google's Map Reduce. http://labs.google.com/papers/mapreduce.html (Visited: July 2008).
3. Apache Hadoop. http://hadoop.apache.org (Visited: July 2008).
4. Gong, L.: JXTA: A Network Programming Environment. IEEE Internet Computing, vol. 5 n. 3 (2001) 88-95.