

MAPREDUCE CHALLENGES ON PERVASIVE GRIDS

Luiz Angelo Steffenel, Olivier Flauzac, Andrea Schwertner Charão, Patricia Pitthan Barcelos, Benhur Stein, Guilherme Weigert Cassales, Sergio Nesmachnow, J. Rey, Matías Cogorno, Manuele Kirsch Pinheiro, et al.

▶ **To cite this version:**

Luiz Angelo Steffenel, Olivier Flauzac, Andrea Schwertner Charão, Patricia Pitthan Barcelos, Benhur Stein, et al.. MAPREDUCE CHALLENGES ON PERVASIVE GRIDS. Journal of Computer Science, 2014, 10 (11), pp.2194-2210. 10.3844/jcssp.2014.2194.2210 . hal-01085287

HAL Id: hal-01085287
https://paris1.hal.science/hal-01085287

Submitted on 21 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Journal of Computer Science 10 (11): 2194-2210, 2014
ISSN: 1549-3636
© 2014 L.A. Steffenel *et al*., This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license
doi:10.3844/jcssp.2014.2194.2210 Published Online 10 (11) 2014 (http://www.thescipub.com/jcs.toc

MAPREDUCE CHALLENGES ON PERVASIVE GRIDS

[1]L.A. Steffenel, [1]O. Flauzac, [2]A.S. Charao,
[2]P.P. Barcelos, [2]B. Stein, [2]G. Cassales, [3]S. Nesmachnow,
[3]J. Rey, [3]M. Cogorno, [4]M. Kirsch-Pinheiro and [4]C. Souveyet

[1]CReSTIC-SysCom, Universit de Reims Champagne-Ardenne, Reims, France
[2]Laboratório de Sistemas de Computação, Universidade Federal de Santa Maria, Santa Maria, Brazil
[3]Centro de Cálculo, Universidad de la Republica, Montevideo, Uruguay
[4]Centre de Recherche en Informatique, Universite Paris 1 Pantheon-Sorbonne, Paris, France

Received 2014-04-22; Revised 2014-06-29; Accepted 2014-07-25

ABSTRACT

This study presents the advances on designing and implementing scalable techniques to support the development and execution of MapReduce application in pervasive distributed computing infrastructures, in the context of the PER-MARE project. A pervasive framework for MapReduce applications is very useful in practice, especially in those scientific, enterprises and educational centers which have many unused or underused computing resources, which can be fully exploited to solve relevant problems that demand large computing power, such as scientific computing applications, big data processing, etc. In this study, we propose the study of multiple techniques to support volatility and heterogeneity on MapReduce, by applying two complementary approaches: Improving the Apache Hadoop middleware by including context-awareness and fault-tolerance features; and providing an alternative pervasive grid implementation, fully adapted to dynamic environments. The main design and implementation decisions for both alternatives are described and validated through experiments, demonstrating that our approaches provide high reliability when executing on pervasive environments. The analysis of the experiments also leads to several insights on the requirements and constraints from dynamic and volatile systems, reinforcing the importance of context-aware information and advanced fault-tolerance features to provide efficient and reliable MapReduce services on pervasive grids.

Keywords: MapReduce, Fault-Tolerance, Pervasive Distributed Computing

1. INTRODUCTION

One of the first challenges a user faces when deploying MapReduce is that its most known and popular implementation, AH (2014a), requires a highly structured environment such as a dedicated cluster or a cloud infrastructure to be deployed. Indeed, Hadoop has been designed to be deployed over a dedicated cluster or cloud computing infrastructures such as AEMR (2014). Hadoop relies on a collection of tools (Hadoop Core, HDFS, etc.) developed by different Apache subprojects, which interact through a complicate set of master and slave daemons. As a result, Hadoop installation, although well documented, requires a stable set of computer nodes that shall be known at startup time. The installation procedure also lacks of automatic context adaption, forcing the administrator to manually define the characteristics of each resource, such as the number of cores, their relative speed or the available memory.

Together, these elements prevent a user to quickly launch MapReduce over a set of unused resources (e.g., the enterprise workers' desktops), at least not without a previous effort to prepare and configure the nodes. Indeed, even if Hadoop is now a popular data analysis tool, several companies/organizations do not have a dedicated infrastructure, as sometimes the demand for computing intensive tasks is punctual or executed only

Corresponding Author: L.A. Steffenel, CReSTIC-SysCom, Universit de Reims Champagne-Ardenne, Reims, France

periodically. In these cases, cloud computing infrastructures represent a popular alternative for usaing MapReduce without a dedicate infrastructure. Unfortunately, several enterprises fear (or are forbidden) to distribute sensible data over the cloud. These externalized infrastructures still suffer from security issues that prevent their application in some cases. We believe that, considering the extreme development of mobile devices and networks inside organizations nowadays, the opportunistic use of existing resources as an internal pervasive grid represents an interesting alternative for those who hesitate to keep a dedicated infrastructure or to rent cloud computing resources. Nevertheless, in order to be fully operational, pervasive grids environments have to first tackle problems related to their dynamic nature where nodes join and leave the network dynamically.

Our project is precisely addressing this point: Proposing scalable techniques to support existing Hadoop applications in the context of loosely coupled networks such as pervasive grids. We consider pervasive grids as a large-scale infrastructure with specific characteristics in terms of volatility, reliability, connectivity, security, etc. According to Parashar and Pierson (2010), pervasive grids represent the extreme generalization of the grid concept, in which the resources are pervasive. For these authors, pervasive grids seamlessly integrate pervasive sensing/actuating instruments and devices together with classical high performance systems. In the general case, pervasive grids rely on volatile resources that may appear and disappear from the grid, according their availability. Indeed, mobile devices should be able to come into the environment in a natural way, as their owner moves (Coronato and De Pietro, 2008) and devices from different natures, from the desktop and laptop PCs until the last generation tablets, should be integrated in seamlessly way. It results an environment characterized by three main requirements:

- The volatility of its components, whose participation on the grid is notably a matter of opportunity and availability
- The heterogeneity of these components, whose capabilities may vary on different aspects (platform, OS, memory and storage capacity, network connection, etc.)
- The dynamic management of available resources, since the internal status of these devices may vary during their participation into the grid environment

The following scenario can illustrate these requirements: Let us consider that, during the execution of a job, a mobile device becomes available, integrating the pervasive grid. After its integration, the same device may change its network connection, passing from a fixed connection to a wireless one. The same can be observed with the available memory: After starting a job, device's owner may start new applications that modify device memory status. All these changes have an impact on the job performance and consequently on the availability of this device for the pervasive grid.

Pervasive grids environments have to deal with such additional constraints related to the heterogeneity and the volatility of the resources. In such environments, it is essential to adapt the application to the network variable behavior and to coordinate the resources (task scheduling, data placement, etc.). According to Coronato and De Pietro (2008), pervasive grid environments should be able to self-adapt and self-configure in order to incoming mobile devices. We strongly believe that context-awareness is needed in order to support such self-adaption. Context-awareness can be defined as the ability of a system to adapt its operations to the current context, aiming at increasing usability and effectiveness by taking environmental context into account (Baldauf *et al*., 2007). In order to support environments changes, context-awareness becomes a critical aspect to boost the efficiency of the applications over pervasive grids.

Such dynamic nature of pervasive grids represents an important challenge for executing MapReduce applications over these environments. Context-awareness and nodes volatility become key aspects for successfully executing such applications over pervasive grids.

This work presents the first results of the PER-MARE (2014), whose goal is proposing scalable techniques to support MapReduce in pervasive grids. PER-MARE proposes to fully explore the potential of unused (or underused) resources at enterprises as pervasive grids for MapReduce applications. Our challenge is to adapt MapReduce to these dynamic grids. For this, we focus on the volatility and heterogeneity of the available resources through two complementary approaches: On the one hand, we propose to improve Hadoop with context-awareness and more fault-tolerance concerns; on the other hand, we propose an alternative pervasive grid implementation based on a P2P computing middleware, fully adapted to these dynamic environments. This study demonstrates this vision by its first results, organized in three complementary sections:

(i) A context-aware scheduler for Hadoop; (ii) a fault-tolerant job tracker for Hadoop; and (iii) a pervasive grid implementation of MapReduce.

The remain of this study is structured as follow: Section 2 introduces basic notions of MapReduce and its Hadoop implementations. Section 3 presents related works focusing on context-aware and volatility issues on MapReduce. Section 4 gives an overview of the PER-MARE vision, before presenting our contributions in next sections. Section 5 presents a context-aware scheduler for Hadoop, while on section 6 proposes introducing more fault-tolerance for Hadoop. Section 7 presents a pervasive grid implementation. Section 8 discusses obtained results and remaining opening issues, before concluding.

2. MAPREDUCE/HADOOP BASICS

MapReduce (Dean and Ghemawat, 2008) is a parallel programming paradigm successfully used by large Internet service providers to perform computations on massive amounts of data. This model is currently becoming popular as a solution for rapid implementation of distributed data-intensive applications. The key strength of the MapReduce model is its inherently high degree of potential parallelism that should enable processing of petabytes of data in a couple of hours on large clusters consisting of several thousand nodes.

A MapReduce computation takes a set of input key/value pairs and produces a set of output key/value pairs. The user of the MapReduce paradigm expresses the computation through two functions:

- Map that processes an input key/value pair to generate a set of intermediate key/value pairs
- Reduce that merges all intermediate values associated with the same intermediate key

A few typical examples of simple MapReduce applications include counting URL access frequency by processing Web page requests, creating reverse Web-link graph or an inverted index from large set of documents.

MapReduce most known implementation is AH (2014a). This framework takes care of splitting the input data, scheduling the jobs' component tasks monitoring them and re-executing the failed ones. Currently, two Hadoop versions are available, both organized as two superposed entities: A 'MapReduce' engine and a distributed file system, named HDFS. The engine provides the ability to execute map and reduce tasks across the cluster and reports results, while the distributed file system provides a storage scheme that is able to replicate data across nodes for processing.

On the Hadoop 1.x architecture, these entities are organized in a master-worker pattern (**Fig. 1**), with two different masters (JobTracker and NameNode) and workers (TaskTrackers and DataNodes, respectively).

When a client launches an application on a Hadoop 1.x cluster, the request is initially managed by the JobTracker. The JobTracker collaborates with the NameNode in order to distribute the work tasks as closely as possible to the data on which it will work. Indeed, the NameNode act as the HDFS master, providing metadata services for data distribution and replication. The JobTracker, on its side, coordinates the scheduling of map and reduce tasks into available slots managed by the TaskTrackers. Each TaskTracker executes map and reduce tasks on data from its DataNode, which represents the HDFS slave. When the tasks are complete, the TaskTracker notifies the JobTracker, which identifies when all tasks are complete and eventually notifies the client of job completion.

Due to its centralized architecture, Hadoop 1.x is particularly vulnerable to failures on its master nodes. While failures on the TaskNode and on the NameNode will be supported by Hadoop 1.x, failures on the JobTracker and on the NameNode will compromise job execution. Besides, as illustrated by **Fig. 1**, Hadoop 1.x is designed considering a single client application at time, which leads to an under-exploitation of the resources.

The new Hadoop 2.x version overcomes this 'simgle client' issue, by replacing the initial MapReduce engine by a new one, called YARN. YARN opens Hadoop to the possibility of managing multiple applications and computing models and notably to the deployment of non-MapReduce applications. For this, YARN replaces JobTracker and TaskTracker by a new set of entities that are independent of the application.

On the top of YARN daemons (**Fig. 2**), we found the ResourceManager, which is in charge of managing the entire cluster and of assigning applications to the underlying compute resources. These resources are controlled by the NodeManagers. When a new job is submitted, the ResourceManager delegates the job supervision to an ApplicationMaster, which executes the tasks in abstract Containers controlled by the NodeManagers. Thus, multiple client applications may execute concurrently, sharing cluster resources.

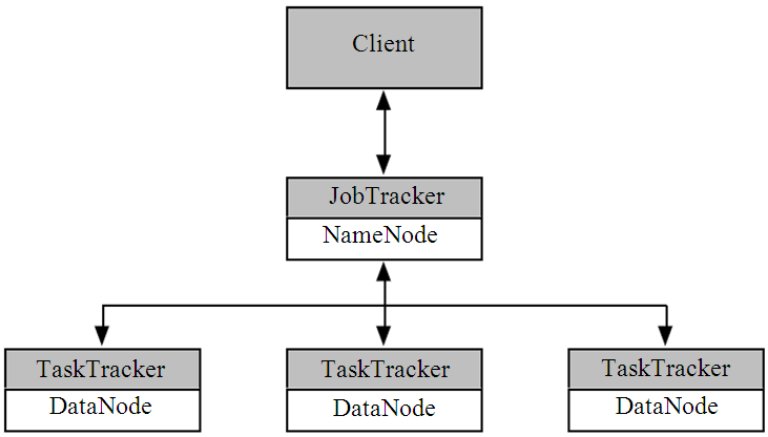


Fig. 1. Hadoop 1.x daemons architecture

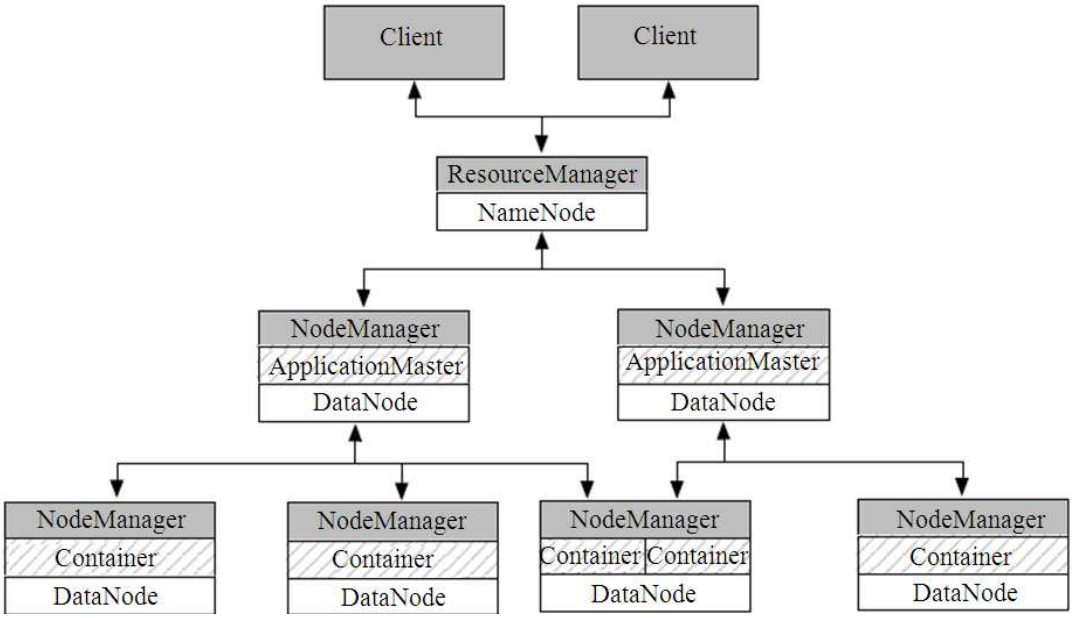


Fig. 2. YARN (Hadoop 2.x) daemon architecture

Unfortunately, Hadoop 2.x still adopts a master/worker architecture, making it as vulnerable as Hadoop 1.x to master (ResourceManager and NameNode) failures. This limitation may considerably affect its performance on pervasive grids, since these nodes may leave the grid during the job execution. Actually, neither Hadoop 1.x nor Hadoop 2.x are able to manage nodes volatility required by pervasive grids. Nodes heterogeneity is not observed either. Both Hadoop 1.x and Hadoop 2.x consider nodes with similar characteristics when managing job execution.

3. RELATED WORKS

This section reviews the main related works on context-awareness on Hadoop and MapReduce implementations on pervasive grids.

3.1. Context-Awareness

Because Hadoop performance is tightly dependent on the computing environment but also on the application characteristics, several researchers focused on bringing context-awareness to Hadoop and can be roughly

classified in three categories: Job schedulers, task schedule optimizers and resource placement facilitators.

In the first category, we find works like Kumar *et al*. (2012), Tian *et al*. (2009) or Rasooli and Down (2012). These works assume that most jobs are periodic and demand similar CPU, network and disk usage characteristics. As a consequence, these works propose classification mechanisms that first analyze both jobs and nodes with respect to its CPU or I/O potential, allowing an optimized matching of applications and resources when a job is submitted. Similarly, Isard *et al*. (2009) propose a generic solution that uses the distribution of resources as context information maps in a capacity-demand graph, calculating the optimum scheduling from a global cost function and with the objective of improving general cluster performance.

While the previous works focus on the improvement of the overall cluster performance through an offline knowledge about the applications and the resources, sometimes this is not enough to ensure a smooth operation. For instance, works like Zaharia *et al*. (2008) and Chen *et al*. (2010) focus on improving tasks deployment inside a job as a way to reduce the response time in large clusters, executing many jobs of short duration. Using the environment context information and the job's estimated time to end, these works rely on heuristics to make the connection between elapsed time and a score that represents how much of the job has already been processed. This information is used to generate a threshold, which will determine when a task is slow enough to start a new speculative copy on another possibly faster machine. Chen *et al*. (2010) also uses historical execution traces to improve its predictions.

Finally, works like Xie *et al*. (2010), aims to provide better performance on jobs through better data placement, using mainly the data locality as decision-making information. The performance gain is achieved by the data re-balancing in nodes, leaving faster nodes with more data. This lowers the cost of speculative tasks and also of data transfers through the network.

From the analysis of these works, we observe that most of them rely on the categorization of jobs and nodes, which is hard in a dynamic environment like a pervasive grid. Even when runtime parameters such as elapsed time or data placement are considered, they assume a controlled and well-known environment. Because these assumptions are too restrictive, these works fail on responding to the requirements of a pervasive grid environment.

3.2. Fault-Tolerance and Volatility

In pervasive grids, nodes have to face both the dynamicity of the resources availability but also the own nodes volatility, where any node can fail/disconnect during the application execution.

To respond to the volatility of nodes and improve fault-tolerance of the execution environment, two main approaches has been considered: (i) To harness Hadoop so that it will be able to support a wider range of failure scenarios; or (ii) to implement the MapReduce paradigm on the top of another middleware support. Hadoop includes basic fault-tolerance techniques, data replication and speculative execution of tasks, to minimize the impact of workers failures, but these techniques are not enough to ensure the operation when running on a true dynamic environment like pervasive grids.

When dealing with Hadoop, one of the elementary problems comes from its master/worker architecture. Hadoop was conceived to tolerate worker's failures through node supervision (heartbeats) and speculative task execution, but a failure at the master level forces the reboot of the entire network. While it is possible to improve fault-tolerance at the file system level by using NameNode replication at the HDFS level or integrating another file system like AC (2014), we observe that, at the job/task level, there is a remaining single point of failure, the JobTracker node. To our knowledge, only the commercial solution MapR (2014) provides fault-tolerance at the JobTracker level, but the details about these solutions are not freely available.

Most of the initiatives to improve MapReduce fault-tolerance prefer to rely on other middleware environments. Indeed, the wide acceptance of Hadoop somehow hides the fact that MapReduce can be implemented on the top of other computing middleware systems. Due to the simplicity of its processing model (map and reduce phases), data processing can be easily adapted to a given distributed middleware, which can coordinate tasks through different techniques (distributed task schedulers, work-stealing/bag of tasks, etc.).

Lin *et al*. (2010) propose a system called MapReduce On Opportunistic eNvironment (MOON), which extends Hadoop in order to deal with the high unavailability of resources. MOON relies on a hybrid architecture, where a small set of dedicated nodes are used to provide resources with high reliability, in contrast to volatile nodes, which may become inaccessible during computations. One inconvenient of this system is that in spite of its improved fault-tolerance, nodes must be known in advance, i.e., no new node can join the network.

Also, Tang *et al*. (2010) propose a system designed to support MapReduce applications, by exploiting the BitDew middleware (Fedak *et al*., 2008), which is a programmable environment for automatic and transparent data management on desktop grids (a particular case of pervasive grids aiming at the leverage of unused processing cycles and storage space available within the enterprise). Unfortunately, this study also requires the presence of a stable master node that runs BitDew services and coordinates data/task distribution.

In P2P-MapReduce (Marozzo *et al*., 2010; 2012), the authors present a distributed architecture implemented in JXTA and following the super-peer approach, where the super-peers serve as cache data server, handle jobs submissions and coordinate execution of parallel computation. P2P-MapReduce supports node disconnection and the integration of new nodes, but is partially dependent on Hadoop as it relies on the Hadoop execution engine to execute the applications. Because this system does not integrate a file system coupled with the task manager, the transmission of data is made together with the tasks to compute. While this study has similar objectives to our own, its architecture is complex and presents several dependencies, which may prevent an easy deployment over pervasive grids. We also believe that performance can be drastically improved by implementing a lightweight execution stack independent of Hadoop and by improving task scheduling through the use of both context and data placement information.

Even if works cited above have improved fault-tolerance for MapReduce applications, they still present some limitations when considering pervasive grids and notably the need for small set of stable nodes and a complex architecture (which make difficult their application to heterogeneous resources). The deployment of MapReduce over pervasive grids remains then an open question, since there is no single solution, for the moment, that solves all previously mentioned issues together. We believe that, when considering pervasive grids, where heterogeneity is a major characteristic, data processing/scheduling must be driven by contextual information (resources characteristics, node reliability, network performance, data location) in order to achieve the expected processing performance.

4. THE PER-MARE VISION

Given the problems presented above, we propose to address the lack of context adaptation of MapReduce applications over pervasive grids all while keeping the compatibility with MapReduce most popular implementation, Hadoop. To meet this global goal, we started the PER-MARE project, which objective is to study the adaptive deployment of MapReduce-based applications over pervasive and desktop grid infrastructures.

Our approach is to improve the behavior of MapReduce applications on pervasive grids using a two-fold investigation method. Hence, to better understand the elements that may impact the deployment of MapReduce over pervasive grids, our teams investigate the problem through two different approaches: On the one hand, we try to modify Hadoop in order to implement on it a context-aware scheduling and to improve fault-tolerance, both with the objective to enhance Hadoop against heterogeneity, dynamicity and volatility of the nodes. On the other hand, the second approach relies on the porting of the MapReduce paradigm (and the Hadoop API) over a P2P distributed computing middleware. Because this platform is already adapted to dynamic and volatile networks, it may represent a good alternative for applications implemented with Hodoop's API as shown in **Fig. 3**.

We believe that this double approach is essential to understand and cover all the facets of the pervasive grid challenges. By comparing these two approaches "side by side" we can propose effective solutions and provide important insights on the adaptability to the heterogeneity of resources and the dynamic nature of the networks. Thus, our vision, illustrated on Erreur ! Source du renvoi introuvable., considers the Hadoop API as common access point for MapReduce applications that will be able to fully exploit resources on pervasive grids through two different implementations, based on a context-aware improved Hadoop implementation, or on a pervasive grid solution.

Next sections describe the first results of our work. Firstly, we improved current Hadoop implementation with a context-aware scheduler, allowing Hadoop to observe real characteristics of the available nodes, instead of static configuration of such nodes. This represents a first step towards a better support of heterogeneous environments. Secondly, we propose a fault-tolerant implementation of Hadoop that is able to replicate its master node in order to prevent crashes due to nodes failures, a necessary step to fully handling nodes volatility on Hadoop. While both solutions extend Hadoop, the third contribution presented in this study considers a completely different implementation, based on a P2P middleware. This latter focuses on the volatility of nodes, allowing new nodes to join or to leave a pervasive grid without stopping job execution.

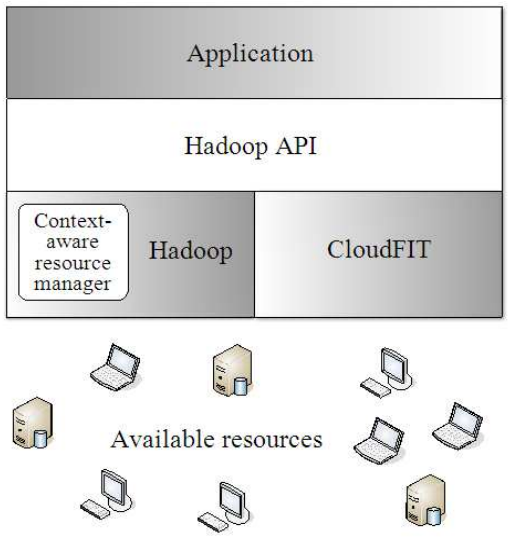


Fig. 3. Overview of PER-MARE vision

5. CONTEXT-AWARENESS ON HADOOP

As dedicated and cloud computing infrastructures have leveraged the use of MapReduce, it is natural that the most known MapReduce distribution has been tailored to such environments. For instance, most IaaS clouds use sets of virtual machines that share similar characteristics, such as computational power and memory. In such cases, MapReduce does not require specific adaptation to the computational context, as all virtual machines are similar. As a consequence, most users simply rely on MapReduce default configurations such as the number of reduce tasks by machine, the maximum memory, etc. Although this behavior can be modified through property files, there is no mechanism to automatically detect and modify these parameters. When dealing with a heterogeneous environment such as a pervasive grid, MapReduce must be able to automatically tune to the nodes characteristics.

While the computational context is tightly related to the processing power of the resources, it also impacts other aspects such as fault-tolerance and data storage. Indeed, Hadoop allows a certain number of duplicated processes/data in order to circumvent fault situations. If the context on pervasive grid is not considered, tasks may be inefficiently allocated or even disappear if the node volatility is high.

Similarly, HDFS tries to place data for the map and reduce phases as closes as possible to the processes/tasks that will need it, in order to reduce the (slow) access over the network. In a pervasive grid, the placement policy must account also on the volatility and speed of the resources, preventing data losses. While the contextual information required for the adaptation of MapReduce can be obtained from the system properties (CPU and network speed, number of cores, memory size, etc.), the diffusion and analysis of such information must be tightly integrated into the MapReduce framework to boost the platform efficiency. For this reason, context-awareness (Preuveneers *et al*., 2009) and context distribution (Kirsch-Pinheiro *et al*. 2008) are important elements to be considered.

Currently, the Apache Hadoop framework scheduler is mostly designed for homogeneous environments in which nodes characteristics are provided at startup in a static way. This section focuses on improving Hadoop scheduler mechanism in order to make it more context-aware towards resources on the cluster.

5.1. Hadoop Schedulers

Hadoop scheduling has evolved along its versions. On Hadoop 1.x, the default scheduler was designed for supporting a batch job submission, organizing jobs in a queue. Similarly, Fair Scheduler (AH, 2013), also proposed on Hadoop 1.x, considers with the same input data size, using a two level scheduling in order to distribute the resources equally: A superior level that allocates queues for each user, using a weighted fair algorithm; and a second level that allocates the resources inside each user queue.

Hadoop 2.x adopts, as a default scheduler, a more sophisticated scheduler, the Capacity Scheduler (AH, 2014b). This scheduler considers a shared Hadoop environment across multiple partners. It focuses on guarantees that a minimum share will always be available for each partner. The benefit comes from the fact that different organizations have processing peaks at different times, therefore the organization (partner) using more capacity, will use the idle capacity of the other organizations. This scheduler tracks the resources registered within the ResourceManager and monitors which resources are free and which are being used.

While these schedulers include some basic awareness about the nodes capacity, we observe that this information is often ignored due to a poor startup configuration. Indeed, Hadoop is heavily dependent on XML files provided at startup and ideally every node should provide its own XML files with tailored parameters to express the node capacity. In a large

heterogeneous cluster, modifying each node configuration can be very time consuming since each node will have a different configuration. Also, the parameters from a XML file are static and any evolution of nodes capacity will not be considered till node reboot.

It is then clear that we must improve the way Hadoop detects and handles context information from the execution environment, by providing updated information about the node capacity.

In order to detect the node capacity, we chose to integrate a context collector into Hadoop, allowing an automatic detection of each node capacity. This information is given to Hadoop scheduler, which can scale the allocation limits in function of the real cluster resource availability. In a first moment, this scaling affects the containers allocation as a function of the available memory and computing cores, impacting therefore on the choice of tasks placement and how speculative task are started. Also, by adapting the capacity to the cluster real resource, no resource would be wasted or left inactive while the scheduler is making tasks wait due to wrong information being received.

5.2. Collecting Context Information

To include context information on Hadoop, we integrated a collector module based on standard Java monitoring API (Oracle, 2014), which allows to easily access the real characteristics of a node, with no additional libraries required.

The collector module, illustrated by **Fig. 4**, allows collecting different context information, such as the number of processors (cores) and the system memory, using a set of interface and abstract/concrete classes that generalize the collecting process. Due to its design, it is easy to integrate new collectors and improve the resources available for the scheduling process, providing data about the CPU load or disk usage, for example.

Context information is described by using a predefined name and a description. Such name corresponds to a given concept identified in the context ontology. This model, inspired from Kirsch-Pinheiro *et al.* (2004), considers context information as an element (a context element), for which multiple values can be observed. Context ontology allows them to semantically describe such element, while the description gives a human readable definition for it.

This collector module was integrated to the NodeManager daemon, since this entity is in charge of processing tasks and managing node definition. In this first prototype, we only collect node capacity (available memory and number of cores) and this information is then sent to the ResourceManager. As a consequence, the information from the context collector module allowed us to improve the Hadoop scheduler operation without having to modify its implementation. This is especially interesting as further works will be able to compare other schedulers from the literature without having to modify their implementation.

5.3. Experiments Description

In order to evaluate our proposal, we compare the container allocation pattern in the original Capacity Scheduler with our context-aware Capacity Scheduler. We perform the analysis of the container allocation pattern when executing the TeraSort algorithm with a 5GB dataset to sort. By requesting enough containers (from nodes expressed capacities) and providing enough data to stress the cluster, we aimed at comparing how the context information influences the containers (tasks) allocation and the overall execution time.

The experiments were performed in a cluster subset of the Grid ‘5000 http://www.grid5000.fr computing environment. The subset had five nodes, one master and four slaves, each node having the following configuration: 2 CPUs AMD@1.7GHz, 12 cores/CPU and 47GB RAM. All nodes were running an Ubuntu-x64-1204 standard image, with Sun JDK 1.7. The Hadoop distribution was the 2.2.0 YARN version.

For unmodified Capacity Scheduler, we adopt the default Hadoop configuration, which defines, on yarn-default.XML, the memory and CPU properties with default values of 8192 and 8 respectively. In the case of the context-aware scheduler, the same properties are obtained from the context collector, overwriting the default parameters from the XML configuration files, resulting in the parameters from **Table 1**.

5.4. Results and Interpretation

The following charts are consolidated by resources represented by NodeManagers. As stated before, the containers are allocated to a given NodeManager and tasks are executed inside these containers. Please note that each segment represents the tasks that are currently being executed on the node, so the end of a segment indicates the completion of some of the tasks (which can still be present in the next segment).

Figure 5a portraits the execution of the TeraSort with original Capacity Scheduler. It is easy to notice that some containers had to wait for the completion of others in order to start processing their tasks.

Indeed, Hadoop splits the work in 38 Map tasks (numbered 2-39), which are distributed to the nodes according to the known resource capabilities. When the first tasks are completed, new tasks are provided to the nodes, if any available (as illustrated in **Fig. 5a**, where tasks 32-39 represent the second execution wave).

If no more tasks are available, the nodes may wait for the job completion of be given additional tasks for speculative execution (for example, tasks numbered 40-44). Because speculative tasks depend on the estimated advancement of current tasks, they usually concern the first tasks deployed and do not contribute to accelerate the execution in a homogeneous cluster. Indeed, we observe that the speculative tasks from **Fig. 5a** do not help improving the execution time as these tasks correspond to tasks from the first wave scheduled on nodes stremi-42 and stremi-44.

Figure 5b portraits the execution of the TeraSort algorithm with context-aware Capacity Scheduler. In this case the overall completion time was reduced, due to the fact that all containers could be started right after the arrival of the request, thanks to the higher resource availability. Without the context information, the scheduler uses the default minimum parameters for the nodes capacities, causing a bad execution performance.

After an analysis and comparison of both charts, we also notice that the default scheduler launches speculative containers 41-43 on node stremi-5 and container 44 on node stremi-42, while the context-aware chart has only the standard containers, which are numbered 2-39. This happens because these extra containers are, in reality, speculative tasks launched because other tasks were taking too long to finish.

To better understand the impact of context information on heterogeneous systems, we also performed a simulation of a heterogeneous cluster. Comparing to the previous experiments, the only difference here is that the nodes are purposely given false capacities when registered to the ResourceManager. Using these false values, a heterogeneous cluster will be simulated with the following capacities:

- Stremi-17: 28 GB of memory and 14 cores
- Stremi-22: 32 GB of memory and 18 cores
- Stremi-33: 48 GB of memory and 24 cores
- Stremi-35: 24 GB of memory and 12 cores
- Total Cluster Resources: 132 GB of memory and 68 cores
- Minimum Allocation: 2 GB of memory and 1 core

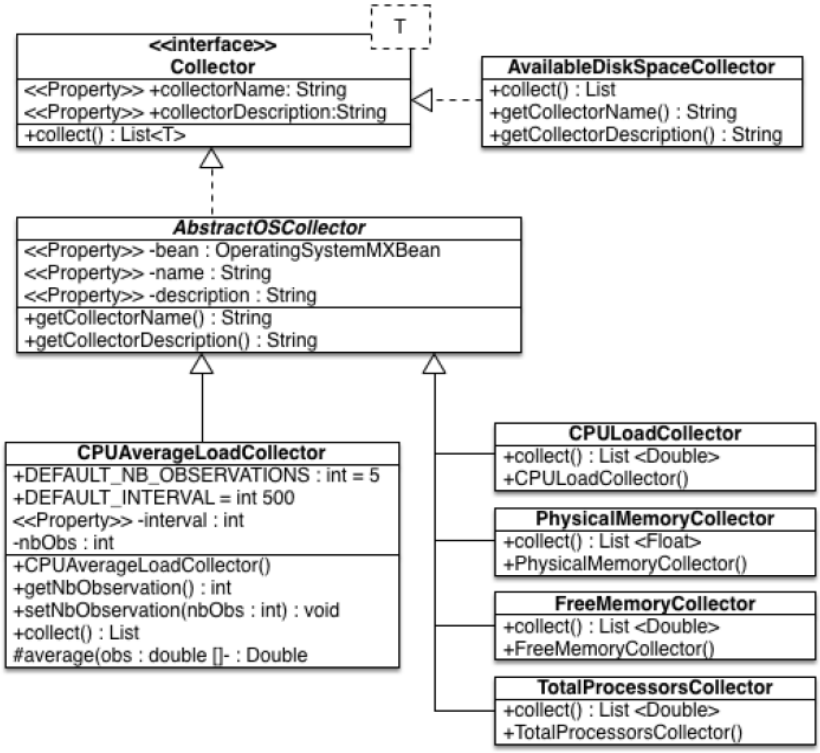


Fig. 4. Elements of the context collector module

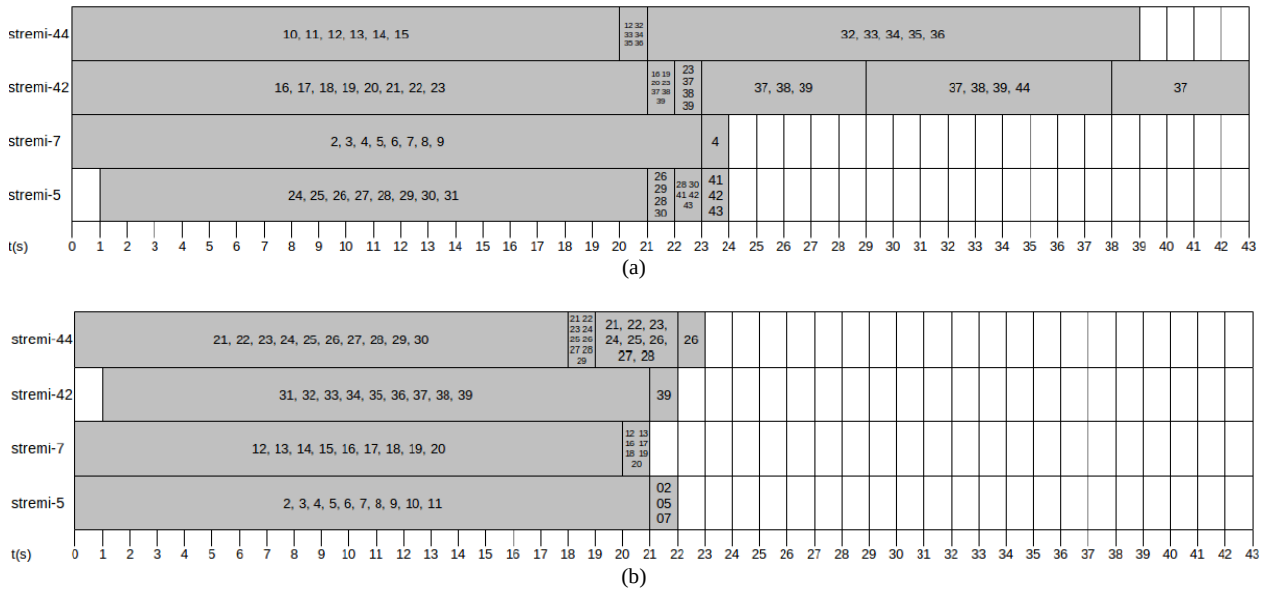


Fig. 5. Container assignment (a) with default configuration and (b) with context-awareness configuration

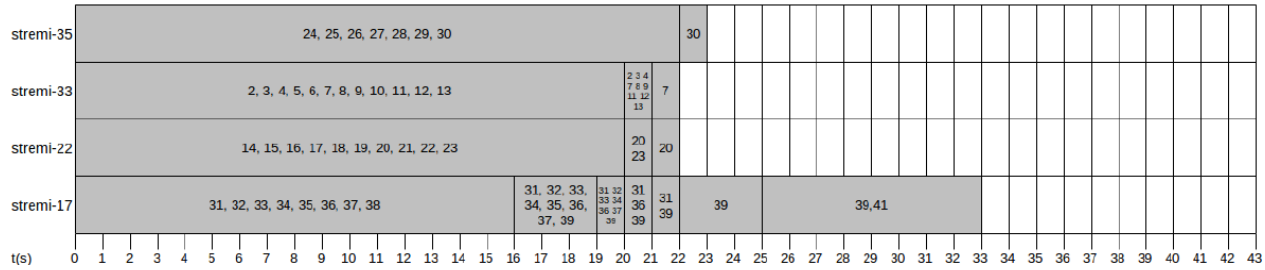


Fig. 6. Container assignment with context-awareness configuration simulating heterogeneous environment

Table 1. Configuration parameters used in the experiments

	Original		Context-aware	
Total cluster resources	32 GB	32 cores	192 GB	96 cores
Min allocation	1 GB	1 core	4 GB	2 cores
Max allocation	8 GB	8 cores	24 GB	12 cores

Figure 6 portraits the execution of the TeraSort algorithm within the simulated heterogeneous environment, also using context-aware Capacity Scheduler. Compared to the default case, the heterogeneous environment execution shows an improvement, but due to the lower cluster capacity, it is a slightly worse than the context-aware Capacity Scheduler executing on a homogeneous environment.

It is possible to note that the containers started the assignment with the node stremi-33, which is the node with the most capacity in the cluster and also was the first to be added in the node list. As in the other experiments, the scheduler launches containers on a node until its resources are all reserved, then move to the next node on the list.

This experiment shows that it is possible to use this context-aware in a heterogeneous environment, the allocations were adapted to a slightly smaller cluster if compared to the real environment. As a future work, it is possible to set the allocation limits in function not only of total cluster resources but also of each individual node resource capacity.

6. A FAULT-TOLERANT HADOOP IMPLEMENTATION

This section introduces our proposal of designing a fault-tolerant implementation of MapReduce. A review of current approaches for fault-tolerance in Hadoop is presented and our proposal for fault-tolerance in Hadoop based in replicating the JobTracker is described.

6.1. Fault-Tolerance on Hadoop

As previously stated, the basic design of Hadoop is still widely directed to cluster and grid computing platforms. In these environments, faults are a second concern as most nodes will operate flawlessly for a long time period. Of course, the failure of a node is still a concern, but some techniques like data replication and speculative execution of tasks may limit the danger on most scenarios.

Because Hadoop was developed to work on a cluster/cloud environment, it has several fault-tolerant mechanisms to circumvent the crash of workers nodes. On Hadoop 1.x this is all orchestrated by the JobTracker, which monitors the status of working nodes while the NameNode coordinates the data replication. HDFS allows the replication of the NameNode (through passive replication), but a failure at the level of the JobTracker forces a job to be restarted.

On Hadoop 2.x, part of the job management responsibility is transferred to the ApplicationMaster, which becomes a task manager. The loss of the ResourceManager does not blocks the execution of a job, only prevents new jobs to be submitted. However, the loss of an ApplicationMaster forces the restart of the job, just like on Hadoop 1.x.

In this section we concentrate on the enhancement of fault-tolerance on Hadoop 1.x (at the JobTracker level), but this solution can also be applied to Hadoop 2.x. We believe that this approach will lead us to improve the reliability of Hadoop, especially in the case of pervasive grids, whose volatility represents a main obstacle to the deployment of Hadoop.

6.2. Fault-Tolerance Through Replication of the JobTracker

As stated in the previous section, we want to develop fault-tolerance solutions that enable Hadoop to operate in pervasive environments, which means that we need to ensure the network would not collapse in the event of a JobTracker failure. The replication of the JobTracker (or the ResourceManager, in Hadoop 2.x) is the key, but several strategies can be applied to replicating, monitoring and resuming the JobTracker. In order to ensure high-availability to the JobTracker on a pervasive system, our solution needs to comply with the following properties:

- Fast recovery in the case of a failure
- Small impact on the performance
- Be able to adapt to the capacity and context of the nodes

The first two properties limit the number of techniques that can be employed. Indeed, a solution that uses an external persistent device would add a non-negligible overhead to the operation and slow-down the recovery. The third property relates to the heterogeneity of the nodes and connections on a pervasive system: Without context-awareness, we risk to resume the JobTracker on a node without the performance or stability levels required for the role.

For all these reasons, we decided to implement JobTracker replication using AZ (2014). ZooKeeper is one of the tools developed initially inside Hadoop that become a full project as its application was extended to other applications. It provides efficient, reliable and fault-tolerant tools for the coordination of distributed systems. In our case, we use ZooKeeper services to storage snapshots of the JobTracker.

Snapshots are made on a per-attribute basis, where JobTracker attributes are stored in ZooKeeper znodes, as illustrated in **Fig. 7**. Depending on the importance of the snapshot, some attributes are replicated synchronously, while other attributed are replicated asynchronously (for example, the blacklist is synced asynchronously while the tasks statuses are synced synchronously). By modifying specific parts of the Hadoop code, we were able to insert snapshot triggers in critical events, minimizing the performance impact of the replication.

In addition, the distributed memory of ZooKeeper is used to coordinate the nodes in the case of a JobTracker failure: ZooKeeper keeps a synchronized ordered list of nodes in the system. This list is regularly updated so that volatile nodes are removed from the list and new nodes are inserted at the end of the list, while the node at the top of the list is current JobTracker node. This procedure naturally organizes the nodes by order of stability, but in the future we plan to modify the nod‘s order to adapt to the context or capacity of the nodes (for example, to avoid giving this role to an old and slow machine).

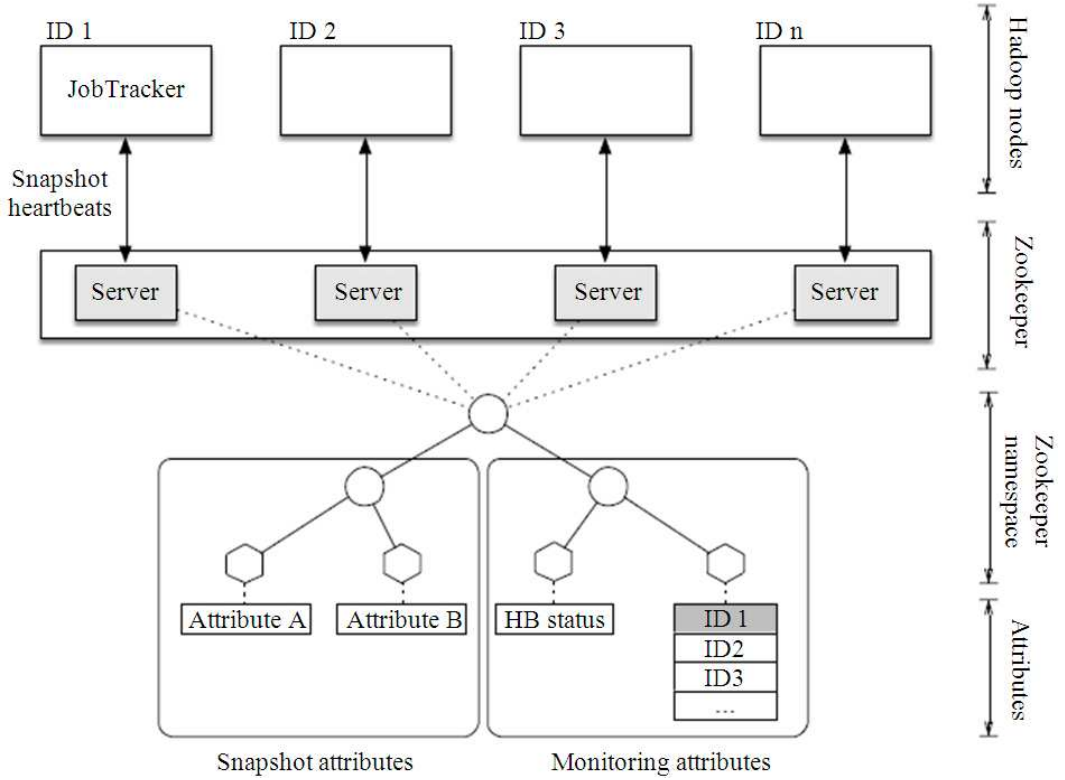


Fig. 7. Replication with zookeeper

The JobTracker is monitored by regular heartbeats. When detecting a connection failure with the JobTracker, all TaskTracker automatically check for ZooKeeper to confirm the status of the JobTracker and/or get the new JobTracker address. Each TaskTracker checks the first node in the ZooKeeper list of nodes; if the TaskTracker is at the top of the list, it replaces the ancient JobTracker spawning a replica using the last snapshot, but if the TaskTracker is not the first node in the list, it tries to connect with this first node that should be the new JobTracker. If the connection fails, the TaskTracker checks the list of nodes again and repeats the process until it connects or becomes the new JobTracker. All these steps are carried using ZooKeeper services.

6.3. Validating the Prototype with Docker-Hadoop

To validate our solutions, however, we need to test different scenarios of node and network faults. Because these experiments require the execution (and reproduction) of well-defined scenarios, we rely on virtual machines (most specifically on container-based virtualization), which allow the researchers to control both as both system images and network interconnections.

In our experiments, we used Docker-Hadoop https://github.com/vierja/docker-hadoop as a testbed to simulate different failure scenarios. Indeed, thanks to Docker-Hadoop dashboard, one can easily switch-off or restart nodes in the environment and reproduce the same scenario at will. Using Docker-Hadoop dashboard allowed us to test different failure scenarios like:

6.3.1. Crash of the JobTracker Node

In this scenario, we kill the JobTracker in order to force a new node to resume the JobTracker role. When a TaskTracker loses connection with the JobTracker, it checks the list of Zookeeper nodes and it tries to connect or becomes the new JobTracker, as described in the previous subsection.

6.3.2. Restart of an Old JobTracker

In this scenario, we investigated the impacts of the return of an old JobTracker node. Two possibilities are analyzed:

- The returning node was simply disconnected from the network and still thinks it is the JobTracker. Zookeeper always keeps a reference to the actual JobTracker, so the JobTracker periodically checks that reference to verify whether it is still the JobTracker or not. If it is not, the returning JobTracker kills himself so the JobTracker referenced by Zookeeper is the only JobTracker in the network
- The returning node has restarted and has lost all its status, but is still on the top of Zookeeper‘s list. In this case, the new node is restarted as a TaskTracker, so it will follows the fault-tolerance mechanism described in the previous subsection to become the JobTracker

6.3.3. Heartbeat Tuning

A too lazy heartbeat slows-down the reaction to failures and may lead to some of the situations described in the previous item. An intensive heartbeat may impact negatively on the overall performance.

While Docker provides an environment to create and destroy nodes, the joining of new nodes requires additional procedures. Indeed, Hadoop was designed to work over a cluster where all the resources are known from the beginning. Inserting new nodes require the restart of the job manager, which may represent an important drawback in a dynamic environment in the next section, we present a P2P approach to solve this problem and discuss the challenges it represents.

7. A P2P IMPLEMENTATION OF MAPREDUCE

Due to its simple task model, MapReduce can be easily implemented in a distributed computing environment. In our project, we rely on the P2P distributed computing middleware CloudFIT (Steffenel, 2013), implemented over the Pastry (Rowstron and Druschel, 2001) overlay network. In CloudFIT, the programmer needs to decide how to divide the problem into a finite number of independent tasks and how to compute each individual task. This is the same principle of MapReduce map and reduce steps, which can be considered as a sequences of Finite number of Independent and Irregular Tasks (Krajecki, 1999) problems.

The CloudFIT framework is structured around collaborative nodes connected over a logical oriented ring overlay network. Task status (and partial results) are broadcasted among the nodes, which contributes to the coordination of the computing tasks and form a global view of the calculus.

A node owns the different parameters of the current computations (a list of tasks and associated results). It is able to locally decide which tasks still need to be computed and can carry the work autonomously if no other node can be contacted. If later a node reintegrates a community, it is able to share the results from the tasks it completed and re-synchronize its task’s list. For the moment, a simple scheduling mechanism randomly rearranges the list of tasks at each node, which helps the computation of tasks in parallel without requiring additional communication between nodes.

From the strict point of view of FIIT, a MapReduce job can be expressed as a two rounds execution: One handling Map tasks and another handling Reduce tasks. By implementing MapReduce over a P2P platform such as CloudFIT, we can introduce interesting properties on MapReduce that are not always available on Hadoop.

Implementing MapReduce over CloudFIT is quite straightforward and can easily mimic the behavior of Hadoop. Hence, during the Map phase, several tasks are launched according the number of input files, producing a set of (k_i, V_i) pairs. The token passing mechanism ensures that all pairs (i.e., the results of each task) are broadcasted to all computing nodes. Therefore, at the end of the Map phase, each node contains a copy of the entire set of (k_i, V_i) pairs.

At the end of the first step, a new CloudFIT job is launched, using as input parameter the results from the map phase. The number of tasks during this Reduce phase is calculated based on the number of available nodes. Once a round starts, each node starts a task from the shared task list and broadcasts its results at the end of the task’s computation.

Using CloudFIT, MapReduce algorithms are supposed to support nodes failures as well as nodes volatility, allowing nodes to dynamically leave and join the grid. Indeed, as long as a task is not completed, other nodes on the grid may pick it up. In this way, when a node fails or leaves the grid, other nodes may recover tasks originally taken by the crashed node. Inversely, when a node joins the CloudFIT community, it receives a copy of the working data and may pick up available (incomplete) tasks on the shared task list. Thus, CloudFIT should offer a more fault-tolerant behavior than Hadoop, supporting not only nodes disconnections, but also nodes (re-)connection.

Because Hadoop relies on specific classes to handle data, we tried to use the same ones in CloudFIT

implementation as a way to keep compatibility with the Hadoop API. However, some of these classes were too dependent on inner elements of Hadoop, forcing us to develop our own equivalents, at least for the moment (further works shall reinforce the compatibility with Hadoop API). For instance, we had to substitute the OutputCollector class with our own MultiMap class, while the rest of the application remains compatible with both Hadoop and CloudFIT.

An initial example of MapReduce over CloudFIT was proposed, using the traditional WordCount application. As indicated previously, this first prototype organizes MapReduce in a two rounds execution, but one single difference between this implementation and the one using Hadoop resides on the need to indicate the number of computing tasks, called blocks. Indeed, this behavior is automatized on Hadoop, which tries to guess the required number of Map and Reduce processes. In our prototype, this parameter was defined as to mimic the behavior of Hadoop, i.e., by setting a number of Map tasks to roughly correspond to the number of input files and the number of Reduce tasks to correspond to the number of computing cores available on the CloudFIT network at the time Reduce starts (this number may varies later, due to nodes volatility).

7.1. Prototype Evaluation

The experiments were conducted over 16 machines on the Helios cluster from the Grid'5000 network. Each machine is composed by 2 AMD Opteron 275 2.2 GHz CPUs, totalizing 4 cores per node and a Gigabit Ethernet interconnects the nodes.

For the experiments, we evaluate the performance of both CloudFIT and Hadoop solutions when varying the total amount of data and the number/size of input files. For each data size, we measure 3 different input splits: One single file, 1MB splits and 512kB splits. The reason for such approach is to analyze the impact of the input files on the map step from both solutions. For the input data, we chose the Gutenberg Project Science Fiction Bookshelf CD http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project, which contains more than 200 books in text format. The results presented on **Fig. 8** represent the median of the performed measures for 16 nodes.

When analyzing the measures, two major scenarios arise: For small data volumes, our prototype largely outperforms Hadoop, while the difference tends to stabilize for large data sets. This is mostly due to CloudFIT lightweight middleware. Even though, the analysis of application and middleware traces shows that the replication pattern used on CloudFIT shall be improved if we want to achieve good performances. Indeed, currently we use a full-replication scheme, so that up to *n-1* nodes can fail without losing the job progress. The inconvenient is that this overloads the network with results transfers and also requires an important storage capacity on each node.

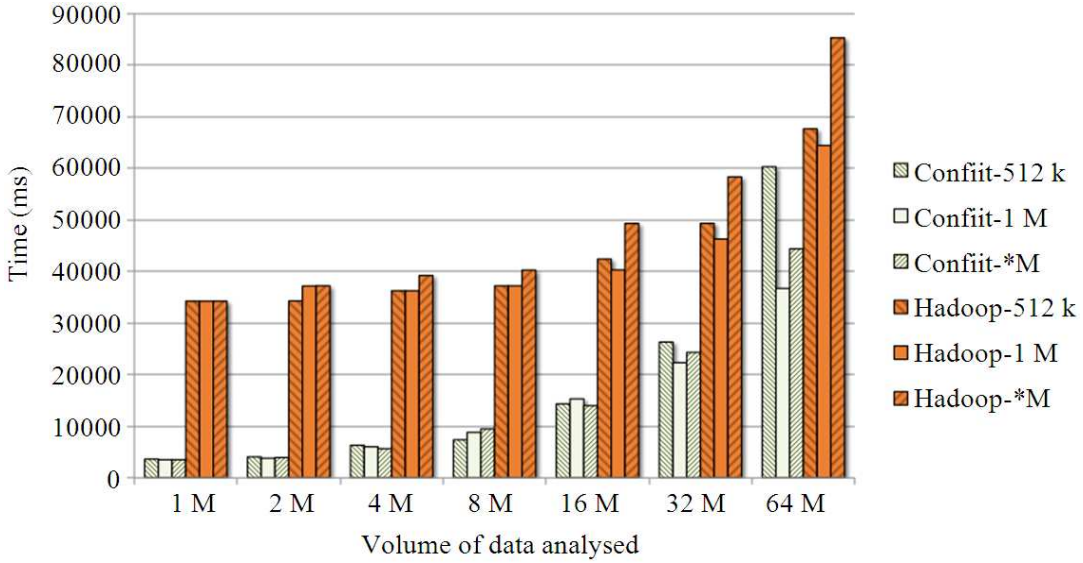


Fig. 8. Hadoop vs CloudFIT performance comparison

To circumvent this bottleneck, we are currently implementing an alternative storage mechanism based on DHTs, like for example the PAST distributed file system (Druschel and Rowstron, 2001). Using PAST, we allow workers to share the results of the tasks with a configurable replication factor, all while minimizing the data transfer between nodes and the storage requirements. PAST integration is still under work but we expect to integrate this solution and our context acquisition module to make tasks scheduling aware of context elements such as data locality, network and processing capabilities of the nodes.

8. CONCLUSION

Pervasive grids represent an important step towards the establishment of mobiquitous systems in which concerns high performance computing. While the pervasive computing model has no intention to supersede classical high performance computing, there is a large domain of applications that require more flexible environments, as provided by a pervasive computing model. Indeed, pervasive grids concentrate three main challenges on dynamic environment composed by a multitude of devices: (i) The volatility of its components; (ii) their intrinsic heterogeneity; and (iii) how to manage the dynamic evolution of these resources.

In this study we study these issues when deploying MapReduce applications on top of pervasive grids. We observed that Hadoop, the most known implementation of MapReduce on clusters and cloud infrastructures, fails to respond to the three challenges listed above. We strongly believe that pervasive grids are especially adapted to deploy MapReduce application on enterprises, fully exploring the potential of unused (or underused) resources and therefore reinforcing the enterprises competitiveness.

To reach this goal, we present the basis of the PER-MARE project, which explores a two-fold approach for implementing effective MapReduce support on pervasive grids: First, by improving Hadoop so that it supports a minimum of volatility and context awareness; second, by developing an alternative middleware for MapReduce directly on top of a pervasive grid platform. By proceeding on both fronts, we aim at obtaining better insights on scientific and technical obstacles towards the development of flexible and adaptive MapReduce middleware.

Therefore, in a first moment we have propose to introduce context information on Hadoop schedulers, in order to take in account the heterogeneity and the dynamicity of the nodes. By injecting real-time context information (such as available memory) on Hadoop schedulers, we circumvent the poor dynamicity management of current Hadoop implementations. As the experiments showed encouraging performance speedups, the next steps will include additional context information (CPU load and data location, for example) and the study of alternative scheduling algorithms more tailored to resource variability.

Later, we focused on Hadoop fault-tolerance, studying how to remove one of the last single point of failure in the architecture and therefore allowing a smooth operation in dynamic environments where any node can disconnect or fail. By coupling context-aware scheduling and an improved fault-tolerant architecture, we are able to support the disconnection of any node in the architecture as well as distributing work tasks according to the nodes capabilities.

Even though the previous contributions improve Hadoop operation, Hadoop remains a complex (and heavy) middleware that not always can be deployed on pervasive systems. Therefore, we also presented our efforts to implement MapReduce over a pervasive grid middleware, as a way to embrace and take profit from the volatility of pervasive grids and devices diversity. By proposing a Hadoop-compliant API over a pervasive grid middleware, we offer MapReduce applications a transparent choice between an implementation optimized for data-intensive problems (Hadoop) and one optimized for computing intensive problems over a highly dynamic environments.

From these contributions we pointed out several elements that can be improved in both Hadoop and pervasive grid frameworks. For instance, future works shall continue towards the association of these contributions in order to provide a complete panorama of MapReduce solutions on pervasive grids.

9. ACKNOWLEDGEMENT

The authors would like to thank their partners in the PER-MARE project and acknowledge the financial support given to this research by the CAPES/MAEE/ANII STIC-AmSud collaboration program (project number 13STIC07). Experiments presented in this study were carried out on Grid'5000 experimental testbed (https://www.grid5000.fr).

10. REFERENCES

AEMR, 2014. Amazon Elastic MapReduce.

AH, 2013. Fair scheduler. Apache Hadoop.

AH, 2014a. Welcome to Apache Hadoop.

AH, 2014b. Hadoop MapReduce next gen-eration-capacity scheduler. Apache Hadoop.

AC, 2014. Welcome to Apache Cassan-dra. Apache Cassandra.

AZ, 2014. ZooKeeper: A distributed coordination service for distributed applications. Apache Zookeeper.

Baldauf, M., S. Dustdar and F. Rosenberg, 2007. A survey on context-aware systems. Int. J. Ad Hoc Ubiquit. Comput., 2: 263-277. DOI: 10.1504/IJAHUC.2007.014070

Chen, Q., D. Zhang, M. Guo, Q. Deng and S. Guo, 2010. SAMR: A Self-adaptive MapReduce scheduling algorithm in heterogeneous environment. Proceedings of the IEEE 10th International Conference on Computer and Information Technology, Jun. 29-Jul. 1, IEEE Xplore Press, Bradford, pp: 2736-2743. DOI: 10.1109/CIT.2010.458

Coronato, A. and G.D. Pietro, 2008. Mipeg: A middle-ware infrastructure for pervasive grids. Future Generat. Comput. Syst., 24: 17-29. DOI: 10.1016/j.future.2007.04.007

Dean J. and S. Ghemawat, 2008. Mapreduce: Simplified data processing on large clusters. Commun. ACM, 51: 107-113. DOI: 10.1145/1327452.1327492

Druschel, P. and A. Rowstron, 2001. PAST: A large-scale, persistent peer-to-peer storage utility. Proceedings of the 8th Workshop on Hot Topics in Operating Systems May 20-22, IEEE Xplore Press, 75-80. DOI: 10.1109/HOTOS.2001.990064

Rowstron A. and P. Druschel, 2001. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Nov. 12-16, Springer Berlin Heidelberg, Germany, pp: 329-350. DOI: 10.1007/3-540-45518-3_18

Fedak, G., H. He and F. Cappello, 2008. BitDew: A programmable environment for large-scale data management and distribution. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 15-21, IEEE Xplore Press, Austin, TX., pp: 1-12. DOI: 10.1109/SC.2008.5213939

Isard, M., V. Prabhakaran, J. Currey, U. Wieder and K. Talwar *et al*., 2009. Quincy: Fair scheduling for distributed computing clusters. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Oct. 11-14, ACM, New York, pp: 261-276. DOI: 10.1145/1629575.1629601

Kirsch-Pinheiro, M., J. Gensel and H. Martin, 2004. Representing Context for an Adaptative Awareness Mechanism. In: Groupware: Design, Implementation and Use, Gabriela Marín Raventós, Luis A. Guerrero and Gert-Jan de Vreede (Eds.)., Springer, ISBN-10: 3540230165, pp: 339-348.

Kirsch-Pinheiro, M., Y. Vanrompay, K. Victor, Y. Berbers and M. Valla *et al*., 2008. Context Grouping Mechanism for Context Distribution in Ubiquitous Environments. In: On the Move to Meaningful Internet Systems, R. Meersman and Z. Tari (Eds.), Springer, pp: 571-588.

Krajecki, M., 1999. An object oriented environment to manage the parallelism of the FIIT applications. Proceedings of the 5th International Conference on Parallel Computing Technologies, Sept. 6-10, Springer Berlin Heidelberg, Russia, pp: 229-234. DOI: 10.1007/3-540-48387-X_25

Kumar, K.A., V.K. Konishetty, K. Voruganti and G.V. Prabhakara Rao, 2012. CASH: Context aware scheduler for Hadoop. Proceedings of the International Conference on Advances in Computing, Communications and Informatics, Aug. 03-05, ACM New York, pp: 52-61. DOI: 10.1145/2345396.2345406

Lin, H., X. Ma, J. Archuleta, W. Feng and M. Gardner *et al*., 2010. Moon: Mapreduce on opportunistic environments. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Jun. 21-25, ACM New York, pp: 95-106. DOI: 10.1145/1851476.1851489

MapR, 2014. MapR closes $110m financing led by Google Capital. MapR Technologies, Inc.

Marozzo, F., D. Talia and P. Trunfio, 2010. A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments. In: Cloud Computing: Principles, Systems and Applications, Antonopoulos, N. and L. Gillam (Eds.)., Springer, ISBN 978-1-84996-240-7, pp: 113-125.

Marozzo, F., D. Talia and P. Trunfio, 2012. P2P-MapReduce: Parallel data processing in dynamic cloud environments. J. Comput. Syst. Sci., 78: 1382-1402. DOI: 10.1016/j.jcss.2011.12.021

Oracle, 2014. Overview of Java SE Monitoring and Management,

Parashar, M. and J.M. Pierson, 2010. Pervasive Grids: Challenges and Opportunities. In: Handbook of Research on Scalable Computing Technologies, Li, K., C. Hsu, L. Yang, J. Dongarra and H. Zima (Eds.), IGI Global Snippet, Hershey, ISBN-10: 1605666629, pp: 14-30.

PER-MARE, 2014, PER-MARE-adaptive deployment of MapReduce-based applications over pervasive and desktop grid infrastructures.

Preuveneers, D., K. Victor, Y. Vanrompay, P. Rigole and M. Kirsch-Pinheiro, 2009. Context-Aware Adaptation in an Ecology of Applications. In: Context-Aware Mobile and Ubiquitous Computing for Enhanced Usability: Adaptive Tech-nologies and Applications, Stojanovic, D. (Ed.), IGI Global, Hershey, ISBN-10: 1605666629, pp: 1-25.

Rasooli, A. and D. Down, 2014. COSHH: A classification and optimization based scheduler for heterogeneous Hadoop systems. Future Generat. Comput. Syst., 36: 1-15, DOI: 10.1016/j.future.2014.01.002

Steffenel, L.A., 2013. Deliverable 2.1: First Steps on the Development of a P2P Middleware for MapReduce. PER-MARE report.

Tang, B., M. Moca, S. Chevalier, H. He and G. Fedak, 2010. Towards MapReduce for desktop grid computing. Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Nov. 4-6, IEEE Xplore Press, Fukuoka, pp: 193-200. DOI: 10.1109/3PGCIC.2010.33

Tian, C., H. Zhou, Y. He and L. Zha, 2009. A dynamic MapReduce scheduler for heterogeneous workloads. Proceedings of the 8th International Conference on Grid and Cooperative Computing, Aug. 27-29, IEEE Xplore Press, Lanzhou, Gansu, pp: 218-224. DOI: 10.1109/GCC.2009.19

Xie, J., S. Yin, X. Ruan, Z. Ding and Y. Tian *et al*., 2010. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Apr. 19-23, IEEE Xplore Press, Atlanta, GA., pp: 1-9. DOI: 10.1109/IPDPSW.2010.5470880

Zaharia, M., A. Konwinski, A.D. Joseph, R. Katz and I. Stoica. 2008. Improving MapReduce performance in heterogeneous environments. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, (SDI' 08), USENIX Association Berkeley, CA, USA., pp: 29-42.