



HAL
open science

First Steps on the Development of a P2P Middleware for Map-Reduce

Luiz Angelo Steffemel

► **To cite this version:**

Luiz Angelo Steffemel. First Steps on the Development of a P2P Middleware for Map-Reduce. 2013.
hal-00858318

HAL Id: hal-00858318

<https://hal.science/hal-00858318>

Submitted on 5 Sep 2013

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.






PER-MARE - Pervasive Map-Reduce
Project number 13STIC07
CAPES/MAEE/ANII STIC-AmSud collaboration program

Deliverable 2.1

First Steps on the Development of a P2P Middleware for Map-Reduce

Luiz Angelo Steffene

Reviewer: Manuele Kirsch Pinheiro

Partner	Author's information
	
	<p><i>Manuele Kirsch Pinheiro</i> Centre de Recherche en Informatique 90 rue Tolbiac, 75013 Paris, France Manuele.Kirsch-Pinheiro@univ-paris1.fr</p>
	<p><i>Luiz Angelo Steffemel</i> CReSTIC – SysCom - Université de Reims Champagne-Ardenne UFR Sciences Exactes et Naturelles Département de Mathématiques et Informatique - BP 1039 51687 REIMS CEDEX 2 Luiz-Angelo.Steffemel@univ-reims.fr</p>
	

Abstract

This document discusses the main requirements of a P2P computing middleware that satisfies the objectives of PER-MARE project. Starting with a comparison between MapReduce and FIIT paradigms, this document makes an inventory of the characteristics of CONFIIT, the P2P computing middleware initially aimed by the project. As CONFIIT does not respond to our needs, we establish the requirements for a new P2P tool that is being developed to replace CONFIIT. The main architectural elements of the new tool (called CloudFIT) are presented, as well as the details concerning its first prototype. Finally, we discuss the preliminary results obtained with this prototype, and set some development directions that shall guide the future research in the coming months.

Table of Contents

Abstract	3
1 Introduction	5
1.1 Project and WP2 Goals	5
1.2 Objectives of this report	5
2 Defining a P2P Computing Middleware for PER-MARE	6
2.1 Computing models: FIIT versus Map-Reduce	6
2.1.1 FIIT	6
2.1.2 Map-Reduce.....	8
2.1.3 Implementing Map-Reduce over FIIT	10
3 Analysis of CONFIIT and its Drawbacks.....	13
3.1 Changelog of CONFIIT	13
3.2 Limitations of CONFIIT when dealing with Map-Reduce problems.....	13
4 Development of CloudFIT, a successor for CONFIIT	15
4.1 Design choices of CloudFIT	15
4.2 Implementing CloudFIT – Architecture.....	16
4.3 Implementing CloudFIT – First prototype.....	17
4.4 Prototype evaluation	21
4.5 How to Improve Data Transfer	23
5 Conclusions.....	25
6 References	26

1 Introduction

1.1 Project and WP2 Goals

The PER-MARE project [14] aims at the adaptation of MapReduce for pervasive grids. The main goal is to propose scalable techniques to support existent MapReduce-based, data-intensive applications in the context of loosely coupled networks such as pervasive and desktop grids.

In order to reach this goal, PER-MARE project proposes a two-fold approach [14]:

- (i) to adapt a well-known MapReduce implementation, Hadoop, including on it context-aware elements that may allow its efficient deployment over a pervasive or desktop grid;
- (ii) to implement a Hadoop-compatible API over a P2P distributed computing environment originally meant for pervasive grids.

This double approach intends to bring us better insights on the deployment of MapReduce over pervasive grids. The first one intends handling problems related to context-awareness and task scheduling, while the second one will deal with data storage and code-compatibility issues. The ambition behind this double approach is to provide a wider vision on the problem, leading to more efficient solutions, by conjointly evaluating these approaches.

The Work Package 2 (WP2) focuses on the study of P2P solutions for the MapReduce paradigm. It is in charge of the development of fully distributed solutions for MapReduce under the form of P2P distributed computing platforms, and its integration within PER-MARE proposals. This work package investigates how to bring MapReduce to pervasive systems in a more flexible way that the current Hadoop framework allows. Indeed, its final objective is to bring Hadoop applications to pervasive systems, through the adaptation of the Hadoop API over a P2P computing middleware.

1.2 Objectives of this report

In this deliverable, we start presenting the works of WP2 through the analysis of an existing P2P middleware, called CONFIIT, and how to adapt it to the project objectives. This document presents the initial works we conducted and introduces some preliminary results we obtained.

Therefore, Section 2 discusses the computational models used on a pervasive computing environment and their limitations. Section 3 analyses an existing middleware (CONFIIT) and discusses how it could be used within the PER-MARE project. As this middleware proved to be insufficiently stable for our needs, we introduce in Section 4 a new computing environment, aimed at improving CONFIIT while responding to PER-MARE requirements. The general architecture of this new middleware is presented, and a first prototype is evaluated on a benchmark against Hadoop, raising several insights that will help the development of this middleware.

2 Defining a P2P Computing Middleware for PER-MARE

2.1 Computing models: FIIT versus Map-Reduce

2.1.1 FIIT

The notion of FIIT applications was defined in [9], and they are composed by a Finite number of Independent and Irregular Tasks (FIIT). We assume that each task satisfies the following features:

- A task cannot make any hypothesis on the execution of any other one. Hence, there is no communication between tasks.
- Execution time of each task is unpredictable. In other words, we cannot have a good estimation of the execution time before a task ends.
- A same algorithm is applied to compute all tasks. Hence, two tasks are distinguished by the set of data they have to process.

This computing model allows the representation of most parallel problems that do not require strong dependency among tasks. Please note that this computing model can be extended to support more complex algorithms through the use of large and fine-grain synchronization:

With a large-grain synchronization, two or more "rounds" of FIIT can be executed in sequence, synchronized at the end of each run. This model roughly corresponds to the well-known BSP (Bulk-Synchronous Parallel) programming model [11], which major characteristic is the succession of *supersteps*. A superstep is defined as a sequence of local operations, followed by a global synchronization barrier. On BSP, no assumption is made about the order of the tasks execution neither on the communication delivery; the only requirement is that all data necessary to start a new

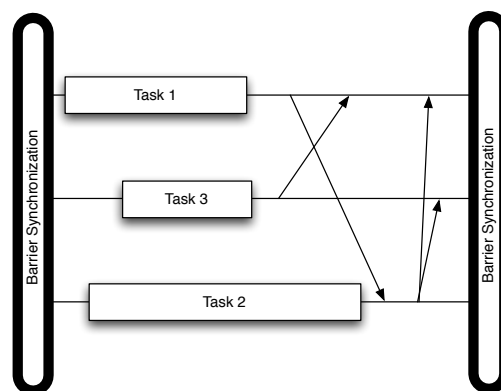


Figure 1 - Execution of a BSP superstep

superstep must be locally available at each node at the beginning of the new superstep. Figure 1 illustrates the execution of a BSP superstep, in which nodes execute their tasks and prepare the next superstep by exchanging data, prior to the barrier synchronization.

The fine-grain synchronization on FIIT can be easily obtained by defining some form of priority among tasks, like a Directed Acyclic Graph (DAG), in which some tasks may depend on the results of other tasks. While this synchronization breaks one of the assumptions of the FIIT model (the independency of the tasks), it can be easily implemented, adding an interesting feature to the FIIT model. For this reason, we can assume that an implementation of FIIT also allows a FIT (Finite and Irregular Tasks) application to be deployed.

A first FIIT implementation, called CONFIIT (Computation Over Network for FIIT), was introduced in [9] as a middleware for peer-to-peer computing. CONFIIT was designed to distribute over the network all the tasks obtained by decomposition of a FIIT problem. At the end of the tasks computation, the working nodes spread the results all over the network.

CONFIIT nodes are structured around a logical oriented ring, set up and maintained by the topological layer of the system. Basically, each node knows its predecessor and successor, and communications between nodes are achieved using a token that carries the state of computation around the ring. As the token travels, it spreads the results of every task so that nodes can follow the progression of the calculus. This diffusion mechanism also improves the overall fault tolerance of the middleware, as any node that stores the tasks results is able to compute the final result by its own, or to restart an execution that was interrupted.

Since constraints of a given application could be different and sometimes in contradiction (fault tolerance, efficiency, etc.), CONFIIT offers two main programming models: distributed and centralized mode.

The distributed mode allows a high fault tolerance level in the computation since task results are distributed to each node in the community. Thus, a broken computation can be re-launched using already computed tasks. Figure 2 shows information exchanges in the community for a distributed application. At first, the launcher sends the computing request to a node. The request is propagated along the community by the token. During computation, results of individual tasks are propagated across the community such that each node could locally store all individual results (data blocks). Concurrently to the computations, information on the global computation is exchanged through the token. Another interesting point from this mode is that the launcher only needs to be connected during the initiation phase. At the end of the computation, the global result can be retrieved from any node in the community.

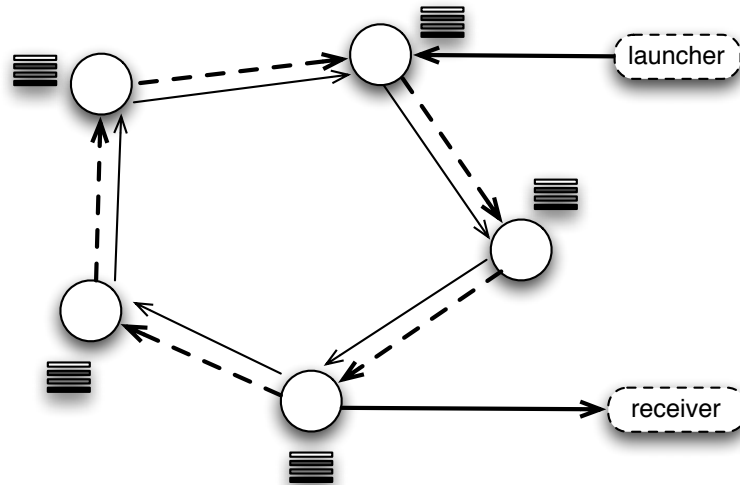


Figure 2 - CONFIT Distributed mode

The centralized mode reduces the global load of storage space and network communication, with the drawback of reducing fault tolerance. This mode is especially suited for the integration of external applications, which manage themselves the data storage. Figure 3 shows information exchanges in the community for a centralized application. During computation, results of individual tasks are sent to the initial launcher, which has the storage in charge (data blocks). As in the distributed mode, information on the global computation evolution is updated through the token. In the case of a crash on the launcher, partial results will be blocked on the computing nodes until the launcher is restored. Contrarily to the distributed mode, however, a computing node alone cannot restore the last known "checkpoint" as it does not stores the partial results from the other nodes.

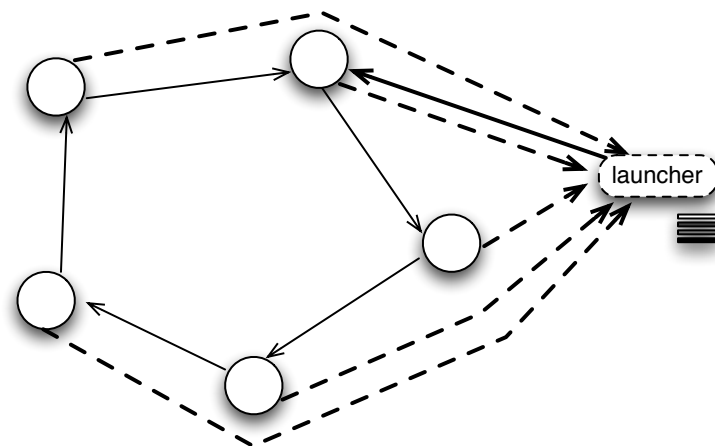


Figure 3 - CONFIT Centralized mode

2.1.2 Map-Reduce

Map-Reduce [3] is a parallel programming paradigm successfully used by large Internet service providers to perform computations on massive amounts of data. After being strongly promoted by Google, it has also been implemented by the open source community through the Yahoo! project,

maintained by the Apache Foundation and supported by Yahoo! and even by Google itself. This model is currently getting more and more popular as a solution for rapid implementation of distributed data-intensive applications.

Computations on Map-Reduce deal with pairs of key-values (k,V), and a Map-Reduce algorithm (a job) follows a two-step procedure:

- 1) **map**, from a set of key/value pairs from the input, the map function generates a set of intermediate pairs: $(k_1;v_1) \rightarrow \{(k_2;v_2)\}$;
- 2) **reduce**, from the set of intermediate pairs, the reduce function merges all intermediate values associated with the same intermediate key, so that $(k_2;\{v_2\}) \rightarrow \{(k_3;v_3)\}$.

When implemented on a distributed system, the intermediate pairs for a given key k_2 may be scattered among several nodes. The implementation must therefore gather all pairs for each key k_2 so that the reduce function can merge them into the final result.

Additional features that may be implemented by the Map-Reduce implementation include the splitting of the input data among the nodes, the scheduling of the jobs' component tasks, and also the recovery of tasks hold by failed nodes. One of the most popular Map-Reduce implementations, Hadoop, provides these services through a dual layered architecture. Indeed, tasks scheduling and monitoring is accomplished through a master-slave platform (*Job Tracker* \rightarrow *Task Trackers*), while the data management is accomplished by a second master-slave platform (*Name Node* \rightarrow *Data Nodes*) on top of the hierarchical HDFS file-system, as illustrated in Figure 4.

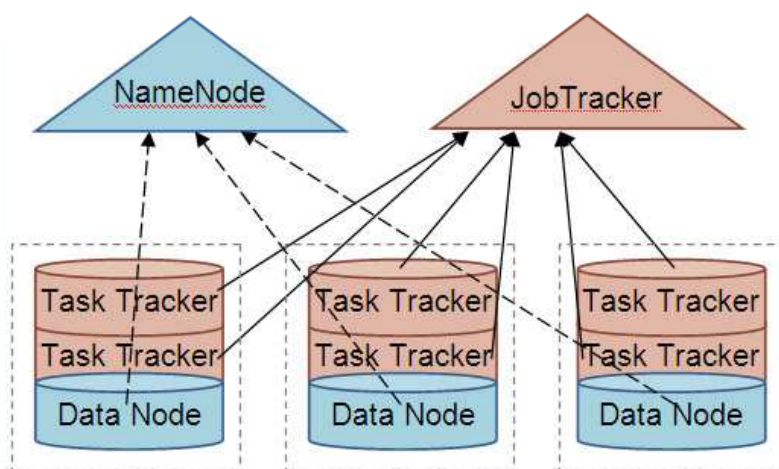


Figure 4 - Hadoop master-slave architecture

While the close collaboration between these two layers ensures good performance levels, it also introduces several constraints and fault tolerance issues. One of the most evident limitations is related to the presence of single points of failure on the *JobTracker* and the *NameNode*. Indeed, failures on *Task Trackers* and/or *Data Nodes* can be recovered, but a crashed *JobTracker* or *NameNode* blocks the entire platform. Also, the Hadoop platform does not allow new nodes to join the cluster during the execution of a job. This limitation is mostly due to the hierarchical structuration of the *DataNodes*, whose restructuring in the case of a join would introduce a non-

negligible overhead. For instance, no new node can be included in the cluster unless the reconstruction of the entire HDFS tree.

While some works like MAPR [12] try to overcome the first limitation through the replication of JobTracker and NameNode, there are few efforts to allow a more dynamic management of node volatility.

2.1.3 Implementing Map-Reduce over FIIT

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

Implementing Map-Reduce over CONFIIT 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, under the "Distributed" mode. 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 CONFIIT 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 CONFIIT, Map-Reduce 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 CONFIIT community, it receives a copy of the working data and may pick up available (incomplete) tasks on the shared task list. Thus, CONFIIT should offer a more fault tolerant behavior than Hadoop, supporting not only nodes disconnections, but also nodes (re-)connection.

As stated in the project planning, the beginning of PER-MARE WP2 works consisted on the development of a prototype of Map-Reduce on top of CONFIIT. This prototype of the *WordCount* application shares the same core elements as its Hadoop counterpart. Indeed, in order to better compare CONFIIT solution with Hadoop, we have developed two word count applications, one based on Hadoop (named *WordCounter*) and one based on CONFIIT (based on the *MapReduceConsumer* class). Both share a single core implementation, named *CounterExample* (see Figure 5).

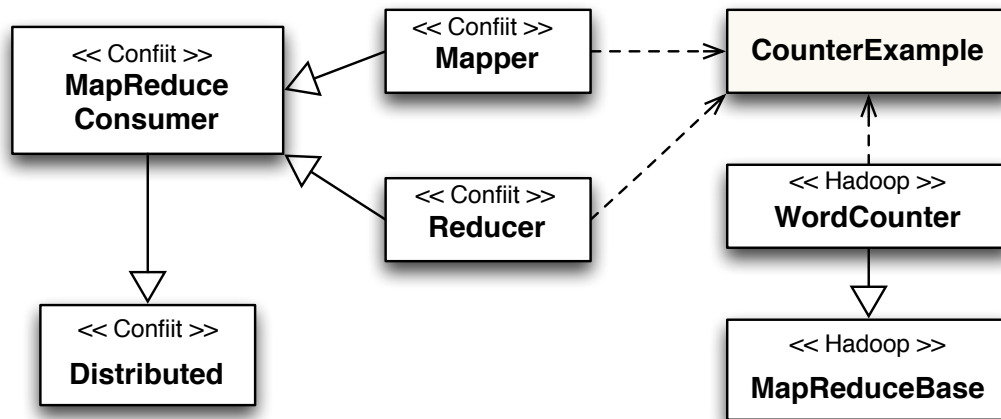


Figure 5 - WordCount core classes

Because Hadoop relies on specific classes to handle data, we tried to use the same ones in CONFIIT 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 (in the next WP2 tasks we shall reinforce the compatibility with Hadoop API). For instance, we were not able to use the Hadoop *OutputCollector* class, and a *MultiMap* class, illustrated in Figure 6, was developed to replace it.

Hadoop *OutputCollector* is an interface that generalizes the facility provided by the Map-Reduce framework to collect data output by either the *Mapper* or the *Reducer*, *i.e.* intermediate outputs or the output of the job. Although it can be used to store all values, *OutputCollector* implementations usually store the pairs in temporary files, which are handled by the HDFS layer to scatter data among the computing nodes.

In the case of CONFIIT, we cannot rely on a storage layer, so we need a data structure that allows the storage and manipulation of multi-value pairs $(k, \{V\})$. Java contains several classes for collections, as for example *Map* or *Hashlist*, but in all cases only a single value is allowed by key. Any additional element stored with the same key will replace the precedent one or raise a collision. In the case of Map-Reduce, however, several values with the same key may be produced during the *Map* phase (in one or several nodes), and the set of all values under the same key shall be available during the Reduce phase. The *MultiMap* class extends this notion of *Map*, as defined by the homonymous Java interface, in order to allow such multiple data representation. Instead of keeping just one value for each single key, *MultiMap* keeps a small Collection, whose size can be set as a parameter, allowing multiple values to be collected and referenced by the same key. Thus, *MultiMap* class, represented in Figure 6, accepts multiple values associated with a single key, allowing *CounterExample* to handle map and reduce outputs independently of Hadoop or CONFIIT implementations.

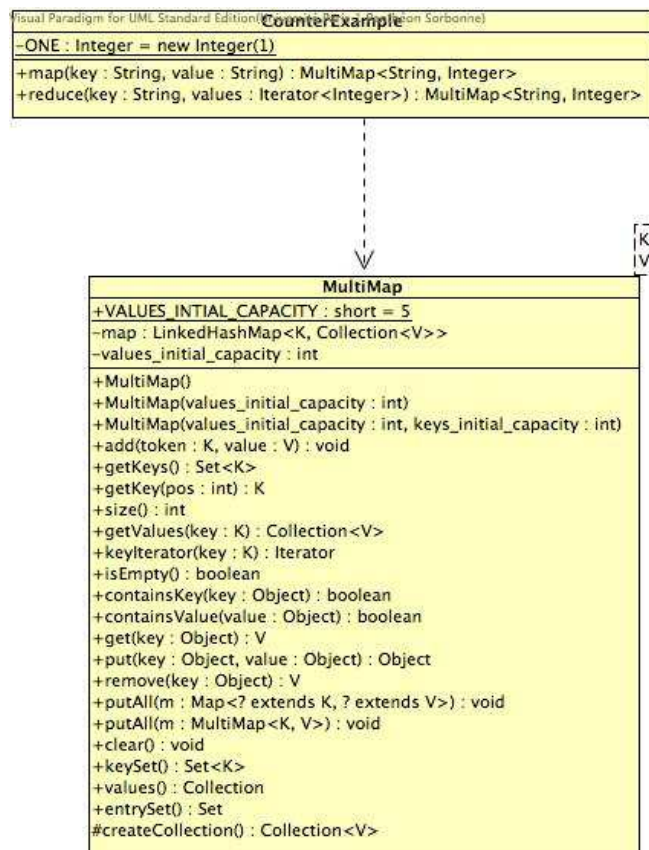


Figure 6 - Generic word count application and its `MultiMap` class.

Thus, an initial example of MapReduce over CONFIT was proposed, using *MultiMap* structure and *CounterExample* core class. As indicated in the previous paragraphs, this first prototype uses CONFIT Distributed mode and organizes MapReduce on a two rounds execution. 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 CONFIT ring at the time Reduce starts (this number may varies later, due to nodes volatility). This very first prototype demonstrated several drawbacks on original CONFIT, forcing us to consider its evolution. Section 3 details the limitations we have face on CONFIT, while Section 4 introduces evolutions we have performed on this platform.

3 Analysis of CONFIIT and its Drawbacks

3.1 Changelog of CONFIIT

CONFIIT was initially developed in 2003 [3]. The basic concept was to develop a middleware for distributed computing, structured around a logical ring overlay and XML-RPC exchanges among the nodes. These choices were also driven by the interest to provide a lightweight middleware, in opposition to other systems like DIET [1], which was based in CORBA and complex tree routing among the nodes. As a consequence, CONFIIT creates and maintains its own P2P overlay network in an attempt to keep the network stack at its strict minimum, which could not be obtained if it relies on a 3rd part P2P middleware.

Following the initial development, CONFIIT suffered a major update in 2004-2006, with the addition of several features like computation modes (Distributed, Centralized), sandboxing, external observers, etc. This work was conducted in the context of a French project supported by the ANVAR agency and a tentative to create a startup company [10].

All these versions share a basic design around a logical ring and node communication through sockets and XML-RPC. While there were some bugfixes on the following years, the 2006 version is the base of the current version of CONFIIT, and was used as the main platform for several experiments, as [18, 8, 5].

3.2 Limitations of CONFIIT when dealing with Map-Reduce problems

The works on WP2 for the PER-MARE project were naturally oriented towards CONFIIT as a P2P middleware solution for Map-Reduce. Indeed, as stated in Section 2.1.3, Map-Reduce and FIIT share several similarities that can lead to a rapid prototyping of Map-Reduce applications over CONFIIT.

The *WordCount* application was then prepared and we started different tests to evaluate the feasibility of Map-Reduce on a P2P network. Soon enough, however, we observed different problems related to the ring stability and to the token passing mechanism that prevent a good evaluation of the environment. Indeed, when launching the *WordCount* application, several disruptions were observed:

- 1) Ring stability – some nodes simply "drop out" of the ring, with no evident reason
- 2) Data diffusion –depending on the amount of data exchanged among the nodes, diffusion doesn't work properly and most data was not exchanged at all.

After investigation, these problems proved to be related to some conception flaws of CONFIIT, which was not prepared for large data exchanges. Indeed, when CONFIIT was developed back in 2003, the main focus of HPC computing was on computing-intensive applications, rather than data-intensive ones. As a result, CONFIIT was tested only with applications that require several hours of calculus but produce a relatively small output. This is the case, for example, of the Langford's problem and SAT optimization.

In the case of Langford's problem, the input is a simple combinatorial tree that will be split among the tasks, and the output consists on the sequences that fit Langford's requirement. As the current

record of Langford's problem is $L(2,24)$, this means that a solution consists on a simple sequence of 24 "colors", which hardly require more than a few bytes to be represented. Similarly, SAT optimization problems are focused on finding a Boolean satisfiability solution (or the absence of solution) for a given set of clauses. Hence, the tasks results contain the literal values of the solutions, if any. In both cases, a reduced volume of data is sent among the nodes.

A simple Map-Reduce application like *WordCount*, however, has a small CPU requirement but produces a large amount of data. Indeed, the intermediate data between map and reduce is larger than the original data as it includes the "annotations" on the number of occurrences of a word. Raising the data size to a few KB was enough to overload the P2P overlay, with result that not even 4 machines were able to keep a ring.

Initial attempts to correct the P2P management on CONFIIT proved to be too hard to be conducted. Hence, the source code of CONFIIT proved to be too complex and bad designed to allow its maintenance. For instance, a profusion of threads and sockets were used to manage message passing, with parts of the XML-RPC messages being parsed at different levels of the code (instead of using encapsulation). This complexity prevented any tentative to "plug" a 3rd part P2P overlay like Pastry [16], which would provide more stable and performing P2P services than the original ring topology managed by CONFIIT.

After several unsuccessful tentative runs using CONFIIT with only a very few nodes and datasets, we decided to abandon the old CONFIIT and develop a new one from scratch. Although extremely radical, this decision was motivated by the maintenance cost of the ancient code, which was too high to meet the project resources and requirements. This new P2P middleware, called CloudFIT, will be described in the next section.

4 Development of CloudFIT, a successor for CONFIIT

When the flaws on CONFIIT were detected, two main paths were available: try to correct CONFIIT or create a new "cleaner" version of the middleware. The cost to refit CONFIIT proved to be too high, and its complexity prevents any tentative to "replace" some parts of the code.

Therefore, we now are working on CloudFIT, a new FIT implementation that aims to succeed CONFIIT while providing much better maintainability and flexibility.

4.1 Design choices of CloudFIT

The discussions that follow the decision to develop CloudFIT lead to the definition of some design rules that must guide this development and future works:

- **[R1]** CloudFIT must be independent of any P2P middleware. This choice allows future works to test/implement different P2P overlays, adapting therefore to the execution environment and application requirements.
- **[R2]** Modularity is essential. CloudFIT must be able to evolve and adapt to different situations/environments through the composition of new CloudFIT modules. CloudFIT shall allow the users to "replace" elements from the CloudFIT "stack" without inducing modifications on the CloudFIT source code.
- **[R3]** The functionalities of a CloudFIT module may be implemented in different ways, using different algorithms and tools. A seamless integration of different implementations requires the use of well-defined interfaces that specify all methods used on the communication between modules.
- **[R4]** Strong coupling between modules shall be avoided. Ideally, the user shall be able to compose the CloudFIT "stack" through a property file or runtime parameters.

The requirement [R1] comes directly from the observation that a rigid dependency on a given P2P middleware may block future improvements to the system. Indeed, we shall be able to provide a computing framework that lies on top of the most adequate middleware for a given execution environment. As pervasive networks tend to vary a lot in their requirements and capabilities, the network level must be able to abstract implementation specificities and allow the use of any solution that complies with the execution environment. Some examples of this variability include non-broadcasting networks, proxies, tunneling and even the support to shared-memory communications, if supported by the environment.

The rule [R2] is also a prerogative to ensure the evolution and maintainability of the platform. By relying on a modular design, the different modules that compose the CloudFIT stack may be replaced by new modules that provide additional features or that support specific requirements from the environment or the application. This requirement also reinforces the exploratory role of CloudFIT, allowing the comparison between different modules, algorithms or networks.

The requirement [R3] imposes the need of well-defined interfaces for each module/level. By relying on interfaces instead of "hard-wired" references, we not only guarantee the modularity required on [R2] but also provide the ground for requirement [R4].

Finally, the requirement [R4] states two different design rules. The first one is partially a consequence of the previous requirements, but must be restated to recall good design practices. Indeed, the less coupling exists on a code, the more flexible it becomes, requiring minimal or no modifications in the case of module replacement. As a design goal all modules shall be independently instantiated and combined, and the composition of the modules may be guided by the end user through external tools like properties files, without any modification on the source code.

4.2 Implementing CloudFIT – Architecture

The design of CloudFIT architecture follows the requirements stated in Section 4.1, especially [R1] and [R2]. The architecture we propose is inspired on the stack models from TCP/IP and OSI, and is presented in Figure 7. For instance, we defined four stack layers that represent different functionality layers usually found on these protocols models: **Network**, **Protocol**, **Service** and **Application**.

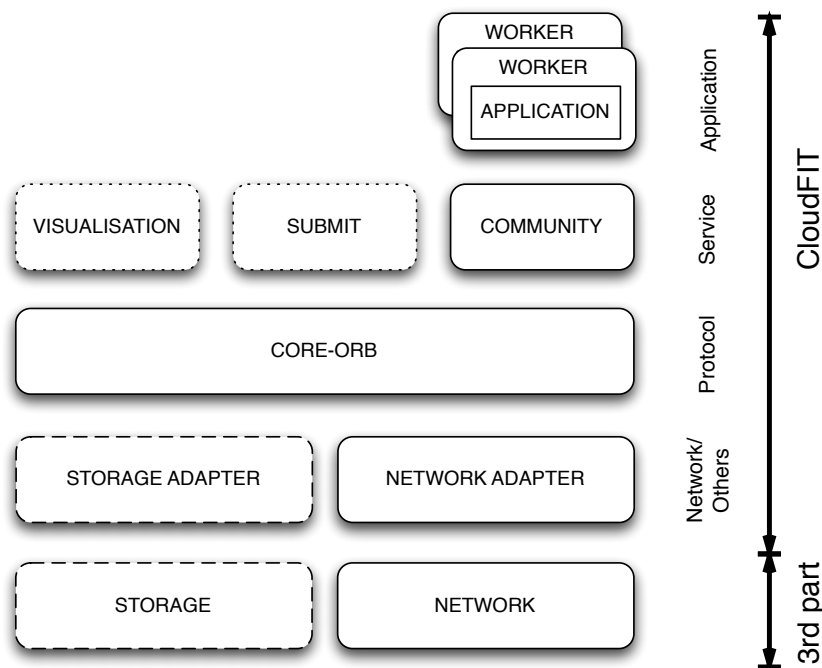


Figure 7 - CloudFIT stack and some prototype modules

The **Network layer** is responsible for the interfacing with the underlying system. Its main responsibility is to adapt to the different P2P middlewares (mostly provided as 3rd part libraries), providing message encapsulation/decapsulation and a set of basic communication primitives (*send*, *sendAll*, etc.). This layer also deals with storage solutions (local files, DHTs, databases, cloud storage) that may be offered to the upper-layers, and implements basic storage primitives (*read*, *write*, *delete*, etc.).

The **Protocol layer** is the first "entry" layer of CloudFIT messages. Its roles include the buffering of incoming messages and the delivery of message to the appropriate upper-level services. For instance, a service "subscribes" to this layer with a unique identifier. This identifier will be used to direct incoming messages to the right service, and also allow the communication between services inside the same node. As the module from this layer plays a central role in the overall framework to connect services and low-level modules, its role can be compared to that of an Object Request Broker, or *ORB*.

The **Service layer** is composed by most of the support services that contribute to the execution of a distributed computing application. The minimal service module is the *Community*, an abstraction of the P2P network responsible by the deployment of the computing jobs and the management of nodes events (joining, leaving). Additional services include *Submission* and *Visualization* frontends, which allow more flexibility on the use of the framework. Other services that can be created include a "network observer", which tries to estimate the number of nodes (and cores) in the network (in a P2P network it is hard to have a deterministic value).

Finally, a main job manager and one or more Workers compose the **Application Layer**. The number of *Workers* depends on the nodes capabilities and application requirements. Workers are managed by a class called *ThreadSolve*. Each *ThreadSolve* is responsible for one job instance, collecting the partial results from the different tasks executed by the node's Workers or by other nodes, as well as summing-up the tasks results to produce the job final result (similar to the Combine phase on Map-Reduce).

4.3 Implementing CloudFIT – First prototype

A first prototype of CloudFIT has been produced in early July 2013. For this first prototype, we based the network layer on the P2P library EasyPastry [6], which is a simple wrapper around FreePastry and the Bunshin DHT [7] storage solution.

FreePastry is an open-source implementation of the Pastry P2P overlay, which is structured around a logical ring topology. Each node is identified by a unique key, with a 2^{160} keyspace (SHA-1). The key may be randomly generated or follow a predefined arrangement, like when using the *IPNodeIdFactory* method. Routing of messages can be done with direct messages (if one does know the ID of the destination) or through "closest matching" routing, i.e., the message will be delivered to the node with the closest ID, as illustrated in Figure 8.

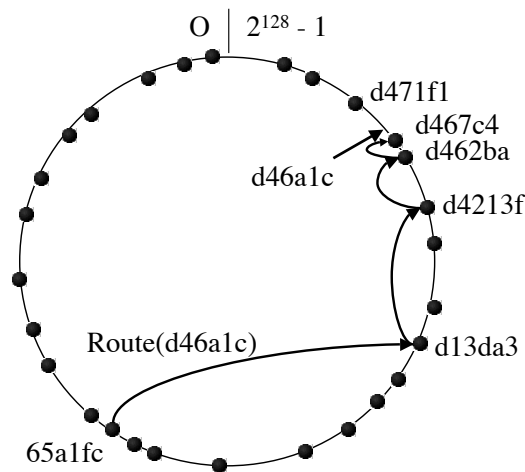


Figure 8 - Pastry routing mechanism [2]

While FreePastry provides all elements necessary to establish a P2P overlay (and even additional services like the PAST DHT and the Scribe group communication tool), our attention was drawn to a complementary library called EasyPastry [6]. Not only EasyPastry simplifies the initialization of a Pastry network (through a Façade pattern), it also includes a different DHT, called Bunshin [7]. Contrarily to most DHTs, Bunshin is supposed to support data replication, which can be extremely useful in a volatile environment like the pervasive networks.

For these reasons, the first prototype created a *NetworkAdapter* and a *StorageAdapter* around EasyPastry, as illustrated in Figure 9. A secondary *StorageAdapter* with local file storage was also implemented.

The following layer, **Protocol**, is represented in Figure 10. The *CoreORB* class intermediates the communication between the other layers, and is assisted by a *CoreQueue* class, responsible by the reception of incoming messages and the delivery of the messages to the right service. Instead of a simple Observer pattern that notifies all observers that subscribed the queue, we used an association of message delivery and *java.concurrent* classes to notify only the service that is concerned by a given message. This mechanism allows a better thread management. Indeed, when *CoreQueue* delivers a message to a service, it puts the message in the service own *ActiveQueue*, which is awoken and can start processing the message.

In the example from Figure 10, a Community service has an associated *ActiveQueue*, which is fed by *CoreORB*->*CoreQueue* when a *JobMessage* arrives (*JobMessage* is encapsulated inside a more generic *Message* object).

Thanks to *CoreORB* and *CoreQueue*, several services may lie on top of the Protocol layer. Indeed, one of the main services in our prototype is the *Community* service, which is responsible by the management of new jobs and the exchange of tasks results among the distributed workers.

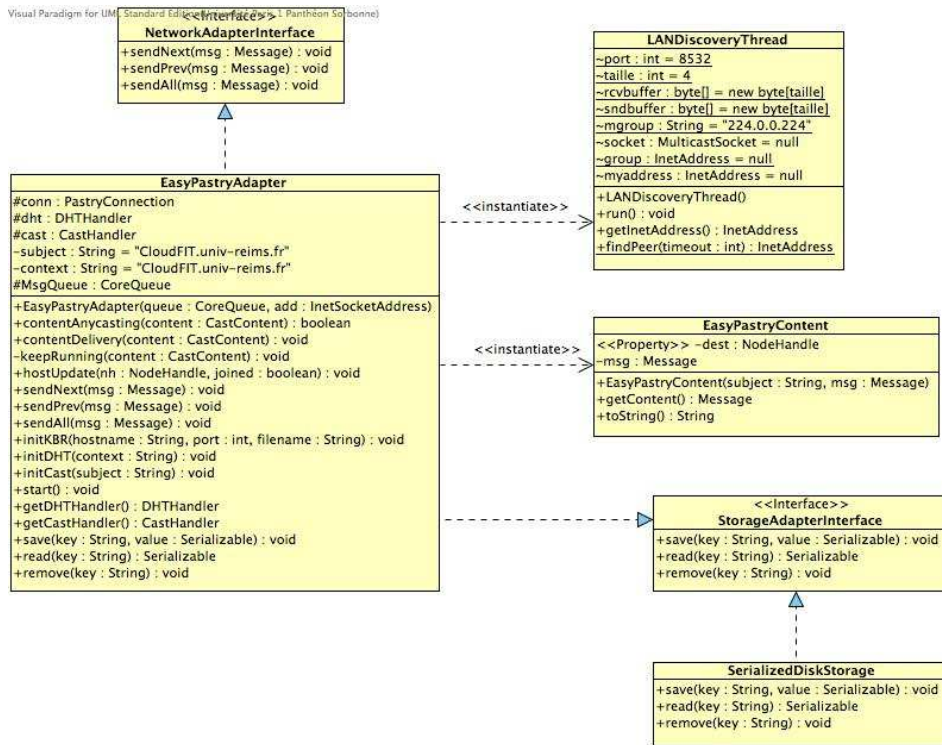


Figure 9 - Implementation of a Network layer adapter

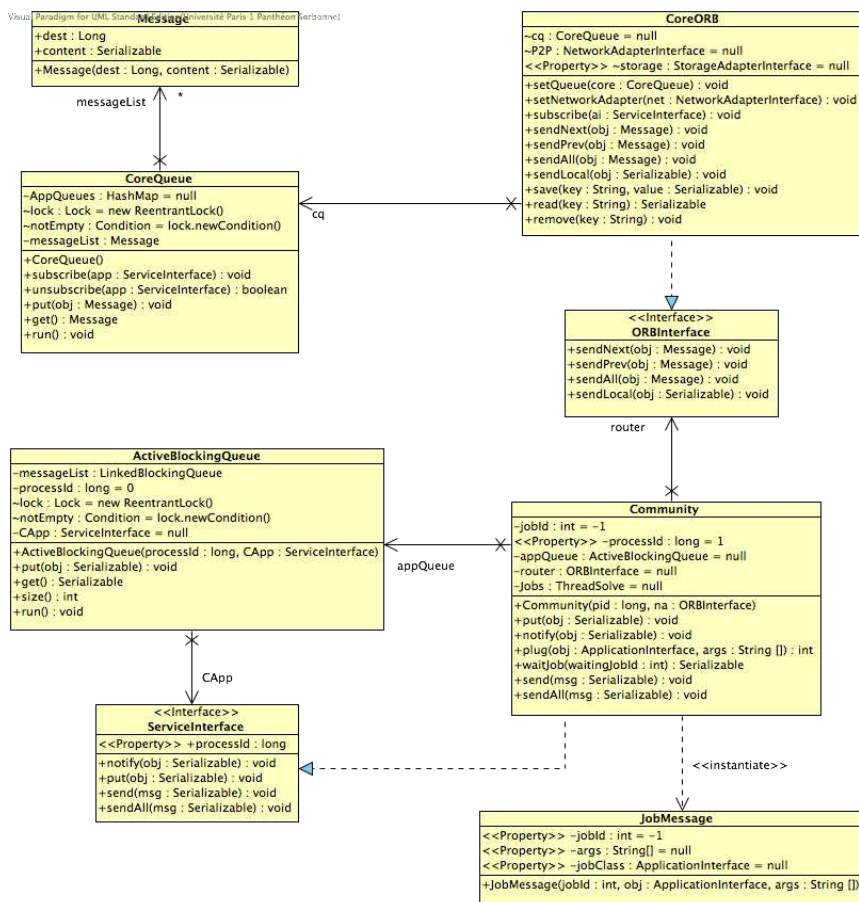


Figure 10 – Interaction between Protocol level and Service Level

In Figure 11, we can identify the main components of the *Community* service, as well as some elements from the Application layer. Indeed, the *Community* service will manage the submission of new jobs by assigning a *ThreadSolve* instance to each job. The *ActiveQueue* associated to the *Community* service will also deliver task messages to the corresponding *ThreadSolve* (these messages contain mostly the results from tasks executed in distant machines).

When a *ThreadSolve* is started, it will get the application parameters (class to execute, command-line parameters, etc.) and start a pool of workers. For the matter of performance, the pool of workers is defined to fit the number of computing cores on the node.

On the prototype, a simple "random" scheduler is used to prevent all nodes executing the same tasks: the task list is shuffled at each machine, resulting in a different task order to execute. Future version shall allow the addition of other scheduling mechanisms, allowing for example a context-aware schedule of the tasks.

While the *ThreadSolve* is responsible for the main administration of the job execution, it is the role of the workers to execute an application class derived from *ApplicationInterface*. Different variants like *Distributed* and *Centralized* are abstract classes derived from *ApplicationInterface* that set the basis of each variant operation.

At the end of each task execution, the status of the task is updated and the task result is broadcasted to the other nodes using *TaskStatusMessages*. This ensures that all nodes collaborate and avoid duplicate work. Similarly, results received from other machines will update the task status. When all tasks are done, the *ThreadSolve* class may compute the Job result, which will be stored using the resources from the storage layer.

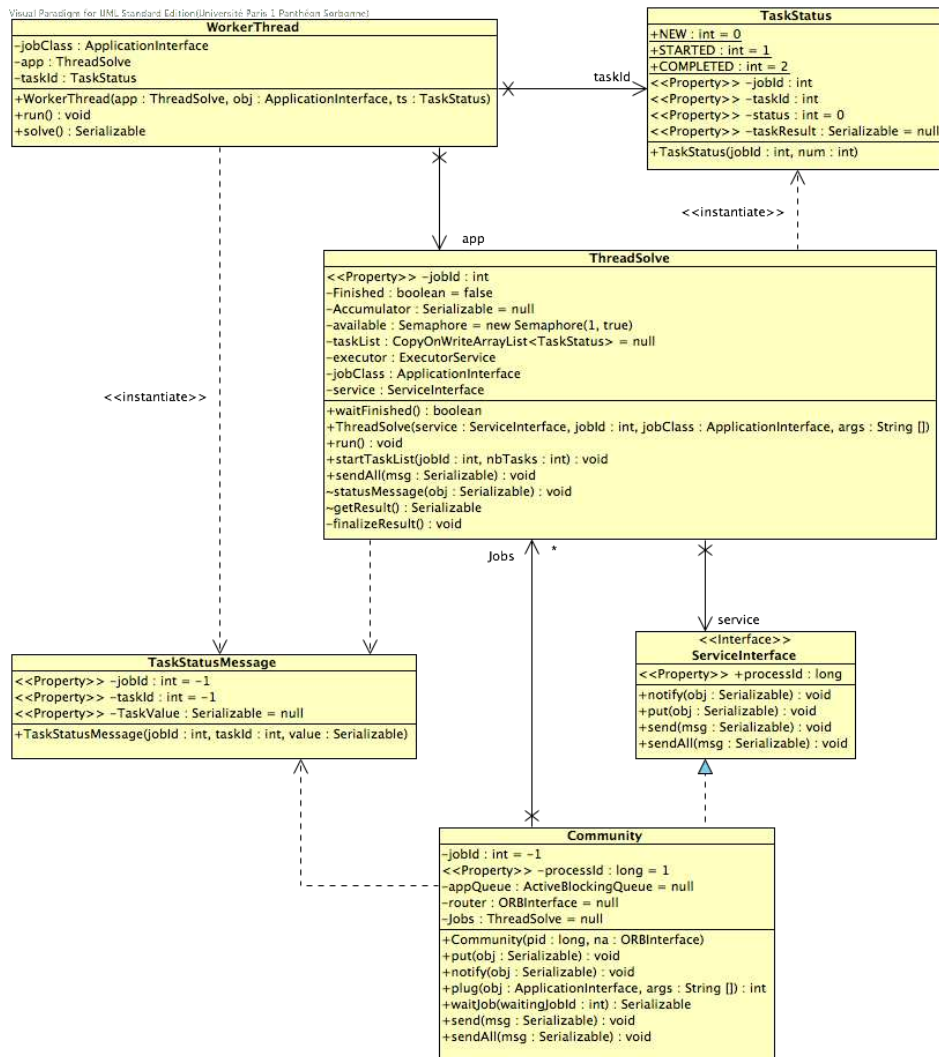


Figure 11 - Community Service and the Workers

4.4 Prototype evaluation

The first prototype was tested in preparation for the 3PGCIC conference [17], and compares the performance of CloudFIT against Hadoop, using the *WordCount* application. The experiments were conducted over 16 machines on the Helios cluster from the Grid'5000¹ network. Each machine from the Helios cluster is composed by 2 AMD Opteron 275 2.2GHz CPUs, totalizing 4 cores per node. 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

¹ <http://www.grid5000.fr>

² http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project

chose the *Gutenberg Project Science Fiction Bookshelf CD²*, which contains more than 200 books in text format. The results presented on Figures Figure 12Figure 13Figure 14 represent the median of the performed measures for 4, 8 and 16 nodes, respectively.

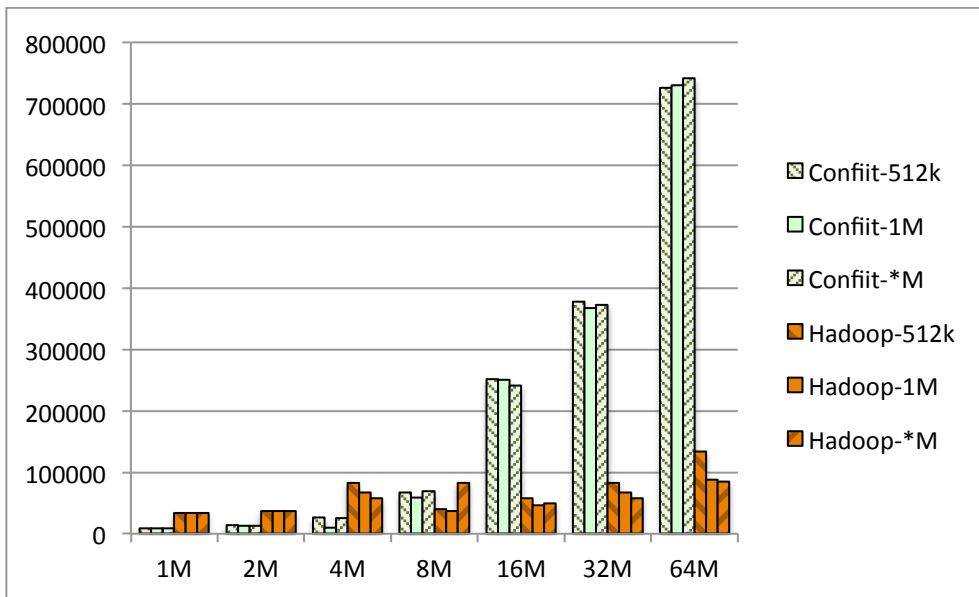


Figure 12 - CloudFIT vs Hadoop - 4 nodes

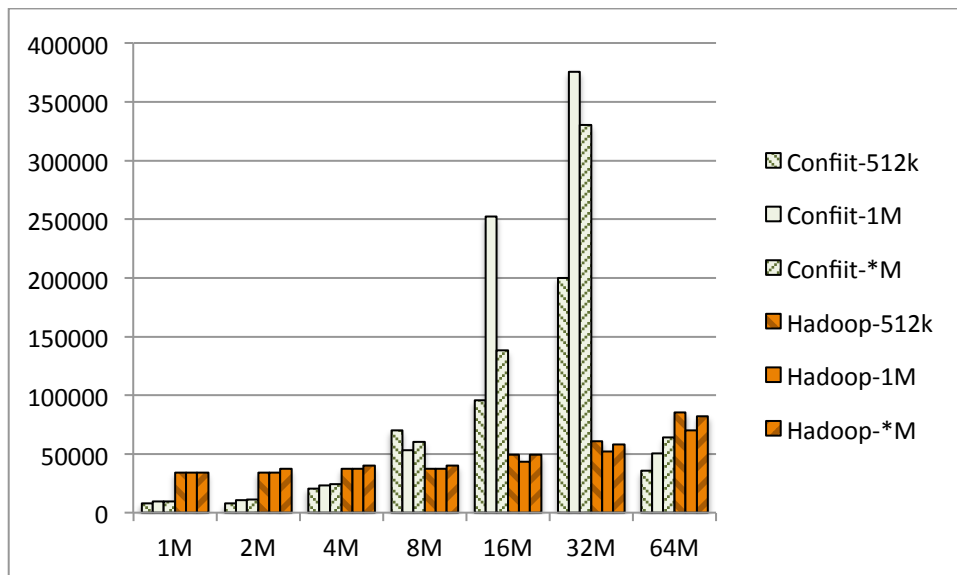


Figure 13 - CloudFIT vs Hadoop - 8 nodes

² http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project

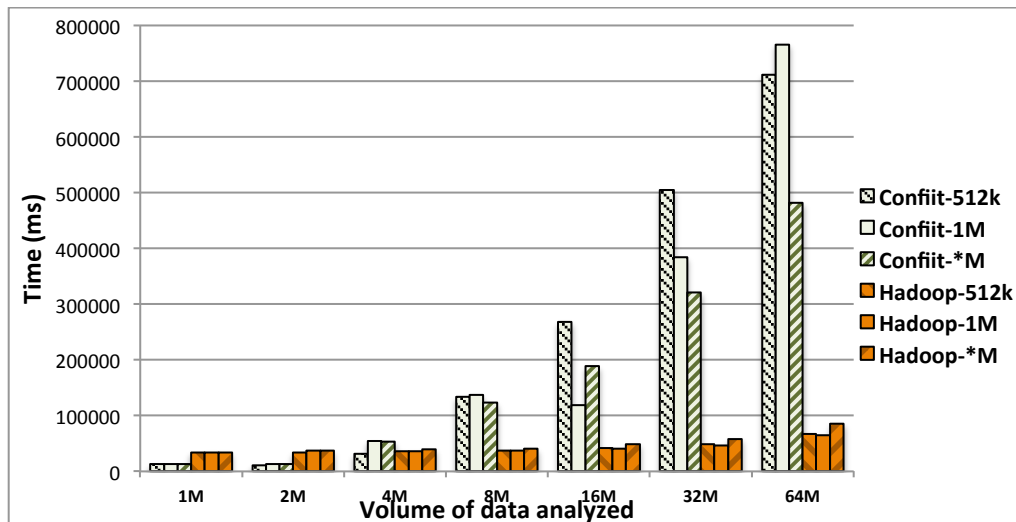


Figure 14 - CloudFIT vs Hadoop – 16 nodes

When analyzing the measures, two major scenarios arise: for small data volumes, our prototype outperforms Hadoop, while Hadoop performs much better for large data sets. In the first scenario, this is mostly due to CloudFIT lightweight middleware. However, when the data volume augments, we observe a huge performance slowdown, even if sometimes CloudFIT succeeds to perform better than Hadoop (as in the case of Figure 13, with 64MB of data).

Investigation shows that this is due to the task update pattern used on the Distributed mode, which overloads the token passing mechanisms and creates a bottleneck. More specifically, the current implementation of CloudFIT Distributed mode spreads results using "service" layer messages: the same messages that are used to keep nodes updated about the tasks completion are used to transmit the tasks results. For small data sets this procedure poses no problem but when the amount of data grows, the service layer becomes overloaded. As a consequence, nodes finish by computing most of the tasks locally because few updates are able to reach to the other nodes. The rare cases where tasks succeed to be diffused, as in the case from Figure 13, seem to validate the interest of the framework.

4.5 How to Improve Data Transfer

As the performance problem is due to the overload of the "service" network, the solutions for this problem passes by the use of specific data exchange channels, created on-demand by nodes that wish to complete their data sets after receiving an update message.

FreePastry proposes a solution for massive data transfers among nodes through the use of private channels known as *ApplicationSockets*. *ApplicationSockets* are especial communication channels that are established on an end-to-end base, and that do not interfere with the main service messaging. The establishment of an *ApplicationSocket* requires the server side to be in a "accept" mode and the client side must know the ID of the server node (for instance, the Pastry *NodeHandle* identifier). According to Pastry documentation, the *ApplicationSocket* benefits from Pastry mechanisms to traverse firewalls and NAT. Therefore, the addition of *ApplicationSockets* requires a new protocol to

exchange updates among tasks. Instead of directly broadcasting the results of each task, we shall split the update in two steps:

1. Broadcasting the "COMPLETED" status of a task, as well as the ID of node that has the results
2. Acquiring the task results directly from the "server", through *ApplicationSockets*.

As this solution is specific to FreePastry (even if other P2P middlewares shall have similar solutions), we are still deciding on how to implement it without violating requirement R1. The main difficulty resides on the fact that such specific solution must be contained in the Network layer, in spite of dealing with tasks results (that belong to the Application layer). By using specific message signatures (request messages) and allowing lower layers to access upper-layer data, we may limit the impact of this solution and keep CloudFIT compatible with requirement R1. In the next weeks we shall experiment this solution, which will be followed by other benchmarks. A comparison with basic TCP sockets is also possible.

An alternative to this solution also exists, passing through the storage system. Indeed, the use of an external shared storage medium like a DHT will allow workers to share the results of the tasks, all while minimizing the local storage requirements. This solution however needs to be better evaluated as the use of an external storage adds a non-negligible overhead to access the data, which may violate one of the main features of Hadoop, i.e., reinforce data locality to speedup the reduce phase.

Therefore, the next works on CloudFIT shall implement both approaches and compare them.

5 Conclusions

In this document, we have analyzed the main requirements to develop a P2P middleware capable of running MapReduce jobs. This analysis leads us to define how MapReduce-based applications could be deployed on a completely distributed computing environment, one of the goals of the PER-MARE project.

Our initial efforts to run MapReduce concentrate on the Confiit middleware [3, 5], whose "paradigm" FIIT can easily accommodate the MapReduce paradigm. While Confiit developed several interesting properties over the years, it proved to be too unreliable to support the PER-MARE developments. For this reason, we decided to begin a new P2P computing middleware, named CloudFIT. While CloudFIT is inspired on Confiit, it is designed to be much more modular and flexible than its predecessor, allowing for example the addition of application-specific modules, and the independency from the P2P overlay technology.

This deliverable presents CloudFIT main architecture and the development directions (requirements) we set to ensure its properties. A first prototype of CloudFIT is presented in this report, using the FreePastry [16] P2P overlay. We conducted different performance benchmarks against Hadoop, and the results of this comparison raise several hints to improve CloudFIT performance. For instance, the last part of this report discusses these tests and concludes the text with comments on the performance issues on the first prototype, whose solutions shall be detailed in a subsequent deliverable.

6 References

1. Caron E. and Desprez F. "DIET: A scalable toolbox to build network enabled servers on the grid". *International Journal of High Performance Computing Applications*, 20(3):335-352, 2006.
2. Castro M., Druschel P., Kermarrec A-M. and Rowstron A., "One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks", SIGOPS European Workshop, France, September, 2002.
3. Dean J. and Ghemawat S., "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
4. Flauzac O, Krajecki M, Fugère J. "CONFIIT: a middleware for peer to peer computing". In: Grailova M, Tan C, L'Ecuyer P (eds) *The 2003 international conference on computational science and its applications (ICCSA 2003)*. LNCS 2669. Springer, Berlin, pp 69–78
5. Flauzac, O., Krajecki, M., Steffene, L.A. "CONFIIT: a middleware for peer-to-peer computing". *Journal of Supercomputing*, Springer, vol 53 n. 1, July 2010, pp. 86-102.
6. <http://ast-deim.urv.cat/easy Pastry/>
7. <http://planet.urv.es/bunshin/>
8. Jaillet C. and Krajecki M. "Solving the Langford problem in parallel". In *International Symposium on Distributed and Parallel Computing (ISPDC'04)*, pages 83--90, IEEE Society Press, Cork, Ireland, jul. 2004.
9. Krajecki M, "An object oriented environment to manage the parallelism of the FIIT applications". In: Malyskin V. (ed) *Parallel computing technologies, 5th international conference, PaCT-99*. LNCS vol 1662. Springer, Berlin, 1999. pp 229–234
10. Krajecki M., Flauzac O., Mérel P.P., Fidanza S., "Start-up - Incubateur ICAR", 2004
11. Leslie G. Valiant: A Bridging Model for Parallel Computation. *Communications of the ACM* 33(8):103-111, 1990
12. MAPR, <http://www.mapr.com/>
13. Mérel P.P., Krajecki M., Flauzac O., Boivin S., Jaillet C. "Utilisation de la grille CONFIIT pour la résolution du problème de Car Sequencing", *Workshop sur la Résolution Parallèle de Problèmes NP-Complets (NP-PAR), Rencontres francophones du Parallélisme (RenPar'16)*, 2005
14. PER-MARE: Adaptive Deployment of MapReduce-based Applications over Pervasive and Desktop Grid Infrastructures, Project Proposal, Regional Program STIC-AmSud 2012 [PER-MARE internal document]
15. Refs limitations on MapReduce
16. Rowstron A. and Druschel P. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, pages 329-350, November, 2001.
17. Steffene L. A., Flauzac O., Schwertner Charao A., Pitthan Barcelos P., Stein B., Nesmachnow S., Kirsch Pinheiro M., Diaz D. "PER-MARE: Adaptive Deployment of MapReduce over Pervasive Grids". *Proceedings of the 8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'13)*, Compiègne, France, October 28-30 2013.

18. Steffemel, L. A., Jaillet, C., Flauzac, O., Krajecki, M. "Impact of nodes distribution on the performance of a P2P computing middleware based on virtual rings". Proceedings of the Conferencia Latino Americana de Computación de Alto Rendimiento (CLCAR 2010), Gramado, Brazil, 25-28 August 2010