SUPPORT FOR MANAGING DYNAMICALLY HADOOP CLUSTERS

by

David de Nadal Bou

Project director: Yolanda Becerra

September 2010
PROJECT DETAILS

Title: SUPPORT FOR MANAGING DYNAMICALLY HADOOP CLUSTERS

Student: David de Nadal Bou

Degree: Master in Information Technologies

Credits: 30 ECTS

Project director: Yolanda Becerra Fontal

JUDGES

Ruth Raventos Pages

Joan Carles Gil Martin

Maria Dolors Costal Costa

GRADE

Numerical grade:

Descriptive grade:

Date:
# INDEX

Abstract.............................................................................................................................................. 1
Acknowledgements................................................................................................................................. 1
1. Introduction........................................................................................................................................ 1
   1.1. Problem statement.................................................................................................................. 1
   1.2. Objectives............................................................................................................................. 2
   1.3. Development strategy............................................................................................................ 2
2. Technology overview.......................................................................................................................... 4
   2.1. Cloud computing.................................................................................................................... 4
   2.2. MapReduce............................................................................................................................ 5
       2.2.1. Use cases........................................................................................................................ 5
       2.2.2. MapReduce implementations....................................................................................... 6
   2.3. Hadoop..................................................................................................................................... 8
       2.3.1. Project and sub-projects............................................................................................... 8
       2.3.2. Cluster overview........................................................................................................... 8
       2.3.3. HDFS............................................................................................................................ 9
       2.3.4. Dataflow........................................................................................................................ 10
       2.3.5. Hadoop internal scheduler.......................................................................................... 11
   2.4. Virtualization............................................................................................................................ 12
       2.4.1. Full virtualization........................................................................................................... 12
       2.4.2. Paravirtualization.......................................................................................................... 12
       2.4.3. Virtualization platforms............................................................................................... 13
   2.5. EMOTIVE Cloud....................................................................................................................... 14
       2.5.1. Architecture.................................................................................................................. 14
   2.6. REST....................................................................................................................................... 16
       2.6.1. RESTful web services...................................................................................................... 17
3. Requirements....................................................................................................................................... 20
   3.1. Functional requirements........................................................................................................... 20
   3.2. Non-functional requirements.................................................................................................... 20
4. Specification......................................................................................................................................... 28
   4.1. System components.................................................................................................................. 28
   4.2. Use cases.................................................................................................................................. 29
5. Design and implementation............................................................................................................... 35
   5.1. Virtual machine....................................................................................................................... 35
   5.2. Hadoop Scheduler................................................................................................................... 38
       5.2.1. Conceptual model.......................................................................................................... 39
       5.2.2. HSP Functions............................................................................................................... 40
       5.2.3. Hadoop Scheduler functions......................................................................................... 44
   5.3. Graphical user interface........................................................................................................... 53
   5.4. Summary................................................................................................................................. 55
6. Deployment......................................................................................................................................... 57
   6.1. Release...................................................................................................................................... 57
   6.2. Installation............................................................................................................................... 57
   6.3. Future updates........................................................................................................................... 58
7. Testing................................................................................................................................................ 59
8. Project plan......................................................................................................................................... 63
9. Summary and conclusions................................................................................................................ 66
References............................................................................................................................................ 67
Abstract

Final master project developed in Barcelona Supercomputing Centre. This project focuses on the virtualization of Hadoop environments and the design of a piece of software for automatizing the configuration of the shared resources for Hadoop environments. In addition, this project uses a modified internal Hadoop scheduler that adapts the available resources in the cluster for running jobs according to time restrictions. With the adapted internal scheduler and the virtualization capabilities the software developed in this project provides a flexible and self-adaptive service for Hadoop environments.

In this report I introduce the context of the technology, the specifications of the system, the design of the solution integrated with EMOTIVE Cloud platform and the testing for showing the performance of the product.

Keywords: Hadoop, virtualization, cloud computing, Xen, Java, MapReduce, EMOTIVE, cloud, self-adaptive, scheduler, REST.
Acknowledgements

Thanks to Yolanda Becerra and David Carrera for the opportunity of taking part of their team and giving me the chance of developing this project with them. I would like to say thank to Jordà Polo, Íñigo Goiri and Alex Vaqué, because of their inestimable help. This project is dedicated to my family because of their patience and support over the years, and to all my friends for dedicating their time to me when I needed them. Thank you all. Gràcies.
1. Introduction

With the significant advances in Information and Communications Technology (ICT) over the last half century, there is an increasingly perceived vision that computing will one day be the 5th utility, after water, electricity, gas, and telephony.

The amount of information stored everyday not just in companies but in personal computers, and the variability on the software and hardware demand is pushing the research community to find new kind of infrastructures and systems that can handle large amount of data and the new requirements.

As a response to this demand paradigms like GRID or cloud computing emerged as possible solutions to solve the problems in the data processing and storage.

GRID computing aims to deal with every kind of resource, such as personal computers, supercomputers, data centres, peripheral hardware, etc. coordinating all the resources and making the infrastructure transparent to the end user, connecting the resources through a large area network, such as internet.

Cloud computing is another paradigm a bit less ambitious than GRID but more realistic. It's aim is offer computation services through internet and is based on the idea that everything can be on the “cloud” and the end user just requires a browser to access the information.

Old technologies such as virtualization, that appeared long time ago, are helping cloud computing becoming not just an idea but a reality. It's flexibility makes life easier for administrators and it's isolation power provides the required security to highly distributed systems.

In 2004 Google designed it's own programming model for cloud computing, they called it MapReduce. They created it as a possible solution to solve their problems for managing large amounts of data sets. The implementation of this programming model satisfied them at the point that decided to publish it, and now is one of the most popular solutions for managing large amounts of data in distributed systems.

The problem with Google's implementation for MapReduce is that is private. Other implementations for this programming model have appeared, some private and others open source.

Hadoop, from the Apache project, is becoming a popular implementation for MapReduce among the research community, due that it is open source and it is supported by big companies such as Yahoo! or IBM.

1.1. Problem statement

The Hadoop project is highly supported by the research community and big companies and the software solution is used in a lot of services we are daily using nowadays, such as Yahoo! searches or Facebook data administration, but at the beginning of this project, there was no publication of any virtualization approach for Hadoop.

Deploying Hadoop over physical machines can provide good results but it presents some difficulties for service providers in terms of management, due to the complexity of managing incoming requests in an efficient way. A lot of resources should be quickly coordinated and properly allocated, and the customers and not willing to wait for it's computing service.

This project appears like a possible solution to the difficulties in resources provisioning for computations in Hadoop environments.

Virtualization can solve those management problems for Hadoop as it does for other software solutions, giving dynamic resource allocation capabilities.
The dynamic resource allocation will give the customer the possibility of making requests to the system and, if possible with the available resources, set up the virtual Hadoop environments, or modify existing ones according with the requirements. Improving the flexibility for this service is going to make it much more interesting for the customer.

For the companies would be a better way of managing it's own resources, because one physical machine can host several virtual machines, and win in quality of service with the dynamic resource allocation.

The implemented system presented in this report virtualizes Hadoop environments, using EMOTIVE Cloud, and provides an interface to interact with the internal job scheduler present in Hadoop for collecting information about the job progresses over the running jobs. This information would be used by higher level scheduler to decide how to distribute the physical resources.

1.2. Objectives

The goal of this project is virtualize the Hadoop environment and design a piece of software for automatizing the resource allocation and manipulation of Hadoop environments. To achieve this goal some background documentation should be analysed and documentation should be done. In summary the project goals are:

- Study Hadoop environments and EMOTIVE Cloud
  - Understand how Hadoop works and how to configure it.
  - Familiarize with EMOTIVE Cloud code and project, which provides dynamic resource virtualization.
  - Ensure Hadoop can be virtualized using EMOTIVE Cloud.
- Design a dynamic resource manager for Hadoop
  - Automatize the virtual machine creation and the cluster configuration.
  - Provide a graphical user interface to manage the system.
  - Provide an interface for the internal Hadoop job scheduler.
- Document the project with a report describing it as well as its development and final outcome.

1.3. Development strategy

This chapter describes the methodology and strategy followed for the achievement of the specified goals.

The methodology followed fits on what is commonly known as Agile methodologies [10]. As far as this project was a research project included in EMOTIVE Cloud, that is a research project too, there was a high level of uncertainty on the possibility of achieving the goals of the project.

Plan-driven methodologies like RUP (Rational Unified Process) require more certainty on the achievement of the goals, due to the amount of documentation that should be done before the implementation phase.

Since the beginning of the project, from the background research to the implementation and deployment of the software, weekly meetings had been carried out with the other members of the
team to see how is everybody progressing. Every time there was a new functionality implemented and deployed we were showing the other members the new features aiming to have everybody up to date with the evolution of the project. If we think on the analogy with the agile methodologies, this time the stakeholders were ourselves and the customer was the project manager. In this way the customer was completely involved and informed of the project processes, as this kind of methodologies require.

The first step for every project is doing some background research and this was not an exception. Background on MapReduce, Hadoop and virtualization was required before the specification, design and implementation phases.

Due to the uncertainty of the project, after the documentation phase, the following step was ensuring that virtualization is possible for Hadoop. This means set up virtual machines, install Hadoop on the machines and ensure that the processes run correctly.

Once the virtualization was granted, the next step was automatizing the creation and manipulation of Hadoop environments. This means the completely integration with EMOTIVE Cloud, as EMOTIVE Cloud was the chosen platform to support the virtualization process. This part was approached as an extension of EMOTIVE Cloud and required the creation of a new interface to give access to the new feature.

Finally, once we had an extension on EMOTIVE Cloud that grants the automatic creation and manipulation of Hadoop environments, the integration with a Hadoop scheduler was carried out. This integration allows the scheduler provide information of the progress of the current tasks carried out by the created environment. This information will be very valuable to EMOTIVE Cloud because it will help to decide how to manage the resources and provide a better service.
2. Technology overview

2.1. Cloud computing

With the significant advances in Information and Communications Technology (ICT) over the last half century, there is an increasingly perceived vision that computing will one day be the 5th utility (after water, electricity, gas, and telephony). This computing utility, like all other four existing utilities, will provide the basic level of computing service that is considered essential to meet the everyday needs of the general community. To deliver this vision, a number of computing paradigms have been proposed, of which the latest one is known as Cloud computing.

Cloud computing is an emerging technology in the field of parallel and distributed computing. In Geelan (2009) [1], many experts define cloud computing with their points of view. Buyya, Yeo, and Venugopal (2008) [2] provide a very concise and clear definition of cloud computing:

“A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers”.

Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). From a hardware point of view, three aspects are new in Cloud Computing [3]:

1. The illusion of infinite computing resources available on demand, thereby eliminating the need for Cloud Computing users to plan far ahead for provisioning.
2. The elimination of an up-front commitment by Cloud users, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs.
3. The ability to pay for use of computing resources on a short-term basis as needed (e.g., processors by the hour and storage by the day) and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

Storage and compute clouds are two major types of clouds that aim to provide services without exposing the underlying infrastructure to customers. This project is focused in a computing approach for cloud computing called MapReduce and its open source implementation Hadoop.
2.2. **MapReduce**

MapReduce is a programming model designed by Google [4] that aims to be a solution to process large amount of data, such as web logs, crawled documents, etc. and derived data like the set of more frequent queries in a given day, inverted indexes, etc. Those kind of computations are conceptually straightforward but the input use to be large and should be distributed across a large amount of machines if we want it to be finished in a reasonable time. Inspired by LISP primitives, Google decided to abstract the problem and define *map* and *reduce* primitives.

The *map* consists in splitting the tasks in small pieces of work that are sent to the computing machines and the *reduce* consists in collecting all the results from the *map* step and, if it is necessary, process the results to obtain the desired output.

While MapReduce isn't something entirely new it helped to standardize parallel applications. With a simple approach, this model proved to be powerful enough to compute large amounts of data in a reasonable time, due to its high scalability. This high scalability comes from the basics of dividing the job in small tasks that can be split through a large collection of computers.

The data is commonly stored in a distributed file system, but for improving the scalability the master tries to distribute the tasks on the the nodes having the input data, speeding up the task because the data is available locally. In addition, if a node fails on finishing it had assigned, the master node can always send the failed task to another available node.

### 2.2.1. Use cases

MapReduce is useful in a wide range of applications and architectures, but specially well suited to solve problems without dependencies or communication requirements, that can be easily parallelized achieving a proportional speed up to the size of the problem. I present four examples where MapReduce is being used nowadays.

#### 2.2.1.1. Word frequency counting

The more common example to explain how does MapReduce work is the implementation of a program that count the frequency of words in a very large collection of documents. This application is the “Hello world!” for MapReduce and was the one used by the creators in the original paper.

To compute the frequency of the words a sequential program would need to read all the documents
and elaborate a list of <word,frequency> pairs, incrementing the frequency value every time a word is found.

For MapReduce the approach is different. The documents would be split through the cluster nodes and the wordcounting would be done in each node independently. This first part is the map. Once the map phase is done the results are merged in what is called the reduce phase, and the final output is ready to be shown.

Even if MapReduce would be a bit overkill for a problem like word-counting, is very good to show and understand the two phases of this programming model.

### 2.2.1.2. Distributed search and sort

Searching and sorting are other common examples to describe the MapReduce model. As wordcount those are helper tools that can be integrated into larger environments, as their counterparts un UNIX: wc, grep, sort, etc.

A distributed version for grep is very easy to implement due that it only requires the map phase. The application only has to read line by line and show the line if it matches the given pattern.

For a sorting problem both phases are required, map to sort locally and reduce to merge all the results.

### 2.2.1.3. Log analysis

With the large amount of log information stored in servers everyday, treating this information became an important part of the business intelligence for almost all the companies. Companies such as Facebook use MapReduce to examine log files daily to generate statistics and on-demand analysis.

### 2.2.1.4. Search engines

Keeping indexes up to date is one of the top priorities for internet search engines, but internet is growing up everyday with new web pages and new uploads. Keeping indexes up to date became a massive work that requires very high scalable systems. MapReduce was designed with this purpose and since its creation Google used it to maintain their indexes. Yahoo! Is another big search engine using MapReduce and is promoting one of the most widely open source implementations, called Hadoop.

### 2.2.2. MapReduce implementations

Several implementations of this model appeared after the the first one created by Google, some as open source and others with commercial purposes. Here we can see a few ones:

#### 2.2.2.1. Google MapReduce

Google implementation was the first and the one who gave the name to the programming model. It was developed by Jeff Dean and Sanjay Ghemawat in 2004 and its idea is open and known by everyone but its code remains in Google and therefore is only used at Google. Even though is known it offers support for Python and Java. Up to date is in some of the largest MapReduce clusters according to the data revealed by Jeff Dean: “It ran 29.000 jobs in August 2004 and 2.2 million in September 2007. Over that period, the average time to complete a job has dropped from 634 seconds to 395 seconds, while the output of MapReduce tasks has risen from 193 terabytes to 14.018 terabytes. On any given day, Google runs about 100.000 MapReduce jobs; each one
occupies about 400 servers and takes about 5 to 10 minutes to finish”. In 2008 Google reported that their MapReduce implementation was able to sort 1 terabyte of data on 1000 computers in 68 seconds [16].

2.2.2.2. Hadoop

Hadoop is a popular open source implementation with a large community base and is also used by big companies as Yahoo!, Amazon, Facebook or IBM, and provides support for Java. Nowadays is a top-level Apache project and hosts sub-projects such as HDFS, HBase, Pig, etc. It was originally created in 2005 and in 2008 Yahoo! announced that they were using a 10,000 core Hadoop cluster to generate their search index. In 2009 Yahoo! reported that they were able to sort 1 terabyte of data in 62 seconds using a cluster of 1460 nodes running Hadoop.

2.2.2.3. Disco

Disco is the private Nokia Research Center implementation for MapReduce and typically runs programs written in Python. Unlike Hadoop it doesn't include a customized file system, but it supports POSIX-compatible distributed file systems.

2.2.2.4. Other

Other implementations are: GridGain, Greenplum, Skynet or Dryad.
2.3. Hadoop

Hadoop has emerged as one of the most popular implementations for MapReduce in the research community as it is open source and is part of the Apache Software Foundation. But not only academics are taking part of the Hadoop project, big companies like Amazon, Yahoo!, Adobe, Facebook, Hulu, IBM, Last.fm, etc. are using and contributing on it [5].

This large contribution helped Hadoop to become the most complete open source implementation for MapReduce, even capable to compete with the original Google MapReduce judging from the results of the Terasort benchmark published by Yahoo! On April 2009 [17].

2.3.1. Project and sub-projects

Hadoop started with its MapReduce implementation but grew up very fast and nowadays includes other projects to provide the required infrastructure, such as HDFS (Hadoop Distributed File System) that provides the distributed file system required for a MapReduce implementation, becoming in this way a complete software solution. Here is a list of the most important subprojects that are part of Hadoop:

- **Common**: The common utilities that support the other Hadoop subprojects (RPC, serialization, configuration, etc.).
- **Avro**: A data serialization system that provides dynamic integration with scripting languages.
- **Chukwa**: A data collection system for managing large distributed systems.
- **Hbase**: A scalable, distributed database that supports structured data storage for large tables.
- **HDFS**: A distributed file system that provides high throughput access to application data.
- **Hive**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **MapReduce**: A software framework for distributed processing of large data sets on compute clusters.
- **Pig**: A high-level data-flow language and execution framework for parallel computation. Programs written in high-level language are translated to MapReduce programs.
- **Zookeeper**: A high-performance coordination service for distributed applications.

2.3.2. Cluster overview

Hadoop can be deployed on a single machine, but is designed to be deployed in large clusters having competitive results with clusters of thousands of machines.

Typically Hadoop clusters have one node acting as master, *jobTracker* in Hadoop terminology, that is the one receiving the requests from the user, splitting the task in several parts and schedule the work, and several nodes called *taskTrackers* that are in charge of the processing of the small portions of the original task.

In Hadoop terminology the user requests are known as *jobs*, the *jobs* include the input data as well as the *map* and *reduce* functions and its configuration. When a *job* is submitted to the *jobTracker*, it splits the *job* in a number of *splits* and send the *splits* to the *taskTrackers*. Commonly each *taskTracker* can run more than one *split* at a time, the amount of *splits* assigned to a *taskTracker* is
known as *slots*.

Before assigning the task to the *taskTrackers*, the *jobTracker* divides the input data depending on its format. Then the *jobTracker* prepares as many *map tasks* as splits and as soon as a *taskTracker* reports a free *map slot*, it assign one of the there one of the *map tasks* waiting in the queue.

![JobTracker diagram]

An important task for the *jobTracker* is to keep track of all the tasks among the cluster, due that it has to control when all the *map tasks* are finished to start the *reduce tasks*. This tracking helps the *jobTracker* to keep the system fault-tolerant and if a node fails or times out, the task assigned to this node can be reassigned to different node.

This centralized control in the *jobTracker* represents a potential point of failure that should be taken into account.

### 2.3.3. HDFS

As mentioned before, Hadoop includes its own distributed file system called HDFS, that stands for Hadoop Distributed File System. Moving data through the system to process it is more expensive, in terms of time, than moving computation to the data, and that's why MapReduce follows this strategy. This means that instead of using another independent dedicated storage, the data is stored in the same low-cost machines used for the computation.

As happened with the MapReduce implementation, HDFS is inspired in another Google system, this time the Google File System [18]. It shares many features with other file systems but it is specially designed to be deployed on commodity hardware.

MapReduce is based on a client-server architecture and the same happens to HDFS. It consists on a single master node called *nameNode* and serveral slaves called *dataNodes*. The *nameNode* is in charge of all the meta-information of the file system and coordinates it. The *dataNodes* are spread among the cluster and are responsible of storing the data.

The terminology for defining the portions of memory space is, as it happens in many file systems, *blocks*. As HDFS is designed to read and write large files, *block* size is large, and normally defaulted to 64MB, but is completely customizable.

In order to ensure fault-tolerance, replication of the data is carried out by the *nameNode* through the *dataNodes*. This process is completely transparent to the user, due that the user asks the *nameNode*...
to upload data and is this one who takes care of the replication. When one of the `dataNodes` fails the `nameNode` goes through the nodes looking for space to recover the level of replication and copying the lost files from the remaining copies of the data stored in the failed node to a new one. As happens with the `block` size, the replication is completely customizable, but is defaulted to 3 because statistically is good level of replication to avoid loosing data.

### 2.3.4. Dataflow

Up to now we know that Hadoop is based in two components, the MapReduce computation and the distributed file system, and we saw how does those two components work individually. To better understand the hole platform is interesting to have a look on the dataflow, from the input until the final output.

Hadoop can work with several distributed file systems, but lets assume, for this section and the following in the project, that the file system is the HDFS, because is the most common situation and that's why I described it above.

The first step is the introduction of the input data on the file system. MapReduce is waiting for the data to be distributed and available to start the computation process. Once the data is available in the `dataNodes`, the `jobTracker` creates map and reduce tasks. The number of map tasks is usually determined by the size of the input data. Splitting the data is what makes the system scalable and
can have a great impact on the performance. As an example, if the input consists on a single 64GB file in the HDFS, with the default block size of 64MB, the job would split into 1000 map tasks.

In order to speed up the realization of the map tasks, the jobTracker tries to send the tasks to the dataNodes containing the split of data that has to process, but if it is not possible data can be read remotely. While processing the data input, partial output is written to a circular memory buffer. If the buffer reaches certain threshold, the partial output is stored in a temporary file. If there is more than a temporary file the map task will merge it and store the merged result in disk. At the end of a map task all the information is stored in disk, due to the goal of finishing as soon as possible, and the space in the memory buffer is served to other tasks.

The number of reduce tasks is not automatically calculated but manually defined by the user depending on the job needs. One reduce task can be enough for a sorting problem but maybe not for other problems. Unfortunately, the more reduce tasks requires the problem, the more increases the overhead of the framework.

Reduce tasks are comprised of three phases: copy, sort and reduce. Reduce phase has to wait to map phase to be finished, but it is possible to run the first of the reduce phases while the partial results of the map phase arrive. As soon as the sort and copy phases are done, the data is passed to the reduce() function and its output is written to the HDFS.

2.3.5. Hadoop internal scheduler

As part of the MapReduce implementation, Hadoop has an internal scheduler for managing the incoming requests. At the beginning Hadoop was designed to run batch jobs, so its default scheduler is very simple, basically it is a FIFO scheduler that run jobs in the order of submission and supports basic priorities. Only one job run at a time, but it takes advantage of the separate map and reduce phases to do some overlapping.

Hadoop 0.19 shipped with a pluggable scheduler interface that was decoupled from the jobTracker, and allowed the creation of alternative and customized schedulers. Since then there have been many proposals, most of them focused on multi-user and multi-job scenarios. Those new schedulers are very powerful and help to speed up the execution. In addition they can provide information of the evolution of the jobs to higher-level schedulers, and improve the resource management.
2.4. Virtualization

Virtualization is a proven software technology that is rapidly transforming the IT landscape and fundamentally changing the way that people compute. Today’s powerful x86 computer hardware was designed to run a single operating system and a single application. This leaves most machines vastly underutilized. Virtualization lets you run multiple virtual machines on a single physical machine, sharing the resources of that single computer across multiple environments. Different virtual machines can run different operating systems and multiple applications on the same physical computer. It is achieved through a virtual machine monitor (VMM). VMMs are also known as hypervisors.

![Diagram of virtualization](image)

VMMs are responsible for keeping track of all activities performed by virtual machines. VMMs are categorized into **paravirtualization** and **full virtualization** VMMs. **Paravirtualization** VMMs run directly on hardware without the need for a host OS, while **full virtualization** VMMs run on top of a host OS. Server virtualization can be achieved using **full virtualization** or **paravirtualization** VMMs.

2.4.1. Full virtualization

Full virtualization is achieved using direct execution of user application code and binary translation of OS requests. The hypervisor traps and translates all OS instructions and caches the results for future use.

2.4.2. Paravirtualization

With paravirtualization, the VMM builder defines the virtual machine interface by replacing non-virtualizable portions of the original instruction set with easily virtualized and more efficient equivalents, this makes paravirtualization more efficient than full virtualization, because it does not need to binary translate all the system calls. Although operating systems must be ported to run in a virtual machine, most normal applications run unmodified. The problem is that paravirtualization doesn't support unmodified OS like Microsoft Windows. Xen is pioneer in paravirtualization and the most popular among academics.
2.4.3. Virtualization platforms

Let's review which virtualization platforms can we find nowadays:

- **Xen**: is open source virtualization software that provides paravirtualization and supports x86 and x64 processors. It allows several guest operating systems to execute on one physical machine simultaneously. The first guest OS is known as Domain-0 in Xen terminology. Domain-0 automatically boots whenever Xen software boots. Users need to login on Domain-0 to execute other guest OS. Although Xen is best known in the Linux world, Xen 3.0 introduced support for running Windows virtual machines.

- **VMWare**: is commercial virtualization software that provides full virtualization. VMWare has many flavors such as VMWare Workstation, VMWare Server and VMWare ESX that provide different levels of flexibility and functionality. VMWare is highly portable as it is independent of the underlying physical hardware, making it possible to create one instance of a guest OS using VMWare and copy it to many physical systems.

- **KVM**: The Kernel-based Virtual Machine (KVM) is a free virtualization platform providing full virtualization. KVM contains a Type-2 hypervisor and supports Linux-based host operating systems. It allows Windows or Linux as the guest OS.

- **VirtualBox**: a newcomer to the ranks of virtualization market, Sun Microsystems Virtual Box is a software package that provides paravirtualization. It was initially developed by a German company, Innotek, but now it is under the control of Sun Microsystems as part of the Sun xVM virtualization platforms. It supports Linux, Mac, Solaris, and Windows platforms as the host OS.
2.5. EMOTIVE Cloud

EMOTIVE Cloud is an ongoing research project in the Barcelona Supercomputing Centre (BSC). Its aim is to create a platform that enables the final user to take the maximum benefit of virtualization capabilities with minimal effort.

EMOTIVE (Elastic Management of Tasks in Virtualized Environments) is a middleware used to provide virtual machines in a very easy way, where the user can run common applications, such as Tomcat, with a user-friendly GUI.

The platform is still in a developing phase but it already performs interesting features such as virtual machine migration, checkpointing and customization for the virtual machines at the creation time. Making the user capable of defining virtual machine features, such as the amount of memory space or the number of CPUs required, is what makes it a very flexible platform and a different alternative to other more extended virtualization supports like Amazon EC2.

2.5.1. Architecture

EMOTIVE architecture is divided in three layers:

- data infrastructure.
- node management.
- global scheduler.

The data infrastructure is based in a distributed file system (DFS) that makes the data available through all the machines in the network. The DFS is what makes virtualization capabilities such as migration and checkpoint possible. In addition, the DFS caches virtual machines in each physical machine to make virtual machine creation process faster.

The node management layer, called VRMM, is in charge of creating and maintaining the whole virtual machine lifecycle. This layer can be the more complex layer of the system due to the amount of features it manages, but it can be considered the 'kernel' of the system.

The VRMM is composed by the resource monitor (RM) and the virtualization manager (VtM) in each machine. The resource monitor component the one in charge of keeping up to date the system topology, monitoring the physical machines performance. The virtualization manager is the component interacting with the virtualization support.

When EMOTIVE Cloud started the virtualization was carried out using Xen, but as the project evolved it was extended to support KVM virtualization and now it also offer an interface to Amazone EC2. Although now it gives support for three different virtualization technologies, we
assume for this and the following sections that the virtualization is done with Xen, due that was the only technology available at the beginning of this project and the most complete solution.

Saying that the virtualization manager is the one interacting with Xen means that this component is in charge of the creation, running, migration, checkpointing, recovering, stopping and deleting of the machines. To speed up the creation process predefined virtual machine images with specified software can be stored in the system, this means that if we want a virtual machine to run a Tomcat server, it is not necessary to create the machine and the download and install the software, the machine can have it already installed. This feature makes life easier for the user and the service much more attractive.

Finally, the scheduling layer is in charge of distributing tasks and VMs among the physical nodes. The scheduler is defined as a web service that catches the user requests and distribute the resources to complete the task as soon as possible. The original scheduler defined in EMOTIVE is a first come first served (FIFO) scheduler, but it gives the possibility of adding new schedulers to run over the VRMM layer, in order to improve the balancing and management of the resources to specific purposes. There already exists some specific purpose oriented schedulers for EMOTIVE such as SERA, a semantically enhanced resource allocator, or EERM, an economically enhanced resource manager.

The schedulers designed for EMOTIVE are RESTful web services. REST(Representational State Transfer) principles encourages applications to be simple, lightweight, and have high performance. Companies like Amazon, Microsoft, eBay, Yahoo! Saw that and are offering implementations for REST, and EMOTIVE couldn't do anything else but to implement RESTful webservice as well, using the Jersey API from Java.

With this structure EMOTIVE can offer fast creation of virtualized environments, making those safe and elastic to manipulations, and gives the possibility of adding new schedulers to manage the incoming jobs, becoming in this way a powerful tool for service providers.
2.6. REST

REST (Representational State Transfer) is an architectural style for distributed systems that specifies a set of constraints, emphasizing scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

REST required constraints:

• Client-server: there is a separation between clients and server made by a uniform interface, so the client does not concern about, for example, data storage.

• Stateless: there is no client context stored on the server between requests, so any request from the clients contains all the information needed to carry out the request.

• Cacheable: responses must be cacheable in the server.

• Layered system: the client cannot know if is directly connected to a server or not.

• Uniform interface: a unified interface between client and server simplifies and decouples the architecture and enables each part to evolve independently.

In the REST architectural style, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple operations. The services are designed to use stateless communications, typically using HTTP requests based on the four well-defined operations: GET, PUT, POST, and DELETE.

In the REST-style, clients and servers exchange representations of resources using a standardized interface and protocol. Thus, an application can interact with a resource by knowing only two
the action required and the identifier of the resource.

The application does, however, need to understand the format of the information (representation) given and returned back from the request, which is typically an HTML or XML, although it may be an image, plain text, or any other content.

These principles and definitions encourage REST-style applications to be simple, lightweight, and have high performance.

2.6.1. RESTful web services

*RESTful web services* are services that are built to work best on the web. There are many framework implementations for the REST architectural style:

- OpenRasta, for .NET developers.
- Jersey, Wink, Restlet, etc. for Java.
- Sinatra for Ruby.
- Catalyst REST for Perl.
- and many more.

As far as EMOTIVE is developed in Java, the RESTful web services created in EMOTIVE used a framework for Java, in this case, Jersey.

In a nutshell, Jersey is an open source framework for developing RESTful web services, and is the reference implementation of the JAX-RS API.

JAX-RS provides a standardized API for building RESTful web services in Java. The API basically provides a set of annotations and associated classes and interfaces. Applying the annotations to Plain Old Java Objects (POJOs) enables you to expose web resources.

The annotations includes:

- `@Path`, specifies the relative path for a resource class.
- `@GET`, `@PUT`, `@POST`, `@DELETE`, specifies the HTTP request type of a resource method.
• @Produces, specifies the returned MIME media types.
• @Consumes, specifies the acceptable request media types.
• @PathParam, @QueryParam, @HeaderParam, @CookieParam, @MatrixParam, @FormParam, specifies the source of the method parameter values, e.g. @PathParam comes from URL path, @QueryParam comes from URL query parameter, @HeaderParam comes from HTTP header and @CookieParam comes from Cookie in HTTP request.

Web resources in REST are structured like directories and to get them is necessary specifying the correct path to reach them. For example, in a bookmark application we will have users, and for each user we will have bookmarks. For this case we can have an structure like the following:

• UsersResource: Represents a list of users
• UserResource: Represents a specific user
• BookmarksResource: Represents a list of bookmarks for a specific user
• BookmarkResource: Represents a specific bookmark

With this structure, we will have the following addresses:

• UsersResource → /users
• UserResource → /users/{userid}
• BookmarksResource → /users/{userid}/bookmarks
• BookmarkResource → /users/{userid}/bookmarks/{bmid}

At this point REST can seem a bit heavy and difficult to understand, but the simplest example will clarify that RESTful web services are easy to design, implement and deploy.

2.6.2.1. Example

When we want to create a web service with Jersey, the start point is creating a project and having all the dependencies. This can be very different if we use Maven, Ant or we put the dependencies by hand. Assuming we have the dependencies, next step is creating a root resource. This time, as we want something very simple, lets use the “Hello world” example to explain this:

```java
/**
 * @Path("/helloworld")
 */
public class HelloWorldResource {

@GET
@Produces("text/plain")
public String sayHello(@QueryParam("world") String world) {
    return "Hello " + world;
}
}
```

The interesting bits are four annotations:
1. @Path("/helloworld") – this annotation exposes methods in this class through helloworld path. Annotation value is the thing that we will add after resources in URI to call this service
2. @GET – annotated method (sayHello) is available through HTTP GET method
3. @Produces("text/plain") – method will return plain text
4. `@QueryParam("world")` – with this annotation we are mapping URI parameter world to method parameter world

Next step is adding the Servlet configuration and mapping. For that is necessary to copy this in `web.xml`.

```xml
01.<servlet>
02. <description>Jersey Servlet</description>
03. <display-name>ServletContainer</display-name>
04. <servlet-name>ServletContainer</servlet-name>
05. <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
06.</servlet>
07.<servlet-mapping>
08. <servlet-name>ServletContainer</servlet-name>
09. <url-pattern>/resources/*</url-pattern>
10.</servlet-mapping>
```

With this in place we have configured Jersey servlet. It will serve REST services with pattern resources in URI.

After that we have to deploy the project on the server and that’s it. Now we can test service using browser. If we enter following URI: `http://localhost:9080/HelloWorldRest/resources/helloworld?world=World!` we should get response in clear text, as expected: `Hello World!`.

With this simple example we have a RESTful web service saying “Hello world!”

Creating an stand alone client for this service is an easy task:

1. Created a new Java project called `HelloWorldRestJava`.
2. Created new folder called `lib`.
3. Copied jars from `WEB-INF/lib` to `lib` folder. Added them to project classpath.
4. Created new class with main method:

```java
01.package org.example;
02.
03.import com.sun.jersey.api.client.Client;
04.import com.sun.jersey.api.client.WebResource;
05.
06.public class HelloWorldRestClient {
07. 
08. public static void main(String[] args) { 
09.  Client client = new Client();
11.  String response = webResource.path("resources").path("helloworld")
12.    .queryParam("world", "World!").get(String.class);
13.  System.out.println("Response: "+ response);
14. }
15.}
```

5. Run the client as Java application.

This simple client can get information from the RESTful web service just knowing the route to the resource and the parameters required for the action.

But, how does this annotation work? It's black magic, right? Yes, sort of, but it's easier to understand than you might think.

The secret lies in the fact that the coded method isn't called directly. What's called is a "hidden" Jersey controller method that uses reflection to examine which of the coded methods are appropriate for the (GET, PUT, POST, DELETE, etc.) call, then chooses which one based on the expected MIME type and performs other work on context, path, etc. Then it calls the method which just works.
3. Requirements

The following section provides a list of the main requirements for the new part of the system. Those requirements are divided between functional and non-functional requirements. Functional requirements are all those requirements related with the functionalities and the behaviour the system should provide. On the other hand, non-functional requirements cover the qualities the system should provide.

3.1. Functional requirements

Here is shown a list with the functional requirements for the system. In order to don't make redundant documentation the description of the uses cases where those requirements come from is done in the specification chapter.

Functional requirements:

- automated creation of a Hadoop environment
- addition of VM's to an existing Hadoop environment
- deletion of VM's from an existing Hadoop environment
- deletion of an entire Hadoop environment
- variability in the number of machines created in the setting up process.
- automated load balancing for physical machines
- define the environment name
- check redundancy on environment names
- provide an interface for managing the features
- provide an interface to interact with the Hadoop scheduler

3.2. Non-functional requirements

This section describes de non-functional requirements of the system and then a closer description of the requirements in shown, with a description, justification and satisfaction condition of each one.

The non-functional requirements are:

1. web-oriented GUI
2. develop the GUI with ZK
3. develop the system in Java
4. virtualization over Xen 3.0
5. run over apache tomcat
6. intuitive navigation
7. coherent information
8. clear information
9. error information
10. extensible
11. scalable
12. service always available
13. Run over i386 and x64.
14. Secure machines interaction
## 1. Web-oriented GUI

<table>
<thead>
<tr>
<th>Description</th>
<th>The user interface should be designed and implemented to be accessible through a web browser.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Justification</td>
<td>EMOTIVE graphical user interface is already implemented to be accessible through a web browser and I am creating an extension of for EMOTIVE.</td>
</tr>
<tr>
<td>Satisfaction condition</td>
<td>The GUI is accessible through a web browser.</td>
</tr>
</tbody>
</table>

## 2. Develop the GUI with ZK

<table>
<thead>
<tr>
<th>Description</th>
<th>The GUI should be implemented with ZK technologies.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Justification</td>
<td>EMOTIVE graphical user interface is already implemented with ZK technologies and I am creating an extension of for EMOTIVE.</td>
</tr>
<tr>
<td>Satisfaction condition</td>
<td>The GUI is implemented with ZK technologies.</td>
</tr>
</tbody>
</table>
### 3. Develop the system in Java

**Description**
The piece of software should be implemented with JAVA technologies.

**Justification**
The other components of EMOTIVE are implemented with JAVA technologies.

**Satisfaction condition**
The system is implemented with JAVA technologies.

### 4. Virtualization over Xen

**Description**
The virtualization of Hadoop should be carried out by Xen.

**Justification**
Xen is the only virtualization technology available in EMOTIVE at the beginning of this project.

**Satisfaction condition**
Virtualization is carried out by Xen.
### 5. Run over Apache Tomcat

**Description**
The system should run over Apache Tomcat.

**Justification**
EMOTIVE is already running over Apache Tomcat.

**Satisfaction condition**
The system runs over Apache Tomcat.

### 6. Intuitive navigation

**Description**
The navigation should be intuitive and pleasant for the user.

**Justification**
User satisfaction is basic for the success of all projects.

**Satisfaction condition**
The navigation is easy and pleasant.
7. Coherent information

**Description**
The information shown should be coherent with the system state.

**Justification**
Inconsistent information is not acceptable.

**Satisfaction condition**
The information shown is completely coherent with the system state.

---

8. Clear information

**Description**
The information shown should be clear.

**Justification**
User satisfaction is basic for the success of all projects and keeping him properly informed is the first step to achieve it.

**Satisfaction condition**
The information shown is clear.
<table>
<thead>
<tr>
<th>9. Error information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Justification</strong></td>
</tr>
<tr>
<td><strong>Satisfaction condition</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10. Extensible</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Justification</strong></td>
</tr>
<tr>
<td><strong>Satisfaction condition</strong></td>
</tr>
</tbody>
</table>
11. Scalable

<table>
<thead>
<tr>
<th>Description</th>
<th>The system should provide information when an error occurs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Justification</td>
<td>EMOTIVE aims to be a large system with several levels of abstraction and complex software. Information of the errors if very important to know what's wrong in the system.</td>
</tr>
<tr>
<td>Satisfaction condition</td>
<td>Information is provided when the errors appear.</td>
</tr>
</tbody>
</table>

12. Availability

<table>
<thead>
<tr>
<th>Description</th>
<th>The system should be prepared to be always available.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Justification</td>
<td>A service provider can not afford have the system online just some hours of the day or some specific days to succeed.</td>
</tr>
<tr>
<td>Satisfaction condition</td>
<td>The system is prepared to be always available.</td>
</tr>
<tr>
<td>13. Run over i386 and x64</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>The system should run over i386 and x64 architectures.</td>
<td></td>
</tr>
<tr>
<td><strong>Justification</strong></td>
<td></td>
</tr>
<tr>
<td>We don't know the architecture available on the physical machines.</td>
<td></td>
</tr>
<tr>
<td><strong>Satisfaction condition</strong></td>
<td></td>
</tr>
<tr>
<td>The system can run over x386 and x64 architectures.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. Secure machine interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>The interaction between virtual machines should be secure.</td>
</tr>
<tr>
<td><strong>Justification</strong></td>
</tr>
<tr>
<td>No manipulations on the traffic would be admitted.</td>
</tr>
<tr>
<td><strong>Satisfaction condition</strong></td>
</tr>
<tr>
<td>The machine interaction is secure.</td>
</tr>
</tbody>
</table>
4. Specification

4.1. System components

The main goal of this project is the virtualization of Hadoop, and to achieve it, I used the three layer composed EMOTIVE platform.

The piece of software designed for taking care of the virtual machine clusters is an extension of the predefined scheduler in EMOTIVE, in the virtualization layer. This virtualization layer (VRMM) is the layer containing the virtual machines and the virtualization manager (VtM) is the component that manages the virtualization, being the scheduler the component that receives the user requests.

The predefined scheduler existing in EMOTIVE is a simple FIFO scheduler that manages requests and is made thinking to ease possible extensions. For this project I have created a new scheduler extending the capabilities of the scheduler, adding then new features that allow the creation and manipulation of Hadoop clusters.

I called this scheduler Hadoop Scheduler and is basically a RESTful webservice [19]. RESTful webservice are designed to work best on the web. They are designed to use stateless communications, typically using HTTP requests based on the four well-defined operations: GET, PUT, POST, and DELETE. In the REST architectural style data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs). In the REST architecture style, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourage RESTful applications to be lightweight, simple, and have high performance.

The scheduler is going to receive requests from the client for the creation and manipulation of the clusters, and from the internal scheduler integrated inside Hadoop, that is in charge of managing the jobs, and manages the resources the cluster has in order to achieve a goal in the performance of the jobs, for example, trying to finish two jobs at the same time, or finishing one job in a certain time.

This internal scheduler in Hadoop is going to interact with the scheduler defined in this project for providing information about the performance carried about with the job running, and the requirement or not of more virtual machines to finish the jobs according to its goal. This information is going to be more specified in the design, but will be provided in terms of time and resources: predicted time for finishing the job according with the current resources, goal time when the job should be done, current number of resources and required number of resources to finish the jobs in the required time.

The scheduler designed in this project has to take care of the cluster manipulation, not only by defining the operations that affecting its state, but also ensuring the integrity of the clusters. To be aware of the integrity of the clusters the scheduler requires a way to keep all the information about the cluster (nodes available, roles of the nodes) and which clusters does is has running, In this way, the scheduler will always know if it can add or remove nodes from the cluster keeping the integrity of the data and the processes running on it.
By definition, EMOTIVE scheduler provides one virtual machine each time it receives a creation request. This means that it doesn't provide a way for creating clusters of virtual machines automatically, and that's what Hadoop Scheduler has to take care of.

There are two possible approaches to virtualize Hadoop using EMOTIVE. The first one is creating a new virtual machine and then upload all the software and configuration necessary to run Hadoop. The second one is create a predefined virtual machine image with a distribution of Hadoop already installed and configured. While the second option can take just a few seconds, the first one is much more inefficient but at the same time it doesn't require maintenance in the virtual machine, for example, if a new version of Hadoop appears. Even knowing is going to carry more maintenance work, I chose the second option due to the size of Hadoop and the configuration requirements, around 38MB that in the other case have to be uploaded to the virtual machine and installed, and as far as the machines used are commodity machines, this upload and installation task can take several minutes to be carried out in each virtual machine, in addition to the time required to create the virtual machine.

4.2. Use cases

In this section I describe the use cases treated in this project. For each use case a short description and interaction diagrams are shown.

There are two actors interacting with the system: user and Hadoop job scheduler. The user is the final user that interacts with the system through the GUI. Commonly is going to be someone knowing about the system and the features it provides. The Hadoop job scheduler is the internal scheduler that Hadoop provides to manage the incoming jobs and is going to interact with the Hadoop Scheduler to provide information about the progress of the job it is managing.
There are six use cases for the user: create cluster, add node to cluster, remove node from cluster, delete cluster, check the state of the HDFS and check the state of the MapReduce service. The communication between the Hadoop Scheduler and the job scheduler is only in one way and it only has one use case: update the cluster status.
Create cluster

Operation: create cluster

Semantics: the user orders the creation of a cluster with a certain amount of slave nodes, specifying the name of the master node that will become the name of the cluster, the replication factor that want to define in Hadoop, the percentage of CPU that the virtual machine requires and the memory of the virtual machine.

Pre: doesn't exists a cluster with the specified name.

Post: a cluster is created with the specified name and amount of nodes, the replication factor configured, and all the machines with the same cpu and memory.

Add node to cluster

Operation: add node to cluster

Semantics: the user requests creating and adding a new node to an already existing cluster.

Pre: -

Post: a node is created and added to the specified cluster.
Remove node from cluster

**Operation:** delete node from cluster

**Semantics:** the user requests removing a node from an specific cluster.

**Pre:** the cluster exists.

**Post:** if the cardinality of the slaves of the indicated cluster is bigger than the replication factor, the node is removed from the cluster.

---

Delete cluster

**Operation:** delete cluster

**Semantics:** the user requests the deletion of a specific Hadoop cluster.

**Pre:** the cluster exists.

**Post:** a cluster is removed.
Check MapReduce

**Operation:** check MapReduce

**Semantics:** the user requests the system to show the state of the MapReduce tasks from an specific cluster.

**Pre:** a cluster exists.

**Post:** -

**Output:** a new window in opened showing the graphical user interface available in Hadoop to show the MapReduce tasks.

---

Check HDFS

**Operation:** check HDFS

**Semantics:** the user requests the system to show the state of the HDFS available in a specific cluster.

**Pre:** a cluster exists.

**Post:** -

**Output:** a new window in opened showing the graphical user interface available in Hadoop to show the HDFS.
Update cluster state

Operation: update cluster state

Semantics: the internal job scheduler of Hadoop sends information about the progress of the jobs running in a cluster. The information sent contains the prediction, in time, for finishing the job according with the current resources, goal time when the job should be done, the current number of resources and the required number of resources for finishing the jobs in the required time.

Pre: there exists a cluster and the job scheduler is running.

Post: the prediction is sent and stored in the system.
5. Design and implementation

In this section I describe how have the different components of the system been designed, from the more lower level with the virtual machine to all the parts of the Hadoop Service Provider (HSP). Note that the order in which are reported the components is important and has been the order of development during the project.

The virtual machine design comes first, integrated with the default scheduler. After a number of iterations to polish the virtual machine and its automatic creation, the Hadoop Scheduler was developed.

5.1. Virtual machine

The virtual machine creation and configuration process should be done as quick as possible, that's why I chose to create a predefined virtual machine with Hadoop installed and configured.

For the firsts iterations I used the other model, which means download, install and configure Hadoop on a virtual machine every time a new virtual machine is created, because it was easy and didn't require any extension for EMOTIVE. It was useful to use this model model at the beginning to ensure the viability of the project. Although both approaches were used in this project, in this section I just describe the final configuration of the predefined virtual machine, due to the similarities between the configuration procedures.

Once decided to create a predefined virtual machine in EMOTIVE, the procedure is create a normal virtual machine, download and install a version of Hadoop including the internal scheduler recompiled for using the client of EMOTIVE Hadoop service, and save the image in the repository of predefined images, called extensions. That's easy due that Hadoop doesn't have so many requirements to be deployed, only the JRE 1.6 should be included in addition to Hadoop and the existing software. But EMOTIVE doesn't provide support to deploy more than one virtual machine at once and we want to have only one predefined type of virtual machine for Hadoop, independently if the machine is going to be the master or a slave in the cluster. This forces the creation of a script that configures Hadoop at the creation time depending on what kind of machine we want each time we call the service.

The configuration in Hadoop is devided between the following files: hadoop-env.sh, core-site.xml, hdfs-site.xml, mapred-site.xml, masters, slaves, hosts and hosts.exclude. All those files are inside the 'conf' directory of Hadoop.

Hadoop provides a predefined configuration for all its configurable parameters in: core-default.xml, hdfs-default.xml and mapred-default.xml. For changing the default value of a concrete parameter in the configuration, we should define this parameter in the corresponding file: core, hdfs or mapred-site.xml, with the value we want for it, before the cluster set up. The file hadoop-env.sh is used to configure the environmental variables. For this project we need to specify the $JAVA_HOME variable, $HADOOP_HOME saving the path for reaching the instalation of Hadoop in the machine, and the $HADOOP_CLASSPATH for the connection with the internal scheduler of Hadoop, due that the scheduler used is not the default one, is a contribution and is defined in the '/contrib' directory. Due to the amount of configurable parameters I just show the modified ones appearing in core, hdfs and mapred-site.xml respectively. Those parameters are modified to improve the testing phase in the project, to make it faster and easier.

```xml
<property>
    <name>fs.default.name</name>
```
hdfs-site.xml

<property>
  <name>dfs.replication</name>
  <value>2</value>
</property>

<property>
  <name>dfs.namenode.decommission.interval</name>
  <value>5</value>
  <description>complete.</description>
</property>

<property>
  <name>dfs.blockreport.intervalMsec</name>
  <value>500</value>
</property>

<property>
  <name>dfs.blocksize</name>
  <value>4194304</value>
</property>

<property>
  <name>dfs.permissions.enabled</name>
  <value>false</value>
</property>

<property>
  <name>dfs.http.address</name>
  <value>nameNodeIp:50770</value>
</property>

<property>
  <name>dfs.secondary.http.address</name>
  <value>secondaryNameNodeIp:50790</value>
</property>

<property>
  <name>dfs.https.address</name>
  <value>nameNodeIp:50470</value>
</property>

<property>
  <name>dfs.datanode.address</name>
  <value>dataNodeIp:50710</value>
</property>

<property>
  <name>dfs.datanode.http.address</name>
  <value>dataNodeIp:50775</value>
</property>

<property>
  <name>dfs.datanode.ipc.address</name>
  <value>dataNodeIp:50720</value>
</property>
In these configuration files, the values of nameNodeIp, secondaryNameNodeIp, dataNodeIp, jobTrackerIp and taskTrackerIp will be replaced by the real IP addresses of the corresponding
virtual machines at the installation time by 'hadoop-starter' script.

Hadoop provides two files (masters and slaves) to define the machines members of the cluster and its role in it when the system switches on, and two other files, optional, to define which machines in the cluster are allowed to take part of the computation and which are not allowed (hosts and hosts.exclude). All the machines are identified by the IP.

In this project the IP assignation to the new virtual machines is done automatically by a name-server (DHCP), this means that the 'masters' and 'slaves' files are not required and remain in blanc, because Hadoop only use those files at the beginning. For the 'hosts' and 'hosts.exclude' files, the modifications are done in running time, as modifications in the cluster topology occur, so if a node is removed from the cluster, the IP will be written in the hosts.exclude file. This is important when there are few machines in the cluster, because when the machine is removed from the cluster Hadoop starts what they call decommission process, that basically consists in ensuring that the data stored in the node is replicated among the other nodes as many times as is specified in the configuration files.

In this section we spoke about one script that is in charge of configuring the machine when this one is created, configuring Hadoop to be executed as a jobTracker, nameNode or as a dataNode and taskTracker depending on the kind of node we want each time. The script is called hadoop-starter and switches between four basic use cases to modify the configuration files: create cluster, add node to a cluster, delete node from a cluster and delete cluster. Note that the script modifies the current machine it runs in, not the other ones.

In the cluster creation case the script configures the machine to act as the master node of the cluster. By default the master will act as jobTracker and nameNode at the same time. In this case, the script will configure the node as jobTracker and nameNode in hadoop-site.xml, upload the files and start the daemons of HDFS and MapReduce.

When a new virtual machine has to be added to the cluster, the script configures the jobTracker and the nameNode in the configuration files, adds the node to the 'hosts' file and starts the dataNode and taskTracker daemons. Automatically Hadoop recognises the new node as part of the cluster and, if necessary, redistribute or copy data to the new node. The result is that the node is added to the existing cluster as a taskTracker and dataNode, contributing in the computation with new resources.

For the deletion of a node from the cluster it belongs the script only needs to assign the node to the 'hosts.exclude' file to let Hadoop the chance of redistributing the data and the calculations before killing the node. Even doing this, there is the risk that the machine is killed before having replicated the information to other nodes, but this is a risk assumed in Hadoop, like when a machine randomly switches off, and that's why there is a replication factor to ensure the success of the jobs.

The connections between the nodes in Hadoop is done using the secure protocol 'ssh', this means that the corresponding keys should defined in each node.

To ensure that all the configuration is done is important to specify the 'extension' of EMOTIVE used every time a interaction with the system is carried out, if the extension is not specified, EMOTIVE will create a different kind of machine, probably the default one, that is not going to have Hadoop on it, so is going to be a waste of resources.

5.2. Hadoop Scheduler

As mentioned before, EMOTIVE is a tool to provide virtual machines in a easy way, but it provides one virtual machine per request. In this project we want virtualize Hadoop and provide an easy way for creating virtualized Hadoop clusters. To deal with this, as mentioned in previous chapters, I made an extension of the default EMOTIVE scheduler. The scheduler developed for this project is a
RESTful web service that will have the basic functionalities of the EMOTIVE default scheduler: creation, migration, checkpointing, and destruction of virtual machines plus the functionalities for managing a cluster. This scheduler needs to keep the topology of every cluster up to date with all the changes made and for that, requires a structure to save the information. I called this structure HSP (Hadoop Service provider).

5.2.1. Conceptual model

To make the new scheduler capable of managing the virtualized clusters it will need to store the cluster structures in memory, up to date with the changes occurred during the execution. An important thing to take into account is that Hadoop uses the nodes IP or the host name as node identifier, but EMOTIVE uses a key-generator to assign the virtual machine identifiers inside the system. This lack of concordance forces the system to register more than one identifier per virtual machine, the one corresponding to EMOTIVE and the one corresponding to Hadoop.

Nevertheless, a set of clusters will be dynamically registered in the system by the HSP component. Each cluster registers information about the cluster name, the IP and the internal EMOTIVE identifier of the virtual machine acting as jobTracker and the same for the nameNode. I considered that is not necessary to specify the two different kind of nodes in the cluster as different classes if the identifiers of the master nodes are available separately of the slave nodes.

Each cluster contains a set of slaves and the system should register which virtual machines acting as slaves belong to each cluster. For each slave only the EMOTIVE identifier is registered. Decision of not storing IPs in slaves is because it takes some time to the DHCP server to assign an IP to a machine and we don't wont to wait in the creation process. This way we speed up the creation and if we are interested in deleting one, find the machine IP is faster than waiting to the DHCP server at the creation process. To facilitate the administrator identify the machines, all the slaves host-name will start with the name of the cluster.

As mentioned on the technology overview, Hadoop replicates the data among the nodes to ensure fault-tolerance in the system. By default the replication factor is 3 but this can be changed in the configuration files. Hadoop is forced to keep this minimum of replication in all the data. This replication factor should be taken into account at the time of deleting a node of the cluster. Even if Hadoop copies the data of a failed node to another running node automatically, is interesting to give Hadoop the chance to redistribute the data before deleting the node. From all that, we deduce that the replication factor should be taken into account and registered in the cluster data, as well as the current cardinality of the cluster.

The HSP is included in Hadoop Scheduler, that is a RESTful web service, and as a web service is is having a server side and a client side. In the server side it has an instance of HSP to keep control of the clusters, and a small set of functions: create cluster, add node to cluster, delete node from cluster, delete cluster and update cluster. To carry out this last operation, that is going to be called by the internal scheduler of Hadoop, is required an specification of a new class, that is called HadoopJobUpdate, that constains all the information transferred from the scheduler in Hadoop to the new scheduler in EMOTIVE. This new class contains information about the available and the required resources for carrying out the job in time and information about every job. For every job running it contains the jobId, the number of resources assigned to this job, the prediction and the goal in terms of time for finishing the job. This new type HadoopJobUpdate needs to be included in the server as well as in the client to allow the client create, fill up and send the data.

With this requirements the resulting conceptual model is the following:
5.2.2. HSP Functions

From the use cases and the conceptual model described in the previous section, a set of functions can be deduced as required, in this sections I list the functions with a brief description of each one.

- Functions corresponding to HSP:
  - Create cluster: this function receives a set of parameters with the name of the cluster, the identifiers of the master nodes, the number of slaves that should be created in the cluster and the percentage of CPU and memory required. Internally it creates a new instance of Cluster and does the assignation to the attributes. Afterwards adds the new cluster to the existing set of clusters. This function doesn't provide a return value. As a result, we obtain a header like this:

\[
\text{createCluster} \ (\text{String } \text{jobTrackerId}, \ \text{String } \text{jobTrackerIp}, \ \text{String } \text{nameNodeId}, \ \text{String } \text{nameNodeIp}, \ \text{String } \text{masterName}, \ \text{int } n\text{Slaves}, \ \text{int } \text{CPU}, \ \text{int } \text{MEM}) : \text{void}
\]
• **Add node**: this function receives a the IP of the jobTracker node from the cluster the node should be added, and the EMOTIVE identifier of the node to add. Internally the node is added to the list of nodes in the cluster and the cardinality of the cluster is increased in one. This function doesn't produce output. The header of the function is as follows:

```java
addNode(String masterIp, String slaveId) : void
```
○ **Can delete node**: this function is used to check if the cluster has enough nodes to keep the level of replication if one node is deleted from the cluster. The only parameter required in this function is the jobTracker IP, to identify the cluster. The function performs the subtraction of the cardinality of the cluster and the replication factor. If there are more nodes than the required to keep the level of replication, at least one node can be deleted. The output is the identifier of one of the nodes if a node can be deleted, if not, the output is null. The header looks like this:

```
canDeleteNode(String masterIp) : String
```
- **Delete cluster:** this function deletes the node from the set of clusters. The only parameter required is the jobTracker IP, that is the cluster identifier inside the set. The functions don't produce output.

```java
deleteCluster(String masterIp) : void
```
The description of the functionalities for the classes Cluster and Slave are basic, like getters and setters, so I just describe them:

- **Cluster**
  - getters and setters of the attributes.
  - **Add slave**: this function adds one instance of node to the set of nodes of the cluster. The only parameter required is the node. No output is produced.
    
    \[\text{addSlave}(\text{Node slave}) : \text{void}\]
  - **Delete slave**: this function deletes one instance of node from the set of nodes of the cluster. The only parameter required is the node. No output is produced.
    
    \[\text{deleteSlave}(\text{Node slave}) : \text{void}\]

- **Slave**
  - getters and setters of the attributes.

### 5.2.3. Hadoop Scheduler functions

Hadoop Scheduler is the core of this project, the piece managing the virtual machines. This piece of software is an extension of the default scheduler in EMOTIVE and is implemented as a RESTful web service using the Jersey libraries from Java. As an extension of the default scheduler, it has the basic functionalities for creating a virtual machine, destroying a virtual machine, access to the lower levels in EMOTIVE for checking the state of the machine, etc. and it includes the functionalities related with this project: creation of a cluster of virtual machines, addition of a virtual machine to a cluster, deletion of a virtual machine from a cluster, deletion of a cluster and updating the topology of a cluster depending on the demand, that normally will come from the internal scheduler of Hadoop.

As a RESTful web service, all the data and the functionalities are considered resources, and the resources are structured like directories. For every resource we can define four basic operation: GET, PUT, POST and DELETE. For this projects data and the functionalities are just few, so the structure is very simple. We have a set of clusters and for each cluster we have addition, deletion
and update operations. For this simple schema I designed the following structure:

- **/clusters** → For this address we have the functionalities of creation and deletion of a cluster with the POST and DELETE methods.

- **/clusters/{clusterId}** → For this address we have the functionalities of addition and deletion of a node to/from a cluster with the POST and DELETE methods.

- **/clusters/{clusterId}/updates** → For this address we have the functionality of updating the topology of a cluster with the POST method.

Once defined the simple address structure, here is a detailed description of all the new methods corresponding to the extension made for this project divided in two blocs, the one corresponding to the server and the one of the client:

**Server side:**

The server keeps logging the requests received is text files, to keep track of what happens in the server and which are the actions carried out while the service is on.

- **Create cluster**: this function receives the cluster name, the number of slaves required, the percentage of CPU required and the amount of MEMORY required for the virtual machine. All this information is sent by the client and once received in the server side, this function sets all this information in a new instance of virtual machine, that inside EMOTIVE is called Compute, defines a new instances of JSDL, that is the object defined for passing date and order to the virtual machine, and sets up the call to the script already installed in the machine for configuring the new virtual machine a Hadoop jobTracker, in this case the call is: “`bash /aplic/hadoop-starter cluster replicationFactor physicalNode`”. Is important to specify the replication factor at the master creation time, and the physical node where this machine is hosted within EMOTIVE, because the internal scheduler used in Hadoop, Adaptive scheduler, requires this information to connect with the service for sernding the updates of each cluster. After that, it proceeds to the creation of the virtual machines acting as master nodes in the cluster, calling the functionality of the default scheduler.

Before creating the rest of the machines, the function wait until the master nodes are created and the DNS server has assigned an IP address to the new virtual machines, because for the rest of the machines we need to know the IP address of the jobTracker and the nameNode to pass them as parameters for the configuration of the machine. Once the new machines have IP, the function calls the Hadoop configuration script (hadoop-starter) using the JSDL object to pass the information of the order, and a new instance of cluster is created using HSP. Afterwards the function starts creating new nodes calling the “addNodeCluster” function present in the same service, as many times as indicated in the parameters.

```java
createCluster(clusterName, numberSlaves, replicationFactor, cpu, memory)::void
```
• **Add node to cluster**: this function receives only the IP address from the jobTracker of the cluster in where the client wants to add a new node. Having the IP address of the jobTracker of the cluster the new node is capable of connecting with is and join the cluster. With this information, this function defines the properties of the new virtual machines copying the configuration values of memory and percentage of cpu from the master node, and calls the creator function of the default scheduler. Finally if the creation is carried out without problems, HSP add the information of the new node to the cluster it belongs.

\[addNodeCluster(jobTrackerIp):void\]
• **Delete node from cluster**: this function receives only the IP address of the jobTracker of the cluster in which the client wants to delete a node. In this function the procedure starts by checking if deleting a node from the specified cluster is going to corrupt the constraint of the cardinality of the cluster. If the result of the checking, using `canDelete` function from HSP, is an IP, this means that the deletion of a node doesn't imply any problem for the replication of the data among the cluster, otherwise the function ends and does nothing. If the deletion is permitted, then the function sends the other to *hadoop-starter*, the script in charge of the configuration of the machine, and the script stops the daemons in the virtual machine corresponding to Hadoop. To let Hadoop time for redistributing the data among the cluster before the node is destroyed, the function gives it a bit of time and then destroy the node, freeing resources from the physical machine.

```
deleteNodeCluster(jobTrackerIp):void
```
deleteNodeCluster

- 1.1: clusters.get(jobTrackerIp)
- 1.2: clusters.get(jobTrackerIp)
- 1.3: canDelete(jobTrackerIp)
- 1.4: canDelete(jobTrackerIp)

returns slaveId if can delete a node in the cluster without corrupting the replication constraint, otherwise null

optional

[ if slaveId!=null ]

1.1: getDomain(slaveId)
- 1.2: getDomain(slaveId)

  take slaveIp from domain

1.4: setCommand("bash /aplic/hadoop-starter deleteNode " +slaveIp)
1.5: setCommand("bash /aplic/hadoop-starter deleteNode " +slaveIp)

grave

[]

1: runTask(jobTrackerId, user, jsdl.toString(), false)
2: runTask(jobTrackerId, user, jsdl.toString(), false)

wait to let Hadoop time to move the copies of the files from the nodes is going to be destroyed to another one

grave

[]

1: destroy(slaveId)
2: destroy(slaveId)
• **Delete cluster:** this function receives the internal EMOTIVE ID of the jobTracker of the cluster the client wants to delete. There are no conditions to check before deleting the cluster, the functions goes straight to obtain from HSP the list of the nodes that are part of the cluster and the cluster is removed from HSP. Then proceeds to delete one by one all the virtual machines from the physical machine. Note that in this case there is no need of stopping the daemons of Hadoop before deleting the machine, the function destroy the virtual machines.

\[
deleteCluster(jobTrackerId): \text{void}
\]

- **Update cluster:** this function receives the IP address of the jobTracker of the cluster the system is receiving the update from, and the update with all the information contained in an object of type HadoopJobUpdate. Note that update is a complex structure and this should be specified for RESTful web services.

The first thing this function does is saving the update in the logs and then checks if the
cluster sending the update is already modifying its topology. This is important because depending on the resources the physical machines containing the service has, the creation or deletion process of a virtual machine can vary from seconds to minutes, and the updates coming from the internal scheduler of Hadoop are can arrive in intervals of 3 or 4 seconds. If the cluster sending the update is not currently being modified, the service checks the update, making the difference between the required resources and the current resources. Depending on this comparison the service starts a new thread adding or deleting a node from the cluster, if needed. Note that the creation of a new thread for this operation is important because if not the client will be stopped until the end of the update, and this will block the job scheduler inside Hadoop, stopping the progress of the jobs.

\textit{updateCluster}(\textit{jobTrackerIp, update}):\textit{void}

Client side:

The client side has the same operations as the server side, but in this case the functions are simpler because are calls to the service.

- **Create cluster**: this function receives the cluster name, the amount of slaves required, the percentage of cpu and the memory required for the machines. It takes the parameters and sends them to the server.
```java
createCluster(clusterName, numberSlaves, replicationFactor, cpu, memory):void
```

- **Add node to cluster**: this function receives the IP address of the jobTracker of the cluster an addition of a node is required. It takes the parameter and sends it to the server.

```java
addNodeCluster(jobTrackerIp):void
```

- **Delete node from cluster**: this function receives the IP address of the jobTracker of the cluster a deletion of a node is required. It takes the parameter and sends it to the server.

```java
deleteNodeCluster(jobTrackerIp):void
```

- **Delete cluster**: this function receives the internal EMOTIVE IP address of the jobTracker of the cluster. It takes the parameter and sends it to the server.

```java
deleteCluster(jobTrackerId):void
```

- **Update cluster**: this function receives an update from cluster and sends it to the server, together with its IP address.

```java
updateCluster(update):void
```

### 5.3. Graphical user interface

To help ease the creation process, in this project I extended the graphical user interface of EMOTIVE for including a new part with an specific view for creating Hadoop clusters. The GUI in EMOTIVE is implemented as a website with Java code using ZK framework, a RIA open source framework.

The functionalities are very simple, creation of a cluster and once this is created, functionalities for changing its topology: add node, delete node or remove cluster. The website is defined to create and call the service created in the project every time a functionality is required. This GUI then, calls HadoopSchedulerRESTClient, that is the client described in the previous section. In addition to the functionalities of the client, the GUI provides three more functionalities related with Hadoop: show nodes, show jobs and run task. The first two functionalities open the web user interface installed in Hadoop, allowing the user having a look on the filesystem (HDFS) and the information about MapReduce and the jobs running over Hadoop. The third functionality is used can be used for running whatever we want inside the machine, from uploading files to the virtual machine, to run jobs, passing by uploading the files to Hadoop file system, so this is what will be used once the cluster is created and we want to use the machine.
We can see in the screen-shot above a form in where is possible to specify the creation of a cluster and a '+' button to actually create it. Then once the cluster is running the machines automatically appear on screen, having the master with an icon and the name we assigned to the cluster and the slaves with a different icon. Selecting the master node a pop-up menu appears showing the functionalities described above.

The ‘run task’ functionality pops up another form for executing commands on the virtual machine, allowing uploading files and defining output files for the command we want to run, as shown in the following screen-shot:
In addition to the GUI, there is also a terminal-based interface with the basic functionalities of the default EMOTIVE scheduler: create virtual machine, destroy virtual machine and show virtual machines running. All this basic EMOTIVE functionalities are complemented with ones corresponding to the extension made for the project: create cluster, add node to cluster, delete node from cluster, delete cluster and update cluster, that in this case is going to send a “hand-made” update, specifying manually the values of the update.

5.4. Summary

In this chapter we saw the approach to the design and the implementation carried out in this project. As a summary, this project has, from bottom to top, a predefined virtual machine image with Hadoop installed on it. In a second level, an extension of EMOTIVE scheduler implemented as a RESTful web service with basic functionalities for managing clusters (create cluster, add node, delete node, delete cluster), and the integration with the internal scheduler of Hadoop (Adaptive scheduler) that sends updates of the requirements of the cluster for running the jobs according to some time restrictions. On top of that, a GUI implemented as a website.

For seeing the interactions described in the design chapter, let's use the example of creation of a cluster. From top to bottom, the GUI can send a request to the system asking for a virtual Hadoop cluster. The request uses the RESTful web service and passes from the client side to the server side, reaching the physical machine. The server side uses the default functionalities inherited from the default scheduler in EMOTIVE to create a virtual machine. The new node will run the master daemons of Hadoop, so the script left in the predefined image of the virtual machine configures Hadoop for running as a jobTracker and a nameNode in this virtual machine. Once the master node is ready and the DNS server has assigned an IP to it, EMOTIVE creates as many nodes as specified in the request, these acting as slaves (dataNode and taskTracker) in the cluster. The new slave nodes connect to the master node and join the cluster automatically and become ready to store and run jobs. Meanwhile, the server stores information about the cluster topology, so later requests can be carried out and the consistence of the system won't be affected. When the cluster is set up, the internal scheduler of Hadoop (Adaptive scheduler) starts interacting with the service, sending information about the requirements the jobTracker has for carrying out the jobs based on the resources it has registered. This information is used by EMOTIVE Hadoop Scheduler for adding or removing nodes in the cluster according to the predictions made by Adaptive scheduler, so if a job
requires more resources for finishing on time, EMOTIVE adds nodes to the cluster running this job and the internal scheduler distributes the resources among the running jobs, otherwise it can remove nodes from the cluster if there are more resources than what the jobTracker requires for the running jobs. Finally when the creation is done and the Adaptive scheduler is sending updates to EMOTIVE Hadoop service, the GUI shows the cluster on screen.
6. Deployment

In this chapter is described the deployment process for the project, dividing the chapter in three parts: release, installation and future updates.

6.1. Release

This project extends EMOTIVE Cloud with Hadoop functionalities. As an extension for EMOTIVE, the compilation of the project is done integrated with the rest of EMOTIVE. Up to now EMOTIVE is using Maven 2 for compilation purposes. Maven uses xml files for structuring and defining dependencies of what is going to be compiled, and defining the paths or the repositories where the project to be compiled, or the dependencies, are. These files are always called pom.xml. This project is divided in two parts, the server side and the client side, so there is a pom.xml file that describes the sub-projects and how to compile them. This file is common to all the schedulers in EMOTIVE, so I just needed to add the two sub-projects to the list. Inside each sub-project there is a pom.xml file too, in this case specifying the name of the artifact created, the type of packing, the dependencies, the repositories where to look for the dependencies, the tasks run by maven and the path where to leave the final product.

6.2. Installation

For installing EMOTIVE in a Debian OS machine or group of machines:

- The first step preparing the servers with XEN, Libvirt and Tomcat:

  XEN and LIBVIRT:
  apt-get install xen-linux-system-2.6.26-2-xen-amd64 libvirt0 libvirt-bin xen-hypervisor-3.2-1
  JAVA:
  # apt-get install openjdk-6-jdk or download and install JAVA-JDK manually
  TOMCAT:
  #apt-get install tomcat6 or download and install manually
  OTHERS:
  #apt-get install make gcc libssl-dev ant maven2 bzip2 build-essential
  #apt-get install libc6-dev zlib1g-dev debootstrap dhcp3-server bind9 module-init-tools
  MAVEN AND SVN:
  #apt-get install maven2  svn  subversion

- Afterwards, download EMOTIVE Cloud, edit the install.cfg file and install

  create directory EMOTIVE “/EMOTIVE_PATH”
  # svn co https://emotivecloud.svn.sourceforge.net/svnroot/emotivecloud/
  /EMOTIVE_PATH*
  *EMOTIVE_PATH=”Your EMOTIVE directory”
  or download here: http://emotivecloud.svn.sourceforge.net/viewvc/emotivecloud.tar.gz?
  view=tar
  edit install.cfg
  export APACHE_PATH=”Your APACHE directory”
  export EMOTIVE_PATH=”Your EMOTIVE directory”
# ./install full

- Next step is configuring “/etc/VtM/*” and directory “domU/”

```bash
# ./install create
# vi /etc/VtM/rm.properties  (in here: check domains name, and set the chosen scheduler)
# vi /etc/VtM/vtm.properties  (check the paths variables)
# ./install domu
```

- Prepare server and run client

```bash
# catalina.sh run
run Scheduler.jar or VtM.jar
```

- Scheduler-client

```bash
java -jar HadoopSchedulerRESTClient.jar
```

For more advanced information check [http://www.emotivecloud.net/](http://www.emotivecloud.net/)

Regarding to the virtual machine, preparing an image of a virtual machine with the software already installed consists on:

- Create standard virtual machine with EMOTIVE and access to it by ssh.
- Create a directory where the modifications of the extension will be saved.
- Download and install Hadoop in the virtual machine together with the software required by Hadoop. The particularity of the Hadoop version installed for this project is that it should include the internal job scheduler called Adaptive, developed by Jordà Polo in his final project degree, and recompiled to force Adaptive scheduler calling the client of the EMOTIVE Hadoop service every time the scheduler makes a prediction about the evolution of the jobs according to the current resources.
- Add the the jar of EMOTIVE Hadoop service client with all the dependencies.
- Modify the configuration files to be ready for the modifications made by the installation script.
- Copy the installation script: hadoop-starter.sh.
- Shut-down the virtual machine and copy the image in the directory `extensions` of EMOTIVE.

### 6.3. Future updates

For future updates of this scheduler is important updating the script used for configuring Hadoop in the virtual machine
7. Testing

In this chapter I describe experiments carried out for testing the smooth running of the system. For testing the functionalities of the project we need to install EMOTIVE, start the service, create clusters and run jobs for checking that the Adaptive scheduler interacts smoothly with the service provided by EMOTIVE. In this way is possible to appreciate the performance of the update, add node and delete node functionalities.

The testing has been done in a single server with:

- number of CPUs: 8
- CPU: Intel® Xeon® Processor E5440 (12M Cache, 2.83 GHz, 1333 MHz FSB)
- GNU/Linux Debian OS
- xen version 2.6.18
- libvirt
- 16GB RAM

Before executing any job the first thing is creating a cluster. The creation process is directly proportional to the amount of nodes we want to have in the cluster, more nodes implies more time of creation. In a machine as the one described above, creating a virtual machine takes on average 48000 ms. approximately, and deleting a virtual machine 1400 ms.

Is easy to appreciate in the tables that the time of creating a cluster is directly proportional to the
amount of nodes we want to create at the beginning, and the same for the deletion. In the creation tests of clusters with 3 nodes, the time is 3 times bigger than the time obtained from the tests of adding a node in the cluster.

For appreciating the performance of the system I show two different executions with the cluster already created and the input files already uploaded to Hadoop file system. The first execution has a job deadline that differs significantly from the final execution time, and the second execution has a more accurate approximation of the job deadline to the real execution time.

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Execution 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>– job deadline 150s</td>
<td>– job deadline 200s</td>
</tr>
<tr>
<td>– job done in 292s</td>
<td>– job done in 225s</td>
</tr>
<tr>
<td>– 100 files in input</td>
<td>– 100 files in input</td>
</tr>
<tr>
<td>– replication factor = 3</td>
<td>– replication factor = 3</td>
</tr>
<tr>
<td>– 1024MB RAM each virtual machine</td>
<td>– 1024MB RAM each virtual machine</td>
</tr>
<tr>
<td>– number of virtual machines trunk at 10 to make the figure understandable.</td>
<td>– number of virtual machines trunk at 10 to make the figure understandable.</td>
</tr>
</tbody>
</table>

**Execution 1:**

In this execution we can appreciate that Adaptive scheduler is predicting since the beginning of the job that the amount of nodes is not enough for carrying out the job in the defined time, so it starts requiring more nodes to EMOTIVE. The service provides more virtual machines, but the creation process of a machine slow in comparison to the needs of the job so even if the cluster is having more nodes, the job can't finish in time. Once the job is finished, the internal Hadoop scheduler, Adaptive scheduler, tells EMOTIVE that it doesn't require nodes because the job is done and EMOTIVE starts deleting jobs until the amount of machine is the same as the replication factor specified for the cluster. Note that both addition and deletion of nodes is not made in parallel, because the difference between the interval of time between the updates and the time the service needs for creating or deleting machines is very big and if it would have been done in parallel the server could have collapsed due to the amount of virtual machines. Even though there is a big difference between both times, EMOTIVE helps the job to adapt its execution time.
**Execution 2:**
In this execution the times of deadline and execution time are more approximated and we can appreciate that it takes a bit to Adaptive scheduler to see that the amount of machines is not enough for finishing on time, because it starts asking for more machines more or less at the middle of the execution. Once Adaptive scheduler asks for more nodes, EMOTIVE provides as many as it can before the end of the execution and when then job is finished removes nodes until reaching the minimum of the replication factor.

Note that both execution are made with a lack of time for the execution of the jobs, this is because if the Adaptive scheduler doesn't need more nodes than the ones it has because of the replication factor, EMOTIVE does nothing but saving the updates.

On the other hand, EMOTIVE adapts the topology of the cluster for helping the Hadoop finishing jobs according to an stipulated time. Is important verifying that this modifications in the topology really help Hadoop and I did it checking the predictions sent by Adaptive scheduler.
Is easy to appreciate that the prediction time line is getting closer to the goal time as the execution receives more nodes created by EMOTIVE. With this test, we verified that EMOTIVE can help Hadoop reaching its goal in terms of execution time thank to the modifications in the topology of the cluster.
8. Project plan

The project began in December 2008. At first it didn't have a fixed long-term schedule, but it was already centred around MapReduce and cloud computing, and there was a rough idea of how it would evolve if everything worked as expected. The dedication to the project has been only partial. It wasn't until May 2009 that the scope and goals of the project were finally set, and the dedication also became full-time.

In September 2009 I went to the University of Reading in the ERASMUS program and, even though I continued working on the project, the distance made it impossible finishing the project for February 2010, due to the changes made in EMOTIVE that required migration from SOAP web services to RESTful web services.

It was after the stay in the University of Reading that the project migrated to REST and reached its final part, the integration with Adaptive scheduler.

Due to this big break in the project, the plan and the real development of the project differ a lot. In the first table I show the plan defined for the project and in the second table the real development.
<table>
<thead>
<tr>
<th>Task</th>
<th>May</th>
<th>June</th>
<th>July</th>
<th>August</th>
<th>Sept</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Jan</th>
<th>Feb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Understand MapReduce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarize with Hadoop</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster administration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarization with virtualization</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Election of virtualization platform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual virtualization of Hadoop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler prototype with SOAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GUI with SOAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preliminary report</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler with SOAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration with Adaptive scheduler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation slides</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity</td>
<td>2009</td>
<td>2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous</td>
<td>11 25 8 22 6 20 3 17 31 14 28 12 26 9 23 7 21 5 19 2</td>
<td>17 31 14 28 12 26 9 23 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Understand MapReduce</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarize with Hadoop</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster administration</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarization with virtualization</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Election of virtualization platform</td>
<td>x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarization with EMOTIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual virtualization of Hadoop</td>
<td>x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler prototype with SOAP</td>
<td>x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GUI with SOAP</td>
<td>x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preliminary report</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler with SOAP</td>
<td>x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarization REST</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Migration REST</td>
<td></td>
<td>x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration with Adaptive scheduler</td>
<td></td>
<td>x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td>x x x x</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation slides</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9. Summary and conclusions

The goal of the project was create a tool for creating and managing virtual Hadoop clusters. For that, the extension of EMOTIVE default scheduler created in this project has the functionalities of creating a cluster, adding nodes to a cluster, remove nodes from a cluster, delete a cluster and, in addition to this, joining this project with the another project that developed an internal scheduler for Hadoop that makes predictions on the executions of the jobs redistributes the resources between the jobs depending on their needs, so the internal scheduler would be able to send information about the evolution of the jobs and EMOTIVE can decide assigning more resources, keeping the assigned resources as they are, or removing resources so if it needs them for another task the resources would be available.

In top of that, the requirement of making it accessible and easy to deal with it, so I developed a graphical user interface extending the one of EMOTIVE, so any user can create and manipulate Hadoop virtual clusters over EMOTIVE very easily.

The experiments showed that with this project, EMOTIVE provides a flexible and self-adaptive service for Hadoop virtual clusters.
References


