A Cloud-Based Execution Environment for a Pandemic Simulator

Maurizio Basile, Massimiliano Gabriele Raciti

Reg Nr: LIU-IDA/LITH-EX-A-09/036-SE
Linköping 2009
Master’s Thesis

A Cloud-Based Execution Environment for a Pandemic Simulator

Maurizio Basile, Massimiliano Gabriele Raciti

Reg Nr: LIU-IDA/LITH-EX-A–09/036–SE
Linköping 2009

Supervisor: Henrik Eriksson
IDA, Linköpings Universitet

Examiner: Henrik Eriksson
IDA, Linköpings Universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Title

A Cloud-Based Execution Environment for a Pandemic Simulator

Författare

Maurizio Basile, Massimiliano Gabriele Raciti

Nyckelord

Condor HTC, Amazon EC2, Pandemic Simulation, Cloud Computing
Abstract

The aim of this thesis is to develop a flexible distributed platform designed to execute a disease outbreaks simulator in a fast way over many types of platforms and operating systems. The architecture is realized using the Elastic Compute Cloud (EC2) supplied by Amazon and Condor® as middleware among the various types of OS. The second part of the report describes the realization of a web application that allows users to manage easily the various part of the architecture, to launch the simulations and to view some statistics of the relative results.
Acknowledgments

We would like to express our gratitude to the professor Henrik Eriksson for having made this work possible, accepting our collaboration and supporting us in all phases of the thesis project.

We would like to thank Anders Fröberg for his technical support, and all the people of the Department of Computer and Information Science for making its environment so friendly.
Contents

1 Introduction 1

2 Background 7
   2.1 The Disease Spread Simulator ......................... 7
   2.2 Condor ........................................... 9
   2.3 Amazon Elastic Compute Cloud ......................... 10
   2.4 Google Web Toolkit ................................ 11
   2.5 Ext GWT: Rich Internet Application Framework for GWT ........................................... 12

3 Architecture 13
   3.1 Overview .......................................... 13
   3.2 Condor on Amazon EC2 Instances ...................... 19
   3.3 Data Storage System ................................ 28
   3.4 Security ............................................ 35
   3.5 Performance evaluation ............................... 42
      3.5.1 Amazon EC2 performance ......................... 42
      3.5.2 Office machines tests ............................ 45
      3.5.3 Considerations ................................ 47
      3.5.4 OS comparison on a Pentium® 4 .................. 50

4 The web interface 53
   4.1 Overview ............................................ 54
   4.2 The environment ..................................... 54
      4.2.1 Support tools ................................ 55
   4.3 Amazon management .................................. 55
   4.4 Condor management .................................. 58
   4.5 Simulation job and scenarios ........................ 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6 User access</td>
<td>63</td>
</tr>
<tr>
<td>4.7 Security</td>
<td>64</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Architecture</td>
<td>67</td>
</tr>
<tr>
<td>5.2 Web Application</td>
<td>71</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>Bibliography</td>
<td>75</td>
</tr>
<tr>
<td>A Use Case</td>
<td>79</td>
</tr>
<tr>
<td>B Deploy</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The study of epidemic phenomena is an important topic that all communities are taking into account to be prepared in case of influenza outbreaks. The aim of these studies is to understand what are the best actions that can be taken in case of that events to contrast and limit the number of transmissions until the disease spread is over. Many actions can be done as preparations. In case of a pandemic influenza local authorities have to follow some prevention rules that are always good in general, as it happens for example for earthquakes or floods. But other actions can be taken as only intervention at unfolding events because not all information about the virus and its spreading are known before. Furthermore the effect of a disease can have a different impact on different conditions, like the community structure or the economic condition of the population. That means that the same intervention cannot be applied everywhere but must be applied after a careful study of its possible results. The evaluation of the best intervention at the moment of the influenza outbreak has a major complexity because many factors have to be considered at the same time, then decision-makers collaborating with epidemiologist and researchers have to choose the best solution in the shortest time possible, considering for each kind of actuable intervention what are its benefits and how it can reduce the reproduction rate. Computer-based simulation can be a good instrument that helps the evaluation of the feasibility of interventions and it can be a good way to understand the possible dynamics of the spreading and the effect of interventions on local modelled communities.
The department of Computer and Information Science of the University of Linköping, in collaboration with VSL Research Lab, developed an high-modular pandemic influenza simulator that takes as input a model of population, a model of disease spread and a model of intervention and gives as output some statistics about the spreading of the disease, in which the most significant information is the reproduction rate that is useful to understand if the influenza spread is rising up or if it is decreasing under certain conditions. The models of population, of disease and of intervention are not built into the simulator, but are taken as external information. The simulator has a single core that is valid for all kinds of experiments and can be used to simulate different scenarios simply changing inputs. In this way it is possible to create different models of population, disease and intervention and it is possible to run the simulations in parallel in order to understand the impact of a disease in different kind of populations, the impact of an intervention on different models of diseases over different models of populations and so on. The main aim is however the possibility to find the best solution as series of interventions keeping fixed the model of disease and population. What happens if we close all schools? Is it a benefit or not? Is a good choice to close all public activities? These are some questions that the simulator is able to answer with numerical results.

The population is modeled respecting as much as possible its real condition, like the number of inhabitants, the number of schools, offices and public places. Many other factors are also considered, like for example the university cities that usually have many students between 18 and 30 years old living on shared apartments or student houses. The population is generated randomly, in particular the number of families, the number of persons per family and their age distribution. To have a correct estimation of the reproduction number it is better to run the simulation on a large number of random-generated populations based on the same model, and considering the average value of all reproduction rates instead of the singular ones.

The large amount of data necessary to model the complete job and the large number of simulations causes a large time-demand and cpu-usage computation, that means that a single computer can spend a lot of time to complete the execution of a job. It is therefore necessary to design an architecture that supports the execution of the simulation,
which reduces significantly the time required for the computation of results and at the same time does not lead high efforts in economic management and by its construction and maintenance.

Given the issues of the simulator (modularity, multiplatform) the basic idea is to follow the same guidelines: the architecture must be flexible, multiplatform and should exploit as much as possible the parallelism of jobs. The first option that we though was to build a distributed environment buying or reusing dedicate hardware to create a cluster: a certain number of computers more or less powerful, a central server and a middleware between the software level and the distributed infrastructure that can transparently split the jobs in parallel through the network and collect all the result to the main server. Since it is an actuatable solution, the problem is the scalability and the low flexibility: adding or removing capacity is of course simple but not immediate, the set-up costs are considerable and it is necessary to keep all the machines constantly updated.

The second solution could be the use of other preexisting architectures of supercomputers used by researchers for other projects, installing just what we need for the execution of our jobs. But also in that case, maybe more than in the previous, the flexibility is very low and we do not have free access of all the resources. What we need is an architecture that is easy to set up, easy to maintain, that allows a simple interaction with the end users and if it is possible is should be fast to be replicated.

This thesis work describes the project and the effective implementation of a solution of an high flexible architecture based on the Elastic Cloud Computing (EC2) service provided by Amazon [2], which allow us to allocate or deallocate a dynamic amount of computational resources in an extremely easy way. Amazon EC2 reduces the problem of maintainability, since the hardware is outsourced and updated by the supplier, it maximizes the scalability because you can allocate a variable computational capacity according to the needs of calculation and you can use it only for the time necessary to conclude a simulation job. It minimizes the costs because you do not have to buy the hardware but you pay just the hours in which you use capacity in relation to the number of allocated computational unit. Keeping the software updated is a simple task because each instance of a computational unit is dynamically created from a base-image that is built
once, so if some updates are necessary they can be applied in a few steps to the base image and then they become valid for all the future allocated instances. The system is fault tolerant because Amazon provides transparent fault tolerance. In other cases if a region of the cloud becomes unavailable, it is possible to allocate other instance in other regions, so we do not have to take care about eventual faults of the computational environment. As middleware between the user application and the underlying architecture we used Condor [7]. Condor takes care of the allocation of the simulation tasks to the computational units through a queue of service, it takes care of the automatic transfer of the input files to the simulator, it transfer the executable that is suitable with the architecture in which the job is going to be executed and it takes care of the subsequent return of the results, all in a complete transparency. Security is an important aspect of our architecture, since the hardware is outsourced and all the data is transferred over Internet. The project of our execution environment considers this aspect, providing authentication and encryption methods using the latest and safe web technologies (SSL). The second part of the project regards the development of a web user interface that allows the management of the whole architecture. It allows to collect simulation jobs, launch them and view the result once they are available. The web user interface also permits users to start and stop all the required instances on Amazon EC2. The web interface is implemented using the latest languages for web application, which makes the interaction user friendly and has also a nice graphic.

**Thesis Overview** The thesis report is structured as follows:

- In chapter two, we present a short overview and some issues of the technologies that we used to build the execution architecture and the web user interface.

- The third chapter focuses on the project of the architecture of execution, explaining from time to time the problems and the solutions.

- The fourth chapter talks about the implementation of the user web interface.
• In the fifth chapter, we discuss about the work done, considering advantages and disadvantages of our choices and presenting eventual updates or improvements.

• Chapter six concludes our report.

Adding to that the appendix number one and two show one use case of a complete simulation job and the deploy of the web application.
Chapter 2

Background

In this chapter we briefly summarize some background information that is necessary in order to understand the context of the project and the technologies that are used to build the architecture.

2.1 The Disease Spread Simulator

The Pandemic Simulator has been described in [24], [19], [29] and [25]. For our purpose we present a brief introduction to show how it works. The simulator takes as input three kinds of models [24]:

- **The disease**: each kind of infection has some different transmission mechanisms, such as person-to-person contact, through aerosolized droplets of respiratory secretions or through contaminated objects. The model includes the representation of the infection phases: usually an infected person has first an incubation phase, in which he may or may not show symptoms, and after that period he can recover with immunity or succumb.

- **The population**: is modeled using the concept of mixing groups. A mixing group is a place where people interact, like households, schools, workplaces, swimming pools, subways, theaters and so on. Each person has much probability to be infected as many mixing groups he frequents.

- **Interventions**: models the actions that can be taken to reduce the number of infections. Vaccinations and antiviral medications
reduce the probability to be infected or to infect other people, but other actions works on the disease transmission reducing the number of contacts between people, like for example closing schools.

The models passed as input of the simulation are represented in an intermediate specification format which is XML. Figure 2.1 shows the architectural layers of the simulator [19]. As reported in [24] (page 7) "The simulation engine executes simulation jobs specified in the modeling environment. From the specification, it instantiates the virtual population, assigns initial cases of infection, and implements the prescribed intervention policy. For each step of the simulation, the simulator engine propagates the state of infection in the population. First, the disease state of people in the incubating or infectus states is advanced. Recovering individuals acquire immunity. Then, each person susceptible to infection is exposed in her mixing group. The simulation engine calculates the probability of becoming infected as the combined probability of becoming infected in any of the mixing groups. In each mixing group, that probability of becoming infected depends on the number of infected persons in that mixing group and the probability of transmission for each contact, which may be age dependent."
The combined probability is used in a random binary test to determine whether the susceptible person becomes infected. Person that become infected transition to the incubating state”. Once the simulation is concluded, the output file is also an XML file that summarizes first the initial hypothesis and then presents the results of the simulation.

2.2 Condor

One of the requisites when running a simulation is to exploit all computer resources in order to improve the performance in terms of time. To obtain top performance a workload management system is required, and Condor [7] can be considered the best solution. Condor is an advanced workload management system that makes it possible to execute intensive computations over a pool of machines in a distributed environment [8]. Like other batch systems, Condor keeps a queue of tasks, a scheduling policy and priority schemes, it submits jobs to the machines choosing dynamically according to the policies, where to execute the job and at the end it returns back the results. But the main difference is that Condor is able to use idle CPUs of desktop workstations, improving the throughput using that computation capability that otherwise would be wasted. It also implements the concept of migration of jobs. The Condor developers philosophy is that of the High Throughput Computing (HTC). Rather than considering the computing performance measured in floating point operation per second, Condor developers’ idea is to evaluate the amount of computation in a long term period. That means for example how many floating point operation per month or per year the system can do. In a distributed environment, Condor manages the computation resources in a way to exploit the idle CPU time to do something useful. As usual, the most of the CPU time of normal PC is under the 10 percent, this can be seen as a high waste of resources. One of the biggest problem is the ownership of computer resource. In order to expand the pool of machines sharing computation resources, all of the customers should be guaranteed, so their needs should be satisfied and their safety must be guaranteed. Condor can run on a single machine, in that case it acts like a monitoring tool that pauses the jobs and restarts it according to the needs of the user, and it can run on a cluster of workstations
machines as submission tool. Condor can have different roles on a pool of machine that are specified at the installation time: an instance can be a manager, a submit machine or an executer. The manager has the role of collecting information about the pool. The submit machine keeps a queue of jobs and is responsible to submit them to the executers. The executer is a simple workstation which shares its idle CPU time for computations. These three roles can be applied together, so a single machine can be both a manager, a submitter and an executer. Condor is released under the Apache License, Version 2.0.

2.3 Amazon Elastic Compute Cloud

In our context, the machines are launched expressly for our purpose; the computation resources are bought from Amazon Elastic Compute Cloud (EC2) [26, 2] that provides the possibility to instantiate and manage virtual machines easily, using the web service interface. With Amazon EC2 it is possible to run and terminate as many instances as needed in that moment, so it can be defined fully scalable (Elastic) and the charges are applied only for the used capacity. The instances are launched using an image of the system that is called Amazon Machine Image (AMI). It is possible to choose one of the many public images, or creating a private one from scratch. When an instance is launched, it is possible to access to it using ssh if it is a Linux machine, or via a remote desktop connection if it is a Windows machine. The control of the status of the instances is by the web service APIs. It is possible to customize an instance with the desired services. The instances are secure, since it is possible to configure a firewall to grant or deny the access from outside the cloud. There are two types of instances using a 32-bit platform, the standard one has 1 virtual core and 1 EC2 Compute Unit; the High-CPU instance has 2 virtual cores and 5 EC2 Compute Units. As Amazon reports, "One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" [2]. Other types of instances can be used with a 64-bit platform. The most powerful is a High-CPU Extra Large with 8 virtual cores and 20 ECUs. Several operating systems are available from the list of public AMIs (Windows Server 2003, OpenSolaris, Red Hat Enterprise Linux, Debian GNU/Linux,
Ubuntu GNU/Linux, and many others). The instances are allocated on-demand and the payment is only for the compute capacity by hour. A complete price list can be found on the Amazon web site. Prices are relatively inexpensive, for example launching 100 instances in Europe for an Unix/Linux OS costs 11 USD per hour.

Java Library for Amazon EC2 Amazon has released a Java library for EC2 [15] that is suitable for our purpose. This library lets users interact with the Cloud, so it is possible, for example, to run instances, reboot it and terminate it with a simple method invocation. It is distributed under Apache License 2.0 and requires Java version 1.5.

2.4 Google Web Toolkit

The only programming language that the browser can understand is JavaScript. Writing JavaScript code is often difficult and errors can occur. Google Web Toolkit (GWT) [14, 17] allows users to build web interfaces based on AJAX and JavaScript components without the problem related to tricky and difficult maintenance of JavaScript code. With GWT it is possible to write a web front-end for any web application using the Java programming language; the toolkit will create an optimized JavaScript code supporting all common browsers. In the development phase it is possible to debug the application as a normal Java application. The deploy operation will compile the Java application into JavaScript code. The web interfaces classes can only use a limited set of packages from the Java standard library, that is logical since all the Java code we write will be translated into JavaScript with its limitations. For instance, GWT programs cannot deal with file system or with your operating system. GWT simplifies the communication between client and server thanks to a Remote Procedure Call (RPC) designed for GWT. GWT RPC is similar to Java RMI, so you can easily call methods from the server and get the results back without thinking about serialization and marshaling of the objects. Server code can use the whole Java standard library and further custom library.

GWT software is developed by Google and it is licensed under the
2.5 Ext GWT: Rich Internet Application Framework for GWT

A better graphical aspect for the web front-end can be obtained using an extension of GWT. Ext GWT [11] is one of the most powerful extension allowing you to build rich and customizable UI widgets.
Chapter 3
Architecture

In this chapter we discuss the project of the scalable execution environment based on a solution of both outsourced and distributed machines.

3.1 Overview

The whole simulator architecture consists of a series of interacting modules that are combined to cover all aspects of the work. To manage the various parts of the Simulation, the MDAlab of the Department of Computer and Information Science at Linköping University developed a simulation framework that integrates all part of the system, from the modeling tools to the reporting tools (Figure 3.1). As we can see in Figure 3.1 that shows the modularity of the simulator, the computational environment is actually a black box, that is not specified: it can be a single computer or a large complex system. Figure 3.2 shows the interaction among the modeling tool, the computational environment and the reporting tool. Data about the population, the disease and interventions are modeled using Protégé [21], an ontology-development environment. In particular Protégé-OWL, an extension of Protégé, is useful since it supports the Web Ontology Language (OWL) that is one of the standards of ontology languages. Population, disease and intervention are then generated with a Java extension of Protégé and represented in an intermediate specification format, XML files. These files define the mixing groups, their members and the models of diseases and interventions. The XML files are then passed to the
Figure 3.1. Pandemic Influenza Simulation User Interface

Figure 3.2. Interaction among the modeling tool, the computational environment and the reporting tool
3.1 Overview

simulator, that runs on the computational environment. The XML format was chosen because it is not platform dependent, and alternative simulation architectures could also be adopted using the same data. After the computation of the job, results are represented again in XML format which is later sent to the reporting tool. All of these parts can be managed by the Pandemic Influenza Simulation User Interface, with the exception of the computational environment.

The aim of this thesis is to develop a computation environment that is completely independent of other parts of the architecture; basically the goal is an environment that can be eventually substituted with another one, without having effects to other parts of the system. The environment is not also aware of the kind of application that runs on it, that means that it can be easily used for other simulation purposes.

The first feature of the project of the execution environment that we have to consider is parallelism, which is essential to exploit as much as possible the available resources. A single simulation task, as you can see in Figure 3.2, is essentially a batch process that does not require any kind of interaction with the end user: it takes an input XML file, processes all data and runs the simulation, and finally it produces an output XML file that must be collected with others. Every task is independent to the others, that means that an interaction among processes is not required. Above all of these consideration, we can see how the characteristic of the execution environment can be assumed to be a distributed batch system, in which every task is submitted to any of the available machines of the computational system, and a central or distributed data storage system is required to store all of the result files. As mentioned before, to have the required computational power at the lowest cost in terms of money, work and space, the best solution is built using parallel workers over Amazon Elastic Compute Cloud. The Amazon EC2 suits our needs, because it gives us the possibility to launch a variable number of instances paying only for the used capability. Having a large number of active instances, the simulation is completed quickly because all of jobs can run in parallel, and all of resources are fully-exploited. A distributed computational environment like this requires a good job management system that is able to allocate tasks to the available resources and take care about all mechanisms of the execution of them over remote machines: these characteristics can be found on a middleware. The best one that well
suit our needs is Condor. Since Condor is a specialized workload management system for compute-intensive jobs, it provides a job queuing systems, a scheduling policy, priority schemes, resource monitoring, and resource management that are necessary to control all parts of execution of jobs on remote machines. Users submit their jobs to Condor, then it places them into a queue, then it chooses when and where to run the jobs based upon a policy, carefully monitoring their progress, and ultimately it returns back the results. One of the features that is very important for us is that Condor is capable to manage heterogeneous computing resources, that means that parallel workers can also have different architecture types and operative system: this is a good feature since the Simulator core is developed to be multi platform.

The parallel computational architecture is then based of Condor workers over Amazon EC2 instances. The computational environment has two figures: the Condor Manager and Condor Workers on Amazon EC2 instances. The Manager has a job queue, submit tasks to the Workers that are the parallel executers of jobs. The complete architecture scheme is shown on Figure 3.6. A user interface was built to manage all parts of the computational architecture: users can launch
3.1 Overview

Figure 3.4. Parallel architecture based on Condor on Amazon EC2 instances

Figure 3.5. The architecture Layout
and stop Amazon EC2 instances to increase or decrease the number of active workers, submit jobs and watch their relative results. Users can control the status of the tasks execution on the Condor pool, eventually adding or removing capacity dynamically. After all the submitted jobs are completed, the user can switch off all instances, and the results are permanently saved on the storage system. The Condor Manager Machine is the central point of the architecture: it interacts with the user providing for that a web user interface, holds all instruments to manage the Amazon EC2 clouds and runs the Condor Manager which is the instance of Condor that keeps a job list and submits tasks to Condor workers on Amazon EC2. The Condor Manager interacts with a storage system to retrieve and store simulation data and their results. We choose to install the Condor Manager on a local machine, instead of creating a dedicated instance on Amazon EC2, for many reasons:

- The EC2 instances are stateless, in a sense that after their shutting down, all of modifies and data stored on them are lost. Instances are launched from the same base image, so it would be difficult to keep information about the jobs queue and the status when the Condor Manager is switched off.

- All the simulator XML input files should be uploaded to that instance, before it can be able to submit tasks to the Condor workers. It means that a transfer of a large amount of data is required before starting a job: this can be a problem if the database is local. There are no problems if the storage system is located on Amazon S3, but as we discuss in the section 3.3, the actual solution chooses a local storage system as a good compromise between costs and efficiency.

To avoid confusion, note that when we refer to the local machine, we call it Condor Manager Machine, while we call simply Condor Manager the instance of Condor that manages the Condor pool of workers, holds a jobs queue and submit them to the Condor Workers.

All of Condor workers are instances of the same base images, and they can be either GNU/Linux or MS Windows machines. It is also possible to launch at the same time both of two types of machines, since Condor supports heterogeneous machines and the Simulator is
multiplatform. When a job is submitted to the Condor Manager, it puts the tasks in his Condor jobs queue. Then the Manager submits one by one all tasks to the available workers, transferring the target platform executable and the input XML file. Condor workers perform the computation running the executable with the XML file as input and return back the output XML file, that is stored by the Condor manager machine. The flexibility of the Worker instances on Amazon EC2 is guaranteed by the fact that the simulator executable is not installed inside the Condor Worker Instance, but it is passed as parameter. In that way it is possible to substitute the executable with another version of it after an eventual update of the simulator without having to change it on the AMI, that would make necessary to bundle this one again. We can also note how the system is open to any kind of application, since we can supply whatever executable we want.

To grant a good level of security, all of the traffic is encrypted and there is a mutual authentication of the Condor Manager and Condor Workers. The next section focuses on Condor on Amazon EC2 architecture.

### 3.2 Condor on Amazon EC2 Instances

This section describes the configuration of Condor to be executed on Amazon EC2 machines [6], because that makes it simple to add computational resources dynamically and removing it after the needed usage time. Since Amazon EC2 can supply a large number of instances and Condor can manage an unlimited number of heterogeneous computational units, it is clear how scalable and flexible this architecture is. To have a working system based on Condor, it is necessary to have one or more Condor Managers, Condor Submitters and Condor Executeers. Our architecture is based on the idea that all the Condor executeers are Amazon EC2 instances, while the Condor Manager and the Condor Submitter are hosted at the same time in a local central machine. The Condor Manager Machine launches and stops instances by commands given on the web interface, collects all data about the Condor pool (Advertisement, ADS) and being a submit machine it holds the jobs queue and submit tasks to the Condor workers.

We first started installing and configuring the Condor Manager
Machine: at that issue it has been used a computer at our office, that has the good power to take that role, installing Debian GNU/Linux version 5.0 (Lenny). To install Condor for the role of Manager (but also Submit instance) it is necessary to pass the following arguments to the installation script:

```
--prefix=path_to_install_dir /
--type=manager,submit
```

The installation script installs all needed components for both roles, and the default configuration which is a good starting point for a well working system. To test the functionality of the Condor Manager without having to configure the Amazon EC2 workers, we simulated a computational cloud using many instances of a virtual machine on the local system: we created a Condor Worker installation on a Debian Lenny instance of VirtualBox, and then we simulated a network to allow all of the running instances to communicate with the Condor Manager. When the simulated network was ready, we tried to add new worker instances dynamically and submit jobs to them: everything worked fine and we had the opportunity to change some parameters
3.2 Condor on Amazon EC2 Instances

of the Condor configuration file in both Manager and Workers in order to understand how that behavior maximizes the performances. Since Condor is essentially conceived to exploit unused cpu time of office workstations, for example when the user does not input any character on the keyboard, our configuration differs from the default behavior, because we want that Condor workers are always active and the cpu should be used at all the time. So we could understand which are the parameters to turn on to maintain workers fully-operative.

```
START = TRUE
SUSPEND = FALSE
CONTINUE = TRUE
PREEMPT = FALSE
KILL = FALSE
WANT_VACATE = False
WANT_SUSPEND = True
SUSPEND_VANILLA = False
WANT_SUSPEND_VANILLA = True
STARTD_EXPRS = START
```

The simulated network was also a good way to test the security aspects of the network communication, because we had the possibility to understand which parameters were necessary to configure Condor to operate on a secure environment. Starting the communication turning off firewalls was a good way to the future implementation on Amazon EC2, since in this case the first problem was to solve the firewall restrictions and the NAT in which Amazon EC2 instances are behind to.

When all tests on the local simulated network were done they gave us all necessary information to change the Workers from the local virtual machines to Amazon EC2 Instances. We started to set up our private Amazon Image (AMI). Amazon give us the possibility to create an image from scratch or using a pre-build image and modifying it in order to bundle a new private image. We chose the second option, because Amazon makes available a basic image of Debian 5.0 Lenny, and if we had created a new from scratch we would be resulted in the same image. For the bundling of an image, see [1]. For our purpose we used two base images of Debian Lenny, one of 32 bit architecture and one of 64 bit.
Condor has been installed with the --type=execute flag, in order to have a worker instance.

The high flexibility and scalability of the architecture is thanks to this feature: we build a set of base AMIs, and all the instances that we are going to launch are clones of that images. If it is necessary to apply modifies or updates, it can be done on few minutes on the base images and all next instances will be updated at the same time.

Once the AMIs were built with Condor installed on them, the next part of the work was to find the optimal configuration of Condor to receive tasks from the Condor Manager.

The biggest problem for which we had to find a solution was the NAT and the firewall in which instances are behind to. Amazon EC2 instances have two DNS names: a private one, that is valid on the internal Amazon cloud only, and a public one that is valid everywhere. Instances do not have a public interface, so they know just their private DNS name, that corresponds to their hostname, and their private IP address. This is a problem when running Condor, because in this way when a Condor worker is placed inside the cloud, it sends advertisements to the Condor Manager that is outside the cloud by using its private DNS name and IP address, so replies from outside are not routable. To solve this problem it is necessary to use some parameters, that are not documented, that permit to indicate to Condor to send advertisements using its public names instead of private ones. First it is necessary to retrieve the public DNS name and IP address: this can be done using a simple RESTful interface and a query tool, provided by Amazon, that allow to retrieve many types of metadata directly from the Amazon EC2 instances[10]. Examples of metadata are the public/private IP address, public/private DNS names, arbitrary data passed to the instance by the user at launch time, the AMI-id, the instance type, etc/dots Using metadata, it is possible to retrieve all necessary information for a workaround of the NAT: on the Condor config file is possible to specify three special parameters that allow to indicate to Condor to use and publish the public IP address on advertisements instead of the private one. The three undocumented parameters are:

- \texttt{TCP\_FORWARDING\_HOST = public\_ip\_address}
- \texttt{PRIVATE\_NETWORK\_NAME = private\_network\_name}
- \texttt{PRIVATE\_NETWORK\_INTERFACE = private\_ip\_address}

The first specifies the public IP of the instance. The second and the
3.2 Condor on Amazon EC2 Instances

third specify the private DNS name and the private IP. In this way, when a Condor instance wants to reply to an incoming request from another one, it looks first its private network name: if it matches with the one of the other machine then the private IP address can be used, otherwise the public IP address is used. In our case, the private IP address is never used because there is not any interaction among Workers. The workers send advertisements to the Condor Manager; it looks at that parameters and determine that the target network is different from its own and then it sends replies directed to the public IP address of the instance.

Since more than one instance can be launched starting from the same base AMI, the configuration file can not be hard-coded. It is necessary to create a setup script that retrieval all necessary metadata, sets the three parameters, sets the public hostname and starts Condor. The startup script sets also many other parameters for secure authentication and encryption of data, as we discuss in 3.4.

To solve the problem of firewalls, the only way is to open the necessary ports on both kind of machines: on the Cloud it can be done using the command line tools from the Condor Manager Machine, and on that computer an option on its firewall is necessary to allow incoming requests. The configured port range for incoming connections on both workers and manager is from 40000 to 40050. It is also necessary to open the port number 9618 on the Manager which is used by the Workers to send advertisements. Other security parameters on Condor configuration files and on firewalls are tuned in order to allow only Condor Workers on Amazon EC2 to contact the Condor Manager and vice versa.

To allow the Condor Manager Machine to interact with the Amazon Cloud, it is necessary to install and configure the command line tools: operations like launching or stopping instances are submitted via the web user interface, but that operations are not executed by the interface, which has a web page on a browser running JavaScript code, but they are always executed by the Condor Manager Machine.

Jobs are submitted to the Condor Manager via the web interface, then the web server hosted on the Condor Manager Machine executes the condor_submit command to submit the job to the local instance of the Condor Manager. This command requires a description file of the job that gives to Condor all necessary information about the job name,
the executable to launch, the files to transfer, the working directory, the arguments to pass to the executable and other information to direct the queuing of all tasks of a job. An example of a condor submit file is:

```
# Job name: A09Job_All
executable = SimCore.exe
universe = vanilla
log = test1.log
requirements = Arch == "INTEL" &&
               (OpSys == "WINNT51" || OpSys == "WINNT60")
#task list
transfer_input_files=test1_A09_StayIndoorsA2B1_Asian-1.xml
arguments= -silent
           -file test1_A09_StayIndoorsA2B1_Asian-1.xml
queue

transfer_input_files=test1_A09_A2B1_Asian-1.xml
arguments= -silent
           -file test1_A09_A2B1_Asian-1.xml
queue
```

In this example we specify the executable name to transfer and launch on the worker, the architecture and o.s. requirements that the worker must have to be able to execute that job, then follows the specification of two tasks to put in queue, and for each one of them we specify the input XML file and other arguments.

Since our architecture allows the use of both Linux and Windows machine on different architectures, the submit description file must be built in order to be more general. In particular the selection of the binary executable to launch on the Condor Worker must be determined dynamically during the matching process of a task on an available machine. For this problem Condor provides a kind of parametrization of the submit file that allow to choose the executable after an available machine is chosen. This can be done using two macros $$\text{(OpSys)}$$ and $$\text{(Arch)}$$. An example of that kind of submit file is:

```
# Job name: A09Job_All
executable = SimCore.$$(OpSys).$$ (Arch)
```
3.2 Condor on Amazon EC2 Instances

```
universe = vanilla
log = test1.log

Requirements = (Arch == "INTEL" && OpSys == "LINUX") || \
    (Arch == "INTEL" && (OpSys == "WINNT51" || OpSys == "WINNT60" ))
when_to_transfer_output = ON_EXIT

# task list
transfer_input_files=test1_A09_StayIndoorsA2B1_Asian-1.xml
arguments=-silent
    -file test1_A09_StayIndoorsA2B1_Asian-1.xml
queue

transfer_input_files=test1_A09_A2B1_Asian-1.xml
arguments=-silent
    -file test1_A09_A2B1_Asian-1.xml
queue
```

In this case we specify as executable a macro that chooses the target binary executable for the particular architecture and operative system after an available machine is chosen. The submit file is now more general, since the requirements allows both MS Windows and GNU/Linux machines on Intel architecture.

Here is a summary of an interaction of the Condor Manager with Condor Workers on Amazon EC2:

1. The user, via the web interface, launches a desired number of instances.

2. Each instance:
   - Bootup the from the base AMI
   - Executes the Condor startup script, which sets the three workaround parameters, the public hostname and other security parameters.
   - Sends advertisements for authentication on the Condor Manager
   - Is ready to accept jobs
3. Simulation jobs are submitted by the user.

4. The Condor Manager submits tasks to the Condor Workers

5. Each Worker, iteratively:
   - Receives the input XML file and the Simulator executable
   - Runs the simulation
   - Gives back the output XML file
   - Is ready to accept another task

6. The Condor Manager Machine collects all results on a storage system

7. Simulation complete - user can stop instances or run another simulation

Figure 3.7 shows the interaction of the Condor Manager with a singular Condor Worker launched by the user.
3.2 Condor on Amazon EC2 Instances

Figure 3.7. Simulation on Condor Workers
3.3 Data Storage System

In this section we make some considerations about our choice of the storage system that well suits the needs of this application. In order to understand how to organize all data, we look first at the structure of the Simulation data. Every job is composed by a large number of scenarios: a scenario is a particular configuration of the three models of population, disease and intervention. In order to evaluate many possible cases and to find the best intervention, many scenarios have to be considered on the same simulation job, so it is necessary to generate different populations, diseases and interventions, and every scenario is a combination of three cases of these models. As explained before, there is randomness in population generation and in the simulation process: it means that every scenario must be generated and executed more than once to have considerable statistical results. For that reason, we say Jobs referring the whole simulation job, scenario for a simulation case in terms of models configuration, and we call task the running of the simulation for a single repetition of a scenario.

A job is submitted to the Condor Manager, and it submits in parallel the various tasks of the job to the Condor Workers. For a job that contains 20 scenarios, which are repeated 10 and 100 times, we have $20 \times 10 + 20 \times 100 = 2200$ XML files. Each XML file usually has a size of about 20MB: for this Job are required about 44GB of disk space. After these considerations, we can see that is required a lot of disk space to collect and store all data of all simulation jobs, and it makes the storage system another important part of the architecture, since the transfer of that amount of files can require a lot of time and the storage may have a considerable cost.

The possible solutions are two also in this case: to keep a local storage system or to outsource it like it has been done with computational capability using EC2. Amazon provides a storage system service called S3 (Simple Storage Service)[3]. As EC2, the access to that services is done with the command line tools or a RESTful interface, in this case to store and retrieve arbitrary kind of data. Each solution has advantages and disadvantages. The local storage system is a possible choice, since the Condor Manager Machine is local and it can easily interact with a local filesystem. It is necessary to provide a large storage capability, and a good organization of the data. The
problem of the local storage system is that the Manager becomes the bottleneck of the architecture: when submitting a job, the Condor Manager transfers the executable of the simulator, that occupies few kilobytes, and the XML input file that has a size of 20MB: if there are for example 20 available instances, and 20 tasks are submitted at the same time by the manager, an upload traffic of 400MB is required. It means that is necessary more time for the file transfer than for the real execution of simulation tasks on Condor Workers. The other solution, using Amazon S3, solves this problem, because the storage system S3 is inside the cloud, then the transfer of the input files from S3 to the EC2 instances is faster and does not involve the Manager. But of course the data must be transferred to S3 first: it takes at least the same transfer time as for a submit of a job, but also if the transfer from S3 to the Instance is fast, it increments to global time necessary to transfer the file to the final Condor Worker.

Amazon charges an amount of money for each Gigabyte of data transferred from and to the Cloud, which means that both solutions have a cost for the used bandwidth, but the second solution is more expensive since Amazon also charges for the amount of stored data in S3. The traffic between instances inside the cloud is not charged, even the file transfers from S3 to Condor Workers. The second solution has however the advantage that Amazon provides security of the data and fault tolerance, so we don’t have to take care about these critical aspects of the system.

The simulation data is generated by the user on his machine with the modeling tools, an in both cases it is necessary to transfer all that amount of files to the target storage system. In case of use of Amazon S3, uploading a whole job data is not a trivial task: it takes much time to transfer for example 44GB like in the previous case. Although it is necessary to transfer that data to the local file system, in case of use that solution, it is obviously faster that S3 since the storage system is inside the network of the University.

We also noted that the transfer speed of files from the Condor Manager Machine to a Condor Worker is faster if we do it directly by Condor or with secure copy (SCP); the transfer rate from the local PC to an S3 bucket instead is very slow. So if we consider that if we use S3 as storage system, the transfer from the S3 bucket to the Condor Worker inside the cloud is also required, the total time spent
in file transfer is more than if we use a local storage system. The

<table>
<thead>
<tr>
<th>Program</th>
<th>Attempt</th>
<th>Time</th>
<th>Max Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>scp</td>
<td>1</td>
<td>0m3.286s</td>
<td>10.2MB/s</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0m3.256s</td>
<td>10.2MB/s</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0m3.259s</td>
<td>10.2MB/s</td>
</tr>
<tr>
<td>S3cmd put</td>
<td>1</td>
<td>0m20.916s</td>
<td>1001.87 kB/s</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0m17.892s</td>
<td>1172.28 kB/s</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0m20.493s</td>
<td>1023.94 kB/s</td>
</tr>
</tbody>
</table>

Table 3.1. Transfer speed comparison of files to EC2 Instances and to S3 buckets

Table 3.1 shows that even in the same network, the same PC and the same 21MB XML file, the transfer rate on an Amazon S3 bucket is about ten times slower than the transfer rate on the same condition but directly on an Amazon EC2 instance using scp. For these reasons, like the costs for transfer and store on S3, the slower transfer rate and a major difficulty to manage files (since S3 use the concept of bucket, that is a generic container of data) we preferred to adopt the solution based on the local storage system.

In that case however the problem to solve is the bottleneck on the Condor Manager in the submit phase of tasks to the Condor Workers.

A solution for this problem can be found using compression of data: since XML files are text files, a compression technique can significantly reduce the size of that files. Using compression, we gain the following benefits:

- The amount of transferred data is reduced, which means that each transfer takes less time, and more than one transfer can be done simultaneously without deteriorating the performance of the system

- The occupied space on the storage system is strongly reduced.

Note that the compression of files is also applicable to the solution using S3, but the rest of the problems still remain.
3.3 Data Storage System

The compression of data can be done after generating it by the modeling tool or directly from it, so data can be stored on the storage system directly compressed, since it is not necessary to look at the content of these files from the web application. When the Condor Manager submits a task, it transfers to the Worker the simulator executable and the compressed file. The Condor Worker has the task to decompress first the XML file before passing it as input to the simulator executable. In that way the time spent in file transfer is strongly reduced, with the overhead of a less considerable time on the Condor Worker for decompression.

An evaluation of different compression techniques can be useful to find which one is balanced in terms of compressing/decompressing time and space saving. We tried four compression tools, that have different features:

1. *zip/gzip* are the most used tools for compression. They are fast, but the compression rate is not high.

2. *lzop* is the fastest in terms of compression and decompression speed, with a lower compression ratio than others.

3. *bzip2* has a lower compression speed, but has interesting compression ratios.
4. 7zip is the file archiver with the highest compression ratio, but it is also the slowest.

Here we have some statistics on compressing and decompressing time of the same 21MB XML file on the same computer (the condor Manager, a Pentium D 3.0Ghz running Debian Linux 5.0) with this four different compression tools, with the default behavior:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Compressed</th>
<th>Comp. time</th>
<th>Dec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lzop</td>
<td>3.8MB</td>
<td>0m0.143s</td>
<td>0m0.124s</td>
</tr>
<tr>
<td>gzip</td>
<td>2.1MB</td>
<td>0m1.205s</td>
<td>0m0.231s</td>
</tr>
<tr>
<td>zip</td>
<td>2.1MB</td>
<td>0m1.525s</td>
<td>0m0.183s</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.6MB</td>
<td>0m8.252s</td>
<td>0m1.203s</td>
</tr>
<tr>
<td>7z</td>
<td>1.2MB</td>
<td>0m10.826s</td>
<td>0m0.456s</td>
</tr>
</tbody>
</table>

Table 3.2. Comparison of compression techniques

As compromise of compressing time, decompressing time, saved space and portability our choice falls on the classical Zip tools. It has a good performance for compressing and decompressing, that is important to transfer the file in a short time and run the simulation without spending much time on decompression overheads. Using this format it is also possible to generate and compress data directly from the Java modeling tool, since there are free compression libraries available for Java [5].

Using compression, the condor submit file differs from before because now it is necessary to decompress the file on the condor worker before it starts the simulation. It means that we can not pass directly the binary of the simulator, instead we have to pass a custom script for the target machine that decompress the archive first and then launch the simulation.

```plaintext
# Job name: A09Job_All
executable=SimLauncher.$(OpSys).$(Arch)
universe=vanilla
log=test1.log
Requirements =
   (Arch="X86_64" & OpSys="LINUX") || \
```
(Arch=="INTEL" && OpSys=="LINUX") || \
(Arch == "INTEL" && (OpSys =="WINNT51" ||
OpSys== "WINNT60" ))

when_to_transfer_output = ON_EXIT

transfer_input_files = simbin.zip,
    test1_A09_StayIndoorsA2B1_Asian-1.xml.zip
arguments = test1_A09_StayIndoorsA2B1_Asian-1.xml.zip
    -silent
queue

transfer_input_files = simbin.zip,
    test1_A09_A2B1_Asian-1.xml.zip
arguments = test1_A09_A2B1_Asian-1.xml.zip
    -silent
queue

In that case in the executable field we put a name of a script, and all binary executables are passed compressed as input file. We cannot use macros on the transfer_input_files parameter, so it is necessary to pass the archive of all executables, and the script on the worker decompress it and launches the appropriate one. For Linux machines on i386 architectures, the bash script is called SimLauncher.LINUX.INTEL:

#!/bin/sh
#$1 input XML file
#$2 arguments
unzip $1 1>/dev/null
unzip simbin.zip 1>/dev/null
./SimCore -file ‘echo $1|cut -d’.’ -f1‘.xml $2
exit 0

For Linux machine on x86_64 architectures the script is called SimLauncher.LINUX.X86_64:

#!/bin/sh
#$1 input XML file
#$2 arguments
unzip $1 1>/dev/null
unzip simbin.zip 1>/dev/null
In that case the 64 bit compiled executable is used.

For the simulation jobs, a database is necessary to keep information about the job names, the number of tasks and if the job has been simulated or not. The physical upload is done transferring data to the target position of the filesystem, and the registration of the data through the web interface is required to add the job in the system and starting with its simulation. The output files, after the simulation, will be stored on the same directory of the input files: in that case a compression is not longer necessary since every output file takes up few Kilobytes. Actually the storage system is an hard drive inside the Condor Manager Machine; to improve security of data a network storage system would be a better solution, allowing users to access and upload data with all the necessary security policies.
3.4 Security

On a distributed system the security is one of the most important targets. Condor architecture requires the Condor Workers to frequently communicate with the Manager in order to keep updated the status of Condor pool, and in addition very sensible data need to be transferred between the Condor Manager and Condor workers, like the XML simulation input file and the simulator executable. All this traffic transits on Internet and the communication over it is definitely insecure; all data that transit on this network is exposed to be read by anyone using "Packet sniffing" but also the services are potentially vulnerable to different types of attacks like "Denial of service" or "Man in the middle". One possible solution that can ensure the security in terms of authentication, confidentiality and integrity is SSL with mutual authentication. It works below the application layer providing a secure channel between the transport layers of the two communicating hosts. It is based on the asymmetric key cryptography (DSA or RSA) and it is very popular in particular in the web-browsing for the server authentication and for the data encryption. Security is obtained using X.509 certificates signed by a trusted Certification Authority (CA) and it is based on the private-public key pair, so this protocol requires an infrastructure that manages the public keys (Public Key Infrastructure, PKI). Another equally secure protocol is SSH; it also uses public-private key pair but without X.509 certificate and certification authority, so it does not require a PKI. With SSH the authentication must be mutual (while in SSL it is optional) and, since there are not trusted authorities, every host should keep a list of known hosts. Even though SSH was originally created to replace insecure telnet and other remote shell connections (like rlogin), it can be used for tunneling un-encrypted traffic and for many other purpose like secure file transfer, VPN, X11 forwarding, SVN, etc... 

In our architecture, we need some security protocols when the Condor collector (the part of the Manager which receives advertisements and manage the pool) have to decide whether or not accept an incoming connection from a worker. We also want our traffic to be encrypted to ensure that nobody can read it. SSH tunneling can be one possible solution, since it can create secure channels between the Condor Manager and a Condor Worker. The main problem of this approach is
that tunnels must be initialized statically since Condor can choose dynamically one of the ports of the fixed range as destination port, and we cannot create a tunnel on-demand: it means that in our case we have to create a tunnel for each port of the range starting from 40000 to 40050. This high number of active ssh tunnels can affect the performance of the Worker, but mainly the problem is on the Manager, which have to keep that big amount of active tunnels for each instance of the Condor pool. This is not a practical solution, a better solution would be using the SSL support in Condor. SSL uses almost the same cryptography algorithms of SSH for authentication and encryption of the traffic, but it doesn’t require to start statically the tunnel for each port, since it can be created dynamically directly by Condor, which is created with a complete support for SSL authentication and encryption. Security in Condor is a critical aspect and it is developed using the concept of access level. A Condor user can execute a command only if it is authorized at the access level of that command. Since Condor allows a submitter to execute code potentially malicious, all the machines that enter into the Condor pool must be checked and authenticated.

When a new Amazon EC2 instance is launched, it is necessary to create the set of key pair to allow the worker to authenticate itself to the Condor Manager.

The authentication is obtained using SSL and a local certification authority, running on the Condor Manager Machine, that creates and signs the certificates for the Manager once and for the Workers whenever a new one is launched. The CA private key, used for signing client certificates by our local certification authority, is stored on the local hard drive. In this way we have our own private PKI that is enough for our purpose.

Before a new EC2 instance is launched, a bash shell script on the Condor Manager Machine using `openssl` generates a new certificate (with the corresponding private key) for the Condor worker. There are 3 steps to follow to create and sign a new certificate: first a new key is created, then a certificate signing request (CSR) is sent to the certification authority, and finally it creates the certificate signing the CSR.

1. `openssl genrsa -out file.key 1024`
2. `openssl req -new -key file.key -out file.csr -config ssl/openssl.conf`

3. `openssl ca -batch -config ssl/openssl.conf -in file.csr -out file.crt`

The command `openssl genrsa` creates an RSA private key, and the parameter indicates the number of bit of the key, 1024 in this case. The key should be encrypted but we decided not to do it because this key is stored locally and it is sent to the Condor worker using a secure channel, as we explain later in the text.

The real complete procedure for the creation of a signed certificate is explained below. With `openssl req` a Certificate Signing Request is created by the user; a CSR is a text file that contains information about the user (Distinguished name) and other details about the certificate like the key usage (only for X.509 v3). The CSR also contains the public key that will be embedded on the certificate after that a certification authority signs it. Briefly a CSR can be considered as an unsigned certificate.

The CSR should be signed by the authority; to obtain a signed certificate, by an official trusted authority, the user usually has to pay a certain amount of money and he needs to demonstrate accurately his identity. Into a PKI, the Registration Authority (RA) establishes the binding between the key and the certificate; so the user should send his personal data and his public key (into a CSR) to the RA, which must verify and approve that CSR and ask to the CA for the signature.

We do not need to have certificate signed and recognized into a PKI, we just want a certificate signed by our local authority so that we can be able to check the identity of that subject and the validity of its certificate. It is easy to sign a CSR using `openssl ca`, we just specify the input CSR and the key of the authority to sign with. The output of this command is a certificate, signed by our local fictitious CA. This certificate and the correspondent key can be used to establish an SSL connection between two hosts and in this case, the hosts are the Condor worker and the Condor manager. The Condor manager also has a certificate signed by the same CA and it uses the corresponding key to mutual authenticate it with the worker. The Condor collector is configured to accept only workers that have a certificate signed by our CA whose certificate is used to verify the
Figure 3.9. The basic interaction for signing a certificate.
validity of certificates of the workers.

When the user starts an Amazon EC2 instances, it is necessary to transfer these files to them that grant the permission to access our Condor pool. The transfer of that file can be done at the launch time by the `ec2_run_instance` command of the EC2 API tools, which allows to pass to the instances a file that they can afterwards retrieve as metadata. Since the command uses secure connection (the web service as default uses HTTPS), we do not care about encryption of this archive containing the private key and the signed certificate. Amazon EC2 makes this file available to all the instances within the same reservation id.

This archive contains:

- The CA certificate.
• The generated and signed certificate of the worker.
• The corresponding private key.
• A text file that contains the public hostname of the Condor manager.

On the Condor worker side, the starting script needs just to get the transferred archive using the default mechanism to retrieve metadata

```
wget http://169.254.169.254/1.0/user-data -O ssl.tar
```

to extract the files from the archive

```
tar -xzf ssl.tar
```

and then to copy the extracted files in the correct path defined in the Condor configuration file.

Here is how to configure the client to work using SSL, this is a portion of the configuration file on the Condor worker.

```
AUTH_SSL_SERVER_CAFILE=CONDOR_HOME/etc/ssl/ca.crt
AUTH_SSL_CLIENT_CAFILE=CONDOR_HOME/etc/ssl/ca.crt
AUTH_SSL_SERVER_CERTFILE=CONDOR_HOME/etc/ssl/worker.crt
AUTH_SSL_SERVER_KEYFILE=CONDOR_HOME/etc/ssl/keys/worker.key
AUTH_SSL_CLIENT_CERTFILE=CONDOR_HOME/etc/ssl/worker.crt
AUTH_SSL_CLIENT_KEYFILE=CONDOR_HOME/etc/ssl/keys/worker.key
SEC_DEFAULT_ENCRYPTION = REQUIRED
SEC_DEFAULT_INTEGRITY = REQUIRED
SEC_DEFAULT_NEGOTIATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION_METHODS = FS, SSL
SEC_DEFAULT_INTEGRITY_METHODS = MD5
SEC_DEFAULT_ENCRYPTION_METHODS = 3DES, BLOWFISH
```

In this configuration, all the security features are required. That is the most restrictive configuration because the Condor manager denies authentication if the worker has no credential and forbids any other kind of communication that is not encrypted both for reading or writing.

The methods specify the algorithms for that features. Once all these operation are completed and all of the files needed by Condor
for authentication are in the correct location, the Condor worker can authenticate itself on the manager and start its workcycle.
3.5 Performance evaluation

In this chapter, we will discuss about the simulator performance which is measured on different types of Amazon EC2 machines and also on some office machines, to evaluate the difference of performance.

We will always refer to the simulator compiled with the $O2$ optimization option for the GNU Linux compiler $g++$ (if not differently specified).

The tests consist of measuring the execution time of the simulation process. We used the same input file for all of the tests and the same version of the simulator.

3.5.1 Amazon EC2 performance

In this section we discuss about the different performance of some Amazon EC2 instance types:

- Amazon EC2 Small instance.
- Amazon EC2 Large instance.
- Amazon EC2 High CPU Medium instance.
- Amazon EC2 High CPU Extra Large instance.

In Table 3.3 we summarize the main features of the Amazon EC2 instance types. For a more detailed description see the Amazon website. We used two base images, a 32 bit one and a 64 bit one, running both Debian GNU/Linux 5.0 (lenny). The simulator executable was compiled for the 64 bit architecture using the $g++$ option $-m64$.

Small instance

Table 3.4 shows some results of tests on a Small Amazon instance. It is clear that there is a big difference between the real time and the user time. That is because the single CPU cannot be used at 100%; half of its time is stolen by the virtual machine emulating the instance. We can also note that, however, it is necessary a lot of time to complete the computation.
3.5 Performance evaluation

<table>
<thead>
<tr>
<th>Resource</th>
<th>Small</th>
<th>Large</th>
<th>H. CPU Medium</th>
<th>H. CPU X Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>32 bit x86</td>
<td>64 bit x86</td>
<td>32 bit x86</td>
<td>64 bit x86</td>
</tr>
<tr>
<td>CPU</td>
<td>1 ECU, 1 core</td>
<td>4 ECU, 2 cores</td>
<td>5 ECU, 2 cores</td>
<td>20 ECU, 8 cores</td>
</tr>
<tr>
<td>Memory</td>
<td>1.7 GB</td>
<td>7.5 GB</td>
<td>1.7 GB</td>
<td>7 GB</td>
</tr>
<tr>
<td>Price</td>
<td>$0.11</td>
<td>$0.44</td>
<td>$0.22</td>
<td>$0.88</td>
</tr>
</tbody>
</table>

Table 3.3. Amazon EC2 instance type CPU description. The price is per hour and it is the fare for Europe.

<table>
<thead>
<tr>
<th>Repetition</th>
<th>Real</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6m43.989s</td>
<td>2m53.375s</td>
</tr>
<tr>
<td>2</td>
<td>6m7.061s</td>
<td>2m37.894s</td>
</tr>
<tr>
<td>3</td>
<td>6m39.709s</td>
<td>2m51.647s</td>
</tr>
<tr>
<td>Mean</td>
<td>6m30.253s</td>
<td>2m47.639s</td>
</tr>
</tbody>
</table>

Table 3.4. Execution time of the simulator on a Amazon EC2 Small instance. The test was repeated three times.

Large Instance

The Amazon EC2 Large instance has a 64 bit platform. In this case, we tested 2 parallel simulation to exploit the 2 virtual cores. We did not perform the test on the Extra Large instance, since it provides proportionally more disk space and memory than CPU performances. In that case it would have been better the use of the High-CPU Extra Large Instance.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Real</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3m22.816s</td>
<td>1m58.867s</td>
</tr>
<tr>
<td>2</td>
<td>3m21.081s</td>
<td>2m15.856s</td>
</tr>
<tr>
<td>Mean</td>
<td>3m21.948s</td>
<td>1m07.361s</td>
</tr>
</tbody>
</table>

Table 3.5. Execution time of the simulator on a Amazon EC2 Large instance.
High CPU Medium Instance

The Amazon EC2 High CPU Medium instance has a 32 bit platform. Also in this case, we tested 2 parallel simulation to exploit the 2 virtual cores. This instance type has 5 Amazon compute units so the performance are logically better than the Large instance even if they have the same number of cores.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Real</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2m02.257s</td>
<td>1m56.715s</td>
</tr>
<tr>
<td>2</td>
<td>2m05.332s</td>
<td>1m57.063</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td><strong>2m03.794s</strong></td>
<td><strong>1m56.889s</strong></td>
</tr>
</tbody>
</table>

Table 3.6. Execution time of the simulator on a Amazon EC2 High CPU Medium instance.

High CPU Extra Large Instance

In Table 3.7, the Amazon EC2 High CPU Medium instance performance are shown. It has a 64 bit platform and 20 ECUs with 8 virtual cores. We tested 8 parallel simulations.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Real</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6m03.004s</td>
<td>6m02.556s</td>
</tr>
<tr>
<td>2</td>
<td>6m03.996s</td>
<td>6m03.511s</td>
</tr>
<tr>
<td>3</td>
<td>6m05.649s</td>
<td>6m05.113s</td>
</tr>
<tr>
<td>4</td>
<td>6m05.256s</td>
<td>6m04.759s</td>
</tr>
<tr>
<td>5</td>
<td>6m04.488s</td>
<td>6m02.836s</td>
</tr>
<tr>
<td>6</td>
<td>6m05.718s</td>
<td>6m05.072s</td>
</tr>
<tr>
<td>7</td>
<td>6m05.145s</td>
<td>6m04.402s</td>
</tr>
<tr>
<td>8</td>
<td>6m05.231s</td>
<td>6m03.537s</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td><strong>6m04.810s</strong></td>
<td><strong>6m3.973s</strong></td>
</tr>
</tbody>
</table>

Table 3.7. Execution time of parallel simulations on a Amazon EC2 High CPU Extra Large instance.
3.5 Performance evaluation

3.5.2 Office machines tests

In our performances evaluation tests, we also tested some real machines. We find out that in same cases the performance of a small office machine are better than the virtual machines of Amazon.

- Netbook Asus eeepc with Ubuntu-eee 8.04
- Workstation with a *Pentium® D* dual core with Debian (Lenny)

**Netbook Asus eeepc**

This machine has a single core Intel® Celeron® M processor 900MHz (see Table 3.8). The operating system is Ubuntu-eee 8.04. We can see how this apparently low performance CPU give us better performance than the Small instance on Amazon, even if Amazon declares that a computational unit (ECU) is equivalent to $1.0 - 1.2GHz$ 2007 Opteron or 2007 Xeon processor.

<table>
<thead>
<tr>
<th>Vendor ID</th>
<th>GenuineIntel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>13</td>
</tr>
<tr>
<td>Model name</td>
<td>Intel® Celeron® M processor 900MHz</td>
</tr>
<tr>
<td>Stepping</td>
<td>8</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>900</td>
</tr>
<tr>
<td>Cache size</td>
<td>512 KB</td>
</tr>
<tr>
<td>Flags</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat clflush dts acpi mmx fxsr sse sse2 ss tm pbe nx up bts</td>
</tr>
<tr>
<td>Clflush size</td>
<td>64</td>
</tr>
<tr>
<td>Memory size</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

**Table 3.8.** CPU description of the EEEPC.
### Pentium D

This machine is used to host the Condor Manager. It has a dual core Intel® Pentium® D (see Table 3.10). The operating system is Debian GNU/Linux 5.0 (*Lenny*). We simulated 2 parallel tasks to exploit the 2 cores. The performance are very good and actually it is the best compared to the others.

<table>
<thead>
<tr>
<th>Vendor ID</th>
<th>GenuineIntel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU family</td>
<td>15</td>
</tr>
<tr>
<td>Model</td>
<td>6</td>
</tr>
<tr>
<td>Model name</td>
<td>Intel® Pentium® D CPU 3.40GHz</td>
</tr>
<tr>
<td>Stepping</td>
<td>4</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>3391.718</td>
</tr>
<tr>
<td>Cache size</td>
<td>2048 KB</td>
</tr>
<tr>
<td>Flags</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc pebs bts pni monitor ds_cpl vmx est cid cx16 xpr lahf_lm</td>
</tr>
<tr>
<td>Clflush size</td>
<td>64</td>
</tr>
<tr>
<td>Memory size</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

**Table 3.10.** CPU description of the Intel® Pentium® D.
3.5 Performance evaluation

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Real</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2m1s</td>
<td>1m58s</td>
</tr>
<tr>
<td>2</td>
<td>2m2s</td>
<td>1m59s</td>
</tr>
<tr>
<td>Mean</td>
<td>2m1.5s</td>
<td>1m58.5s</td>
</tr>
</tbody>
</table>

Table 3.11. Performance of the simulator on a Pentium® D.

3.5.3 Considerations

The results previously presented show the performance of Amazon EC2 instances and of some office machines. This evaluation was made launching processes and getting their execution times, so all results are not calculated with absolute precision. However, they can give us a good overview to compare the performance in order to understand which kind of instance gives us a good balance of performance and costs. As we can see in the tables, the Small instance is the cheapest, but the performance is very low. Even if it has a relative fast user time, the real total time is not so good. The Large instance, which has 2 computational units, has better performance than the Small one, but it costs four times more. The High-CPU Medium instance has an interesting execution time considering its low cost per hour. The High-CPU Extra Large was a sort of disappointment for us, since the high number of cores can only have singularly the performance of a small instance: in that case would it be better to launch eight Small instances instead?

Table 3.12 shows the execution time on the architectures we tested. For the multi cores architectures, we report the average execution time of the parallel simulations (we run a parallel simulation for each core).

If we relate the real execution time shown in Table 3.12 to the number of tasks done, we can evaluate the effective performance of the machines. We used the following criterion to obtain a performance measure for each instance type:

\[
\text{performance} = \frac{\text{number of cores}}{\text{execution time}}
\]

In Figure 3.11 we can see the improvement of the performance that is linearly related to the instance type.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Real</th>
<th>User</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asus eeepc</td>
<td>5m13s</td>
<td>5m08s</td>
<td>1</td>
</tr>
<tr>
<td>Pentium D</td>
<td>2m1s</td>
<td>1m58s</td>
<td>2</td>
</tr>
<tr>
<td>Amazon EC2 Small</td>
<td>6m30s</td>
<td>2m48s</td>
<td>1</td>
</tr>
<tr>
<td>Amazon EC2 Large</td>
<td>3m22s</td>
<td>1m07s</td>
<td>2</td>
</tr>
<tr>
<td>Amazon EC2 H.CPU Medium</td>
<td>2m4s</td>
<td>1m57s</td>
<td>2</td>
</tr>
<tr>
<td>Amazon EC2 H.CPU Extra Large</td>
<td>6m5s</td>
<td>6m4s</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.12. Execution time summary of the simulator on tested machines.

We can also consider the number of ECUs (see 2.3 and Table 3.3) per each cores as declared by Amazon and weight the performance with this value. We used the following ratio:

$$\text{performance}_{\text{ECU}} = \frac{\text{performance}}{\text{ECUs}}$$

In this case (Figure 3.12) we can notice an equivalence among the instance types except for the High CPU Extra Large instance that
3.5 Performance evaluation

Figure 3.12. Amazon EC2 instance types performance related to the cores number and to the ECUs number.

officially has 20 ECUs but in practice basing on its performance, it seems to have at least 10 ECUs.

But if we also consider the cost of each instance (Figure 3.13), we can see an interesting exploit of the High CPU Medium instance. This can be explained considering the low price (see Table 3.3) that can be justified by the low amount of memory provided by this instance type. For our purpose this is not a problem because 1.7 GB of memory is enough for the simulator, which needs more CPU capacity than memory space.
3.5.4 OS comparison on a Pentium® 4

This machine has a dual core Mobile Intel® Pentium® 4 CPU 3.06GHz. In this case, we compared two operating system running the same simulation: Debian 5.0.1 (Lenny) and Microsoft® Windows XP SP3. The simulator was compiled using g++ 4.3.2 for the Linux version and Microsoft® Visual Studio® 9.0 for the Windows version.

Using Microsoft® Visual Studio® with the following optimization options:

Inline Function Expansion  Default
Enable Intrinsic Functions  Yes
Favor Size or Speed  Neither
Omit Frame Pointers  No
Enable Fiber-safe Optimization  No
Whole Program Optimization  Enable link-time code generation
3.5 Performance evaluation

<table>
<thead>
<tr>
<th>Vendor ID</th>
<th>GenuineIntel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU family</td>
<td>15</td>
</tr>
<tr>
<td>Model</td>
<td>4</td>
</tr>
<tr>
<td>Model name</td>
<td>Mobile Intel® Pentium® 4 CPU 3.06GHz</td>
</tr>
<tr>
<td>Stepping</td>
<td>1</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1862.000</td>
</tr>
<tr>
<td>Cache size</td>
<td>1024 KB</td>
</tr>
<tr>
<td>Flags</td>
<td>fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe constant_tsc pebs bts pni monitor ds_cpl est tm2 cid xtpr</td>
</tr>
<tr>
<td>Clflush size</td>
<td>64</td>
</tr>
<tr>
<td>Memory size</td>
<td>512 MB</td>
</tr>
</tbody>
</table>

Table 3.13. CPU description of the Intel® Pentium® 4.

and varying the parameter **Optimization**, we obtain the results shown in Table 3.14.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Bin size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>165 KB</td>
<td>17m10s</td>
</tr>
<tr>
<td>Minimize Size</td>
<td>98 KB</td>
<td>4m0s</td>
</tr>
<tr>
<td>Maximize Speed</td>
<td>122 KB</td>
<td>2m44s</td>
</tr>
<tr>
<td>Full</td>
<td>123 KB</td>
<td>2m45s</td>
</tr>
</tbody>
</table>

Table 3.14. Execution time and size of the simulator on a Pentium® 4 and MS Windows.

On the other side, the `g++` compiler has many flags for the optimization and it is hard to compare the two compiler varying the same parameter so, in this comparison we just limited to test the different optimization level even if there is not perfect correspondence between these level on the `g++` and on the Visual Studio®. Table 3.15 shows the results of a simulation using a different flag for the optimization. For the meaning of these flag we refer to the `g++` manual [12].

In any case, we can remark that the performance obtained using
<table>
<thead>
<tr>
<th>Optimization</th>
<th>Bin size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>O0</td>
<td>315 KB</td>
<td>6m19s</td>
</tr>
<tr>
<td>Os</td>
<td>146 KB</td>
<td>2m48s</td>
</tr>
<tr>
<td>O1</td>
<td>194 KB</td>
<td>2m25s</td>
</tr>
<tr>
<td>02</td>
<td>197 KB</td>
<td>2m24s</td>
</tr>
<tr>
<td>03</td>
<td>215 KB</td>
<td>2m23s</td>
</tr>
</tbody>
</table>

Table 3.15. Execution time and size of the simulator on a Pentium® 4 and Linux.

The g++ compiler is better than obtained using Visual Studio compiler under Windows at the expense of a major binary size.
Chapter 4

The web interface

Human-computer interaction is a critical aspect of an information system. The system should understand what the user wants to do and the user must exactly knows what he is going to do. The user and the system can interact in several ways; a remote terminal logged on the server may be a fast and powerful instrument to control the system but it requires a deep knowledge of the architecture and of the command line tool. Internet is getting more and more widespread which means that it is possible to find a web browser in almost all portable devices such as smartphones or PDAs. The development of the Internet, together with the related technologies, shows us a new prospective about the use of the computer network. The client-server interaction is now more easy and responsive then even before; the client application is fully integrated in the web browser the user prefers. The usability of this kind of application is limited by the loading wait caused by synchronous requests to the server. The development of JavaScript asynchronous calls to the server improves the responsiveness and the usability of the web interfaces. In addition, moving some logic of the application into the client-side reduces the workload on the server. That is thanks to the Web 2.0 and AJAX [23]: using these technologies the browser is no longer a mere HTML interpreter but a real front-end application to the server.

Technically, web browsers can only execute JavaScript code, so every component of the client application must be developed in this language. Writing JavaScript code, in order to obtain interactive and good-looking web pages, can be tricky and hard to manage. However,
there are many tools that automatically generate JavaScript code. GWT (see 2.4) is one of the most popular and widespread. By using this tool, it is possible to create web applications that can be considered like normal desktop applications and they can be easily integrated with the most common server side technologies like basic HTTP request using plain text, but also XML and JSON format for complex data type. In addition, GWT has an integrated high level mechanism for invoking methods on the server: GWT-RPC. This technology is similar to JavaRMI and uses asynchronous calls to the server. The client just knows the signature of the methods it can invoke, while the server class, that implements the methods, is deployed as a servlet and it can be published in a servlet container like Tomcat (see Appendix B).

4.1 Overview

In this work, we have developed a web front-end and a full featured web application that allows the user to interact with the server and with the underlying architecture.

After the user logs in to the system, he can interact with his AWS account. So he can see his personal AMIs and he can decide to launch one or more instances of them. Once launched, he can monitor the list of the running instances and their actual status. All the instances can be rebooted or shut-down with a simple click on a button of the web interface. Another part of the application allows the user to manage Condor jobs and to monitor the status of the workers collected by the Condor manager. Finally, the web application allows to see the results of the simulations and of the single scenario in a pretty and clear way.

The interface is organized in tabs to separate the different tasks the users want to accomplish.

4.2 The environment

Google™ provides some tools to start a web application using GWT framework and to develop it using Eclipse IDE (http://www.eclipse.org). These tools, once installed in the system, can be also used to test the application and Eclipse is very useful for debugging both client and server side code. GWT applications can be run in hosted mode and
4.3 Amazon management

web mode. The difference between these two modes is essentially that
using the first one, the application is quickly compiled and the com-
pilation produces Java bytecode, so the application can be executed
only on the GWT’s hosted mode browser. This mode allows also to
debug the client and server side code using Eclipse. With the web
mode, instead, GWT translate the Java source code into optimized
JavaScript. So in host mode, the Java virtual machine executes the
code while in web mode, the client side code is interpreted by any web
browser JavaScript compatible.

4.2.1 Support tools

The application needs to store information about the AWS account of
the user. In particular the private key and the certificate of the user
are necessary to run instances using the tools provided by Amazon.
For privacy reasons, the private key must be encrypted. To encrypt
this file (and to decrypt it), the openssl tool was used.

The application working directory contains the user certificates,
the simulator executables and the script used for launching instances.
In addition, this directory contains all the necessary files for creating
and signing certificates to pass to the workers to authenticate itself
(see 3.4).

The system uses a MySQL database for the storage of the account
information and the description of the jobs. The database server is
located on the same machine in which runs Condor manager and Tom-
cat. It contains 2 tables, for the users and for the jobs.

4.3 Amazon management

Amazon provides a command-line toolkit for managing the AMIs and
the instances. Amazon also provides a Java library that allows full
control of the instances. In our application we used this library in
addition to the command-line tool. Even if the Java library for Amaz-
on EC2 is enough for the common use, for our purpose, it misses
something that we will explain later.

From the Amazon Management tab the logged in user can see the
list of his registered AMIs and the list of his running instances. These
lists are refreshed periodically in background doing asynchronous calls
Figure 4.1. The web interface of the Amazon management.
4.3 Amazon management

Figure 4.2. The prompt for launching instances.

to the server. Using this mechanism, when the client makes an invocation to a remote method, it also declares a callback action to do when the call returns. This is a general mechanism of the asynchronous calls. The server side code, in this case, is very simple, since it has only to use the Java Amazon EC2 library and to invoke the method for describing the Amazon images.

When the user wants to launch one or more instance, he only needs to use the button on the right of the list and select which type of instance and how many instances to run. The user must also specify the minimum and the maximum number of instances to run; the meaning of this double choice is: Amazon tries to launch the maximum number of instances the user indicates and if for any reason this is not possible, for example it is too busy, Amazon tries to allocate less instances and if it fails to run the minimum number of instance, all the launch process will fail and it returns an error. The user is prompted (Figure 4.2) to define these options and then the call to the server is
executed. To ensure that Condor can authenticate the newly launched instance, some information must be passed to the instance itself; in particular it needs the DNS name of the Condor manager, a private key and a certificate to authenticate using SSL and the CA certificate. The Amazon Java library does not allow to transfer a file to the new launched instance like the command-line tool does. So, in this case, we need to use the command-line tool. Before launching an instance, the server must prepare the files with all the necessary informations to let the Condor worker authenticate. We decided to use a bash shell script that by using `openssl` creates and signs a new pair certificate/key for the Condor worker (see section 3.4). These files together with the CA certificate and the name of the Condor manager are put in an archive and passed to the `ec2-run-instances` command-line tool. This command requires as parameter the private key and the certificate of the Amazon AWS user who is launching the instance. Before executing this command we need to decrypt the private key of the user. If everything goes right, after few moments the user will see the instances he launched and their actual status. The instances takes some minutes to go in the state of running, this time is necessary to create the new virtual machines in Amazon Cloud. When an instance starts, the start-up script of Condor gets the archive with the certificates and sets-up Condor config file for the connection to the manager we specified.

### 4.4 Condor management

Condor workers and Condor queue can be monitored using the *Condor Management* tab in the web interface. This panel contains a list (Figure 4.3) of Condor workers (Condor pool); these can be machines from Amazon Cloud, but also any other machines that have the credentials to connect to the Condor manager. In this panel there is also a list of the tasks in the queue. The list of the machines registered on Condor manager and the queue of tasks, use a paging grid in which only a limited number of entries are shown. The reason for this choice is that both lists can be very long and the amount of data transferred between the server and the client can become very big. Using the paging instead, the client specifies the range of data that can be viewed.
### 4.4 Condor management

When the server receives the request for listing the Condor worker, it uses a shell script that calls the Condor command `condor_status` with the option `-xml`. Using this option, the output of the command will be presented in the XML format. The XML output of the command is then parsed to extract informations about the Condor workers. These informations are elaborated in order to remove useless data and to select the requested range of entry for the paging view.

The list of tasks (Figure 4.4) is very useful to monitor and control their execution. Similarly to the list of workers, also the tasks list needs the paging to improve the usability and reduce the amount of transferred data.

In this panel there are two buttons that control the tasks in queue. The first one sends a command to the server asking for removing all tasks in queue. The other one sends the reschedule command to the machines in the pool, forcing the update of the information about the pool and the queue. This command, also, forces the Condor manager to start a negotiation cycle, which is a phase in which the manager probes all the machines in the pool searching for workers whose features match the requisites defined in the submit description file. See Condor manual for more details [9].
Figure 4.4. The queue list of Condor tasks.
The data of the simulations are stored locally on the Condor manager machine and consists of a set of XML simulation parameter files. Each set of files constitutes a job. A job includes many scenarios and each scenario consists of one or more XML files that differs only for the seed of the random numbers generator. The jobs listed in the Simulation Management (Figure 4.5) tab are retrieved from a table in the database (see section 4.2.1). Information about the state of the running jobs are evaluated based on the Condor queue and on the condor_history. The Condor history is simply a list of all the jobs that are no longer in the queue, so they are completed or removed. To obtain a list of scenarios within the job, the user has to click on the corresponding item from the list of jobs. When the server receives this kind of request, it gets from the database the path - on the local file system - of the requested job; this path represents a directory in which are stored the XML files constituting the job. The server creates the list of scenarios basing on the content of this directory. As we talked before, the XML parameter files are compressed using zip format, in order to reduce the amount of data to transfer to the workers.

The submission of a job to Condor workers can be done using the corresponding button, after the user selects a job in the list. The condor_submit command is executed using the submit configuration file that should be in the same directory of the job. This file is created automatically by Protégé [21] together with the XML compressed files of parameters for the simulator. However, the simulator cannot read from a compressed file the parameters for the simulation, so we work-around this problem submitting a shell script and passing the simulator executable as parameter. When a job is submitted to Condor, it passes from the status of 'New' to the status of 'Running' (Figure 4.6). In this status, the server evaluates the percentage of completed scenarios basing on the Condor queue and on the Condor history, this percentage can be monitored through the table of jobs in the web interface. The status is stored in the database and it is reseted from 'Running' to 'New' when the user removes all jobs from the queue; that should not be done except for particular reason, for example when there is a problem in the submission file and all the job need to be aborted.
Figure 4.5. The panel for managing the simulation.

Figure 4.6. The list of jobs and their status.
Another functionality of this part of the application allows the user to add a new job. The user will be prompted to fill a form in which he specifies the name of the job, the path and the number of XML files. The path must be a directory on the local file system of the server containing the compressed XML files and the submit file. This file is written according to the specifics of Condor and including the following rules:

- The executable is the shell script called `SimLauncher.$(OpSys).$(Arch)`
- The Requirements should be:
  
  ```
  Arch == "INTEL" or "X86_64" and OpSys == "LINUX" or "WINNT51" or "WINNT60"
  ```
- The output has to be transfer on exit
- The input files to transfer are the archive of the binary executables and the XML file in the format:
  
  `<scenario_name>-<simulation_num>.xml.zip`

The archive of binary executables will be copied into the directory of the job by the web application. The job directory owner must be the same user that run the servlet container (Tomcat). That is because it needs to write the results of the simulation in that directory.

### 4.6 User access

The web application is built to support multi user access. The Condor manager and the Condor pool is unique for all the users. So if one user runs an Amazon EC2 instance, this is shared among all the other user, nobody can stop it or access it but the resources are shared inside the Condor pool and they can be used to execute Condor jobs.

The Amazon registered user that want to use this application, must first register using the express form. In this dialog window, he must insert the AWS account information. These information are strictly confidential and in particular the private key file, so we decided to crypt them before inserting in the database. Also the private key file is encrypted using `openssl` with AES algorithm.
4.7 Security

The web browsing security is an appreciated requisite but sometimes underestimated. The web communications that transit on Internet are liable of interception and the end user must be aware that when he enters personal information on a web form, it potentially can be watched by a malicious hacker.

In this web application, much of the information that transits between the web browser and the web server are critical. For example the registration procedure and the login process send personal information to the web server. So it is essential to encrypt the connection using the SSL or TLS protocol, fully supported by the majority of web browsers.

Tomcat web server and Catalina can be enabled to support secure connections. The SSL configuration requires the creation of a certificate (even self-signed) inserted in a Java keystore. We configured the Catalina web application to only accept secure connection, so if the user attempts to connect using standard HTTP protocol, he will be redirected to HTTPS.

To improve the security of the stored user data on the server machine, all the critical informations are encrypted using enhanced cipher algorithm. The user informations the server retains are:

- User name
- User password
- AWS access key ID
- AWS access key
- AWS owner number
- AWS user certificate file
- AWS user private key file

Much of this information is critical and it is easy to understand that the server must take care of them storing securely. The password is stored on the database as \textit{MD5} digest. The AWS access key id, the AWS access key and the AWS owner number are encrypted using DES
and stored in the MySQL database. The key used to encrypt them is the password plain text.

The most critical data is the private key file. This is used by the web application to launch the instances on Amazon EC2. This file must be passed as a parameter of the EC2 command line tool (ec2-run-instances) together with the certificate file. The web application has this file encrypted on the web server machine file system and it temporarily decrypts it when the user launches one or more instances. The algorithm used for encryption is AES and the tool used to cipher/decipher it is openssl (see 4.2.1).
Chapter 5

Discussion

In this chapter we discuss our choices with respect to the architecture and which kind of changes or improvements can be done to achieve better performance or better scalability of the system.

5.1 Architecture

We start talking about the computational environment. The execution platform, as we presented with practical results, gave us excellent results in terms of execution time related to the cost per hour of running Amazon EC2 instances. The solution using Amazon instances gave us the maximum scalability allowing users to expand the system according to his requirements. The charges for used capability are very low compared with the cost of building a dedicated cluster of parallel machines. Also other costs that should be considered if using dedicate clusters are not required using Amazon EC2, like costs of maintaining the system, updating it, handling failures etc. The flexibility of the system is very high, since we can make changes without having to upset the architecture, but simply changing the component we need. Instances are simply clones of a base image, which means that it is possible to change, delete, create new base images of the Condor Worker without having to change other parts of the architecture. Simply bundling an new instance and it is ready to be executed. Once the solution of the mix of Condor on Amazon EC2 Instance was approved and confirmed, the following step was the decision about two important points of the architecture:
Discussion

- The Central Manager

- The Storage System

The choice of the local Central Manager and Storage System has advantages and disadvantages. Starting from disadvantages, the first one is that the Central Manager is the central point of the architecture, which means that an eventual failure of it compromise the operations of the whole system. Using a dedicated EC2 instance, this problem would be solved since Amazon guarantees a good level of reliability. It would be possible to bundle a dedicate image on Amazon and start it before launching Condor Workers, then stop it after the conclusion of the simulation jobs. The choice of the local storage system is related of the first one, since it is more simple by the Condor Manager Machine to access to local data instead of managing data on S3 buckets. But also in that case, we do not have the reliability that Amazon offers for its services. It is however true that the storage system of the computational environment can be separated from the storage system of the modeling tools and reporting tools, which means that the data stored on our local system can be generated again by the modeling tools after a failure. We pay disadvantages in reliability with advantages in terms of high scalability and flexibility. Using a local machine and a local storage system, we are not strictly coupled to the Amazon services. We use Amazon only for the more expensive parts of the system: the cluster of workers. Using Amazon instances for parallel workers gives the possibility to save money and have the maximum scalability and flexibility, but the architecture is not really dependent on Amazon. If for example Amazon has not sufficient incomings from this services and decides to retire the EC2 system, our architecture is sufficiently flexible to switch supplier and run Condor Workers on other compute cloud. It is also possible to have more than one compute cloud running workers at the same time: it can be done simple letting the Condor Manager accept connections from the new cloud and bundling a Condor Worker image for it. Also office workstations could host Condor Workers, using their unused CPU time, which is the original aim of the Condor project. Adding more Workers from different networks would not change the configuration of the Condor Manager and of the Condor Workers: scheduling policies, CPU usage configurations, authentication and cryptography methods are still un-
Figure 5.1. The Computational environment using more compute clouds
changed on other execution systems. For that Condor is a very good instrument, since it can work with multiple environments at the same time supporting many kinds of architectures and operative systems.

Other open source alternatives to Condor could be found on clustering systems, like *Maui Cluster Scheduler* [16] or *Beowulf project* [4]. The problem of these clustering projects is that they do not support such a good heterogeneity of systems like Condor: for example they cannot manage MS Windows clusters. Also grid architectures can suit our needs, *Globus toolkit* for example could be a good instrument to build a grid system. But in that case a grid system would change radically the system architecture.
5.2 Web Application

The complexity of the a web interface should always be related to the usability. If the interface has few but well defined widget the user can easily interact but perhaps he will not find the functionality he is looking for, on the contrary, many widget can create confusion and the user can make mistakes. In our work, we preferred to use few widget but enough to achieve the main task rather then fill the web interface with buttons and tables displaying useless information. When programming web interface, there is no limit on the amount of widgets and facilities you can add, so one of the possible developments can be a more rich and interactive user interface.

The web interface implemented in this work uses a flexible and modular framework. In fact, GWT allows to quickly remove or plug modules on the application. On the server side, the servlet concept as well as the GWT RPC are a powerful instrument to implement a client-server infrastructure at high level. Google\textsuperscript{TM} provides a very detailed and exhaustive documentation about this technology, so we did not have particular initial difficulties. The client-server interaction can be obtained using the default GWT widgets, but we enriched the interface using the GWT extension \textit{Ext GWT}. This extension provides more good-look appearance widgets and it is based on the \textit{Ext JS}. This is a very nice alternative to the GWT widget library that in some cases requires the composition of base widgets in order to obtain complex but frequently used widget.

A possible alternative to Ext GWT for the client side code is \textit{GWT Ext}, which provides the same functionality as Ext GWT. They differs in class structure and in licensing terms. This extension affects only the client side code, so its replacement can be obtained with minimal modification on the web application.

The modularity of this web application allows us to add new components for managing other parts of the architecture. We talked about the possibility to easily insert new computation resources on the pool of Condor, so the web application could also manages other kinds of cloud as well as internal network machines.

In the simulation management panel, there is the functionality called \textit{Add job} (see section 4.5); this functionality makes sense only if the user is working on the same machine where the Condor manager
is running. This part of the application must be improved because we supposed the application to be used also by remote users and since this functionality requires a local path on the Condor manager machine, in which the simulation job is stored, the user should copy the job directory on the Condor manager machine file system before adding it to the list. However, this can be easily obtained using \textit{ssh} or \textit{samba} protocol. Another solution to this problem might be the creation of the job directly through the web application; we know that the input of the simulator is an XML file created using Protégé [21, 22] and Web Protégé editor (see section 3.1). This allows the creation of ontology and the web editor could be integrated in our web interface. In this way the whole process of creation of population, simulation, results viewing can be done in an unique environment.
Chapter 6

Conclusions

In this report we talked about the architecture of the computational environment based on Condor Workers over Amazon EC2 instances. We discussed the reasons for the choice of Computational Cloud, the middleware used to execute jobs on remote machines, the choice of the location of the Central Manager and of the Storage System. We also presented security aspects of the communication, which is an important issue when dealing with networking. Next we talked about the Web User Application, that allows the managing of the architecture with a friendly user interface. Users can manage instances on the Amazon EC2 cloud, submit jobs and have a fast way to see the results before processing them with the reporting tools.

We tested the application with a real simulation Job: it consists of 28 scenarios repeated 10 times, which means 280 XML files for the same number of simulations. The size of each file is 21MB, that becomes 2.1MB with compression using zip. We launched 2 High-CPU medium instances and then submitted the Job: the total execution time was 2h40m. Considering that the cost for renting two High-CPU medium instances for three hours is 1.32 USD, and the cost for the transferred data, 590 MB at 0.10 USD per GigaByte, this simulation job cost around 1.38 USD. If we consider that instead of running two instances for three hours we should launch six instances for one hour, the cost would be the same saving a lot of simulation time.

The Execution Environment is then cheap but fully scalable at the same time, with the support of a Web management system that allows all simulator users, even who don’t have technical knowledges
Conclusions

about the underlying architecture, to easily manage the system and complete their simulation jobs.
Bibliography


Appendix A

Use Case

In this chapter we discuss a typical use case of the web application; in particular we focus the attention on the registration of a new user, on the managing of Amazon EC2 instances, on the Condor queue management and on the job submission.

When a new user wants to access the system, he must first register (Figure A.1). The registration process requires that the user inserts the data about his Amazon account. These data are stored on the server and they are protected by cryptography. The required data are:

- The AWS access key ID
- The AWS access key
- The AWS owner ID

in addition, the user must insert a chosen username and a password. The registration also requires two files to be uploaded to the server. These files are the certificate and the private key of the AWS user. The private key will be stored encrypted on the server file system.

After that, the user can log on to the system (Figure A.2) and manage his Amazon private images (Figure A.3).

The user can now run one or more instances of his private AMI; to do so, he just needs to select the AMI from the list and click on the "Run selected" button on the right. The user will be prompted for additional information about the instances to run (Figure A.4).
In particular, the application asks for the type and for the number of instances. The meaning of min/max instance number is described in section 4.3.

After a few seconds the user will see the instances launched on the list below, this list also shows the status of the instance (pending, running, terminated).

From the simulation tab, the user can see the actual status of the jobs registered in the server database. These jobs can be submitted to Condor just clicking on the button "Submit" (Figure A.5). The
Figure A.3. The Amazon EC2 private AMIs list.

Figure A.4. The Amazon EC2 run instance dialog box.

submission will enqueue the whole job to Condor and change its status into "Running".

When a job is completed, the user can inspect the results of the scenario on the scenarios list below (Figure A.6).

The user can also monitor the Condor status and the Condor queue. These information are useful to check if everything works properly.

When the user logout, his instances as well as his submitted jobs
Use Case

Figure A.5. The jobs list.

![Jobs List](image1)

Figure A.6. The jobs list and the list of scenarios with the results.

![Scenarios List](image2)

continue to work. When he logs in again, he will see the progress of the jobs and he can decide to stop his instances or to add new ones.
Appendix B

Deploy

The web application has its core on the servlet that implements the server side code. A servlet is an object, instance of a class that implements a service, the interface of this class is the contract between the client and the server. A servlet needs a particular environment to be executed and a particular web server that allows the execution of its code. These kind of web servers are called servlet containers and Tomcat is one of the most spread. It is a web server constituted by components that allows the execution of servlet and JSP pages. The servlet container component of Tomcat is called Catalina.

The GWT web application can be deployed in such environment and the deployment process requires some precise steps to be followed. For this purpose, we used a bash script containing all necessary operations to insert the application in a servlet container.

Here we describe how the script executes this task. The script must be in the same directory as the Eclipse project.

At the beginning some variables are set, like the package name and the servlets names; we can modify it if, for example, we add a new servlet.

```bash
#!/bin/sh
gwt_path=/usr/local/gwt-linux-1.5.3/
package=se.liu.ida.conama
modulename=WebManagement
application_name=ConAmaWeb
servlet_names=( UploadServlet ConAmaService )
servlet_classes=( $package.server.UploadServlet
```
$package.server.ConAmaServiceImpl
servlet_url=( /UploadServlet /ConAmaService )

eventually previous deploy folder are removed and created a new one
if [ -d deploy ];then
  rm -rf deploy
fi
#GENERAL ACTION
mkdir deploy
	hen the application is compiled using the GWT tool
./$modulename-compile

now the compiled JavaScript and the other resources are copyed in the deploy directory and the default servlet structure is created
cp -r www/$package.$modulename/* deploy
mkdir WEB-INF
mkdir WEB-INF/classes
mkdir WEB-INF/lib
cd WEB-INF

the web.xml file specifying the servlet features is dinamically crated with all the servlets we specify before in the script
echo "<?xml version="1.0" encoding="UTF-8"?>" "
> web.xml
i=0
for servlet in "${servlet_names[@]}"; do
  echo "<servlet>
  echo "<servlet-name>"$servlet"</servlet-name>" "
  >>web.xml
echo "<servlet-class>${servlet_classes[$i]}</servlet-class>" >>web.xml
echo "</servlet>" >>web.xml
i='expr $1 + 1'
done

for servlet in "${servlet_names[@]}"; do
echo "<servlet-name>$servlet</servlet-name>" \ \ 
"<url-pattern>${servlet_url[$i]}</url-pattern>" \ \ 
" >>web.xml
echo "</servlet-mapping>" >>web.xml
i='expr $1 + 1'
done

This is an important part of the Catalina application. This section forces the web user to require an SSL connection to the web server running Tomcat. For example if the user tries to access the web application using the normal HTTP protocol, he will be redirected to the HTTPS protocol using the corresponding port specified in the Tomcat configuration file. Obviously Tomcat must have in its configuration file (server.xml) the SSL connector enabled.

echo "<security-constraint>" >>web.xml
echo "<web-resource-collection>" >>web.xml
echo "<web-resource-name>$application_name" \ \ 
</web-resource-name>" >>web.xml
echo "<url-pattern>*.html</url-pattern>" >>web.xml
echo "<web-resource-collection>" >>web.xml
echo "<user-data-constraint>" >>web.xml
echo "<transport-guarantee>CONFIDENTIAL" >>web.xml
echo "</transport-guarantee>" >>web.xml
echo "</user-data-constraint>" >>web.xml
echo "</security-constraint>" >>web.xml
Deploy

```
echo "</welcome-file-list>" >>web.xml

echo "</web-app>" >>web.xml

    all third part libraries and the GWT servlet jar package are copied in the deploy directory

cd ../../bin
cp -r * ../deploy/WEB-INF/classes
cd ../lib
cp -r * ../deploy/WEB-INF/lib
cp $gwt_path"gwt-servlet.jar" ../deploy/WEB-INF/lib
    cd ..

    in the end of the script, some necessary resources are copied in the deploy directory like an XML file containing the parameters for the application. Finally the deploy directory is copied in the Catalina webapps directory and the web application can be executed browsing the subfolder called with the application name we specified

#SPECIAL ACTION
    cp properties.xml deploy
    rm -rf $CATALINA_HOME/webapps/$application_name
    mkdir $CATALINA_HOME/webapps/$application_name
    cp -r deploy/* $CATALINA_HOME/webapps/$application_name
```