Títol: Policy-Driven Resource Management for virtualized Grid providers

Volum: 1/1
Alumne: Gemma Reig Ventura

Director/Ponent: Jordi Guitart Fernández
Codirector: Mario Macías Lloret
Departament: Arquitectura de Computadors
Data: 17 de juny de 2008

DADES DEL PROJECTE

Títol del Projecte:	Policy-Driven Resource Management for virtualized Grid providers
Nom de l'estudiant:	Gemma Reig Ventura
Titulació:	Enginyeria Informàtica
Crèdits:	37.5
Director/Ponent:	Jordi Guitart Fernández
Codirector:	Mario Macías Lloret
Departament:	Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Jordi Torres i Viñals
Vocal:	Ana Cristina Zoltán Torres
Secretari:	Jordi Guitart Fernández

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Agraïments

En primer lloc, m'agradaria donar les gràcies als directors d'aquest PFC, Mario Macías i Dr. Jordi Guitart per l'oportunitat de realitzar aquest projecte i per la seva guia i els seus consells al llarg dels darrers mesos.
Sense cap mena dubte, la meva família es mereix que li dongui les gràcies per la seva confiança en mi i la seva paciència.
Voldria agrair especialment a en Xavi i a la Montse el seu suport i els ànims que m'han donat en tot moment.
Per últim, vull recordar als companys tant de la UPC com de la UAB per tot el que hem compartit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last years, the computing necessities have changed. The necessity of solving large scale computational intensive problems as in genetics or physics fields, or those that require access to big amount of data, or to provide high available services as in business, etc. brings us to a cooperative and large scale computing paradigm. Besides, due to the competitiveness of the business world, users demand high configurable and adaptative software to enable changing the system behaviour as quick as the world change.

Grid computing has emerged for solving large scale problems by means of a cooperative infrastructures. But this computing paradigm entails the server underutilization problem. It is due to the fact that user necessities of these servers change frequently throughout the time. This is a huge trouble because underutilized servers lead to a waste of resources, money and space, besides they are contaming unnecessarily.

Virtualization and resource management are two topics that can help to solve these problems. We can improve server utilization by virtualizing it. This way, there is no need for a server per client as we might be able to consolidate them. Different clients can share the same physical server in a secure, efficient and transparent way. And with an efficient resource management, the virtual machines can be dimensioned, started, stopped and even replicated on demand. Moreover, driving this resource management through policies allows building high configurable and high adaptative systems.

Therefore, in this PFC, we propose a Policy-Driven Resource Management system (PDRM). This allows resource providers to control which policies they want to apply to adapt the behaviour of the system in order to fulfil their business goals. Furthermore, our proposal avoids resource underutilization by means of an efficient resource management and server consolidation through

virtualization.

1.1 Context and opportunity

This PFC arise from a collaboration grant of the Ministry of Education and Science (MEC) to collaborate with the Computer Architecture Department (DAC). The work done in this collaboration grant has been used as a starting point for the project.

This project is developed within the framework of the EERM, Economically Enhanced Resource Manager component of the European Project SORMA, Self-Organizing ICT[1] Resource Management [1].

The main goal of SORMA is to develop a platform that allows ICT resources trading on demand. This platform also has to control the fulfilment of the contracts between resource clients and resource providers and to provide them with mechanisms to efficiently exchange resources without clients concerning where their jobs are being executed.

Once in the SORMA market a resource usage has been agreed, a contract is supplied to EERM over the Grid Market Middleware (GMM). It contains the necessary information to allocate the resource, fulfill the agreement between resource provider and consumer and to execute the client's requested job.

The EERM component has the onus of the isolation between the economic and technical part of SORMA. Its main responsabilities are to evaluate whether a job can be done –carrying out with the agreement– or not, to fix a price for performing the job, and to monitor both resource performance and SLA violations. Besides, these jobs are executed in isolated machines by means of resource virtualization [2] which allows the separation between application semantics and resource management.

1.2 Project overview and objectives

The main purpose of this project is to construct a prototype that render resource providers with a mechanism to create virtual customized execution environments on-demand and define policy rules to maximize the revenue yielded by selling these execution environments and to maximize clients' satisfaction.

In order to achieve this goal, we must build three EERM components and define the interactions

[1] Information and Communication Technologies

with the rest of components. These components are: Policy Manager (PM), Economic Resource Manager (ERM) and Resource Fabrics (RF). Thereafter, we can divide that main goal in other more specific goals which can be classified depending on the EERM component they belong:

Resource Fabrics For our purpose, it is necessary a way to interact with resources to provide consumers with an isolated and customized environment to execute their jobs and with an API to manage them.

Economic Resource Management To ensure an efficient use of local resources.

Policy Manager To store and manage policies to adapt the EERM behaviour at runtime.

1.3 Task description and initial plan

The project is divided in the following tasks to reach the goals presented in the previous section:

Initial planning : A work plan to develop the project and to define its scope and objectives.

Resource Fabrics : We will use virtualization to provide users with an isolated and customized environment for the jobs' execution. Thus, we will need:

- **Virtualization** Services to manage virtual machines.
- **Repository image** A repository image service to provide on-demand customized disk images for the client's virtual machines.
- **Job Management** To describe orders to manage clients' jobs.

Economic Resource Management To allocate resources to clients. It should query Policy Manager and take simple decisions.

Policy Manager To provide an infrastructure for rules definition and to construct a simple example of policy.

Test and evaluations At this point, each component must be tested separately. This task includes integration, test and collect the results of the overall project.

Final Report In the tasks described above it is included a documentation subtask. Hence, this task consist in the final report writing and revision.

Presentation To prepare the final presentation.

Figures 1.1 and 1.2 show how we had scheduled these tasks along the time.

WBS	Name	Start	Finish	Duration
1	Image Respository	Feb 11	Apr 3	38d 6h
2	Progress report	Mar 17	Mar 17	2h
3	Resource Fabrics	Apr 3	Apr 14	7d 2h
4	Policy Manager	Apr 15	Apr 17	3d
5	Economic Resource Management	Apr 18	May 2	10d 2h
6	Prototyping	May 2	May 5	1d 2h
7	Testing	May 5	May 8	3d
8	Final report	May 8	Jun 4	18d 6h
9	Presentation	Jun 4	Jun 11	5d

Figure 1.1: Initial plan.

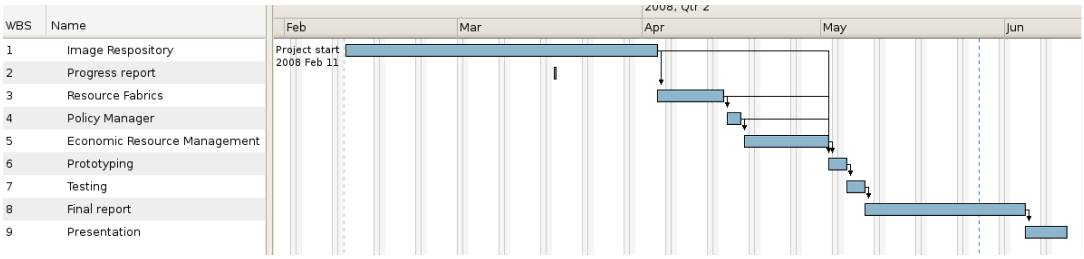


Figure 1.2: Initial plan gantt.

1.4 Report structure

The remainder of this document is structured as follows: we explain several concepts to facilitate the understanding of this report in chapter 2; in chapter 3 and chapter 4, project requirements and specification are introduced; later on this report, we discuss the architectural view of the system in chapter 5; we introduce our design and implementation decisions in chapters 6 and 7 respectively; we gather the results of testing our system in chapter 8; through chapter 9, we present the related work of this project and through chapter 10 we compare the initial development plan with the real one and we calculate the cost of the project; finally chapter 11 concludes and presents some future work. These chapters are structured in project components in a bottom-up fashion (RF, ERM, PM) when possible.

Moreover, this report is enclosed with several appendixes to complete the information about the overall project: in appendix A we provide a guide for the deployment and installation of the system; in appendixes B and C, a developers manual is supplied. In the first, we describe the language used to communicate remote components and in the second, we specify how to interact with and how to extend the lowest level components; finally, the reader can refer to the Glossary appendix D to find out about concepts and abbreviations definitions used along this report.

Chapter 2

Preliminary concepts

In order to help the reader with the comprehension of this report, some basic concepts are presented in this chapter. For a quick reference of these terms the reader can also look up for them in the glossary chapter D.

2.1 Grid

Grid intention is to share resources between virtual organizations (VOs). Those resources may be heterogeneous, geographically distributed, accessed on-demand and owned by diverse organizations which potentially have different policies for administer them. VOs are logical entities formed by a set of individuals and/or institutions created dynamically to resolve a common problem.

In 2002, Ian Foster wrote a checklist [3] with the intention of formalizing what is the Grid because there was a trend to use freely this term and the truly scope of the concept Grid was not clear . In its checklist he publicizes that *a Grid is a system that coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of services*.

Earlier definitions were advertised before, maybe the most popular is the analogy between Grid computing and electrical power, in both (electrical/computing) power are provided on demand without the user caring about where and how this power is generated.

2.1.1 Grid architecture

Next, is explained the Grid architecture as defined in Foster et al [4].

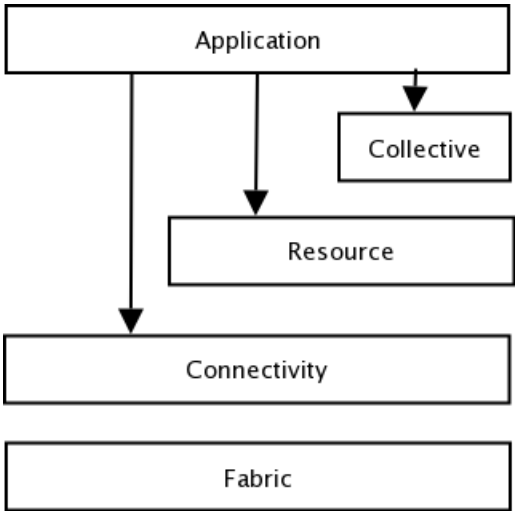


Figure 2.1: The layered Grid architecture.

Grid architecture is a protocol architecture in nature because it requires the capability of sharing relationships among any potential participants. In a networked context, that means reach interoperability by means of common protocols. VO users and resources negotiate, manage and exploit sharing relationships using the basic mechanisms defined by those protocols. Figure 2.1 shows Grid architecture layers. From bottom to top those layers are:

Fabric: This layer implements resource specific operations to provide access to those resources and resource management mechanisms. (e.g. in computational resources, mechanisms are required for starting/stopping applications and for controlling and monitoring the execution of that process).

Connectivity: Defines transport, routing and naming protocols for the secure exchange of data among Fabric layer resources.

Resource: Defines protocols for the secure negotiation, initiation, monitoring, control, accounting and payment of sharing operations on individual resources.

Collective: It is in charge of coordinating interactions across multiple resources.

Application: User applications within a VO are constructed in terms of services defined at any layer.

2.2 Virtual machines

The term *virtual machine* has a bunch of very different definitions in the literature. For this reason, the intend of this section is to clarify which types of virtual machine exist and what we would mean in this document *for virtual machine*.

First of all, it is necessary to define what a machine is in the context of virtual machines. Therefore, there are two big families of virtual machines depending on what the definition for machine.

- On one hand, from the point of view of a process executing user code, a machine is what a process needs to execute its associated code. In this case, a logical memory address space, registers, user-level instructions and I/O resources which are available only through operating system. Thus, from the operating system perspective and its applications a system is a full execution environment in charge of supporting several simultaneous process and assign them memory and I/O resources. This kind of virtual machines are termed *process virtual machines*.
- On the other hand, from the perspective of the system, a machine is the ISA provided as an interface to interact with the hardware. This type of machines are called *system virtual machines*.

Hereafter, throughout this document we will refer to *virtual machine (VM)* as *system virtual machine*.

In the following sections, we present a summary of the different types of process and system virtual machines. This information and figures have been extracted from [5] and [6]. Please refer to them for a more extended explanation.

2.2.1 Process Virtual Machine

A process virtual machine is a virtual platform that executes a process. So, the virtual machine is created with the process and is terminated when the process dies.

The virtualization software that implements a process virtual machine is called runtime. In figure 2.2 a) we can see that the virtualizing software is at level of API or ABI interfaces, over the operating system and hardware. Thus, runtime emulates user-level instructions as well as operating system or library calls.

Within the family of process virtual machines we can also distinguish many classes:

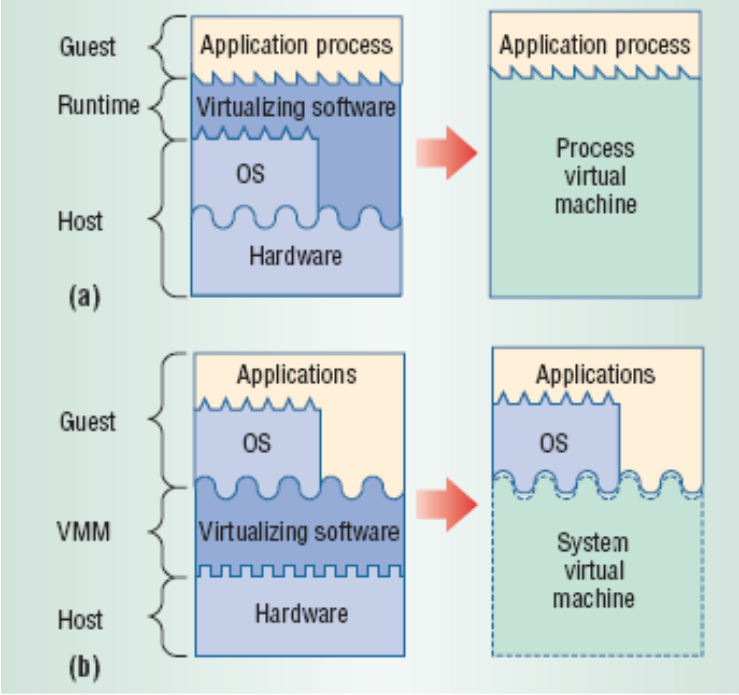


Figure 2.2: a) Process virtual machine perspective of the machine b) System virtual machine machine's point of view

Multiprogrammed systems: They use multiprogramming to support multiple user processes, it gives each process the illusion of a whole machine for itself (e.g. most operating systems).

Emulators and dynamic binary translators: VMs that support binary programs compiled in an instruction set which differs from the one executed by the host (e.g. Intel IA32-EL).

Same-ISA binary optimizers: VMs with the solely purpose of optimize code during translations. The instruction set of the host and the guest must be the same (e.g. Dynamo).

High-level-language (HLL) VMs: They are intended to reach cross-platform portability. In a HLL VM a compiler front end generates abstract machine code in a virtual ISA that specifies the VM's interface. Each host platform implements a VM capable of loading and executing the virtual ISA (e.g. Java).

2.2.2 System Virtual Machine

A system virtual machine is a complete, persistent runtime environment that allows an operating system and the applications that it supports to run over it.

The term of Virtual Machine Monitor (VMM, a.k.a hypervisor) is used to refer to the virtualizing

software used in a system virtual machine. As depicted in figure 2.2 b) we can notice that the VMM is atop the host hardware and under the guest software. So the VMM emulates the hardware ISA.

Within the family of system virtual machines we can also distinguish many classes:

Classic system VMs: in this approach the VMM runs in the most privileged mode, whilst guest system runs with reduced privileges. VMM can intercept and emulate guest operating system actions that manipulate hardware resources.

Hosted VMs: Implementation that builds virtualizing software on top of an existing host operating system. User installs it just like a typical application program (e.g. VMware GSX server).

Whole-system VMs: it virtualizes all software, including the operating system and applications. It is useful when the host and the guest do not have the same ISA (e.g. Virtual PC in which a Windows system runs on a Macintosh platform).

Multiprocessor virtualization: it is used when the underlying host platform is a large shared-memory multiprocessor.

Codesigned VMs: its sole purpose is to emulate the guest's ISA (e.g. Transmeta Crusoe).

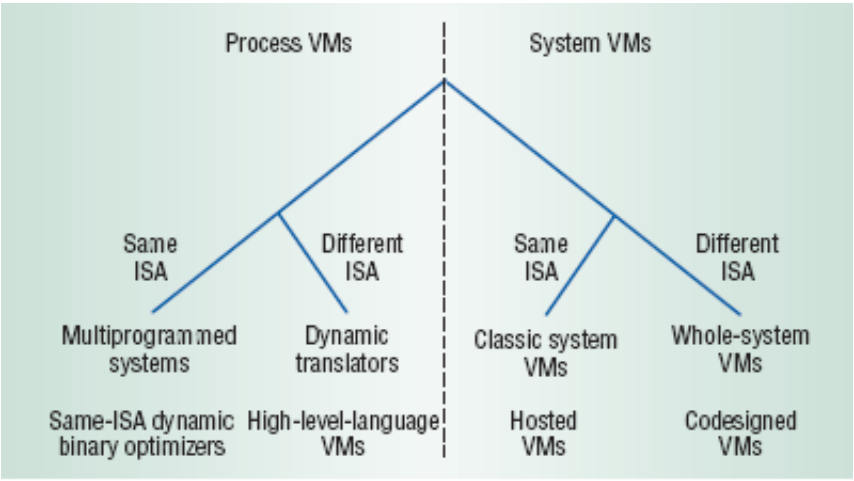


Figure 2.3: Virtual machine taxonomy. First they are divided into either process or system VMs and within this two big categories they are classified in terms of whether both the guest and the host use the same ISA or not.

2.2.3 Virtualization in Xen

Xen is a VMM based on a x86 processor architecture [7] [2] [8] [9]. It was developed by the Systems Research Group at the University of Cambridge Computer Laboratory as part of the XenoServers project [10]. Nowadays the project has grown enabling researches to investigate techniques for virtualize resources (CPU, memory, disk and network). Some of the project contributors include: Intel, IBM, HP, AMD, Novell, RedHat.

The most important goals and challenges of Xen are:

- To support great many operating systems to accommodate the heterogeneity of applications in a functional and easy way.
- To isolate virtual machines' instances from each others.
- The overhead introduced by the virtualization software should be small.

To reach these objectives, Xen offers two forms of virtualization:

Paravirtualization: This type of virtualization force guest operating systems to be modified to use a special hypercall ABI. This allows Xen to achieve a performance nearly to the one of the host machine (i.e. the overhead introduced by the hypervisor is minimum) without any hardware support. The hypervisor must be loaded in kernel space as is in charge of the resource management between the different operating systems.

Full virtualization: This form of virtualization allows to run any operating system without being specifically ported to Xen. In this case, the hypervisor introduces some more overhead on the performance. It is needed hardware support to do full virtualization.

2.3 Business Rules Engine

A Business Rule Engine is a software that permits the isolation of business policies from the application design. That means, in some business environments or in any frequent changing environment it is useful to adapt the behaviour of our applications without compiling the source code, whether it be because the source code is not available or because any change in the business logic must to be immediately reflected in the application behaviour (e.g. dealing with the dynamic changes of market economics). A Rule Engine (RE) could evaluate and execute business rules and use them to manage some of the business logic. Therefore, each day becomes more frequent that applications will externalize their business logics as much as possible using rules.

A business rule is a statement, expressed as *if-then-else statements*, that represents a business logic. Rules define or constraint some aspects of our business that must to be present in the application.

Some examples of REs are: Drools, Fair Isaac Blaze Advisor, ILOG JRules, and Jess, the lastest is explained in the section below. Existing so many different REs, some efforts have been done to define a common API for the interaction between a high level programming language and REs. Concretely, JSR 94 (Java Specification Request) is an API to access to a RE from Java. The JSR 94 reference implementation is built as a wrapper over the Jess Rule Engine.

2.3.1 JESS: Java Expert System Shell

Jess [11], is a small, lightweight and fast Rule Engine for the Java platform developed by Earnest Friedman-Hill at Sandia National Laboratories in Livermore, CA. Jess is also a scripting environment that provides access to the complete Java API.

On the one hand, this RE is able to inference knowledge using forward chaining reasoning. For this purpose, it uses an efficient mechanism for solving the problem of many-to-many matching. Concretely, it uses an enhanced version of Rete [12], an algorithm for solving this matching problem. On the other hand, Jess has some extra features that outstand Jess from the rest of REs. These include: backwards chaining and working memory queries.

Therefore, Jess is a good choice if we want to externalize the business logics from a Java application.

Chapter 3

Requirements

This chapter presents the background and requirements of the project. The global context of the project is explained here to facilitate the understanding and to justify project's requirements which are listed later on.

3.1 Background

3.1.1 SORMA

As explained in the Introduction (see chapter 1) SORMA is an European Project which aims to build a platform for an efficient market-based allocation of resources. For a better comprehension of the scope of SORMA, here is explained its logical architecture view.

Figure 3.1 shows the layered architecture of SORMA in terms of its functional components, their responsabilities and their dependencies. Thereafter, we explain each layer in a top down fashion, from system users to hardware resources:

Layer 5: Grid Application Layer

This layer takes into account Grid applications which are going to run in the resources and the humans involved in it, such as Grid applications end-users or IT support staff.

Layer 4: Intelligent Tool Layer

This layer provides to the upper layer with mechanisms for an easier access to the SORMA market. It offers tools to the clients for describing the technical requirements of their Grid

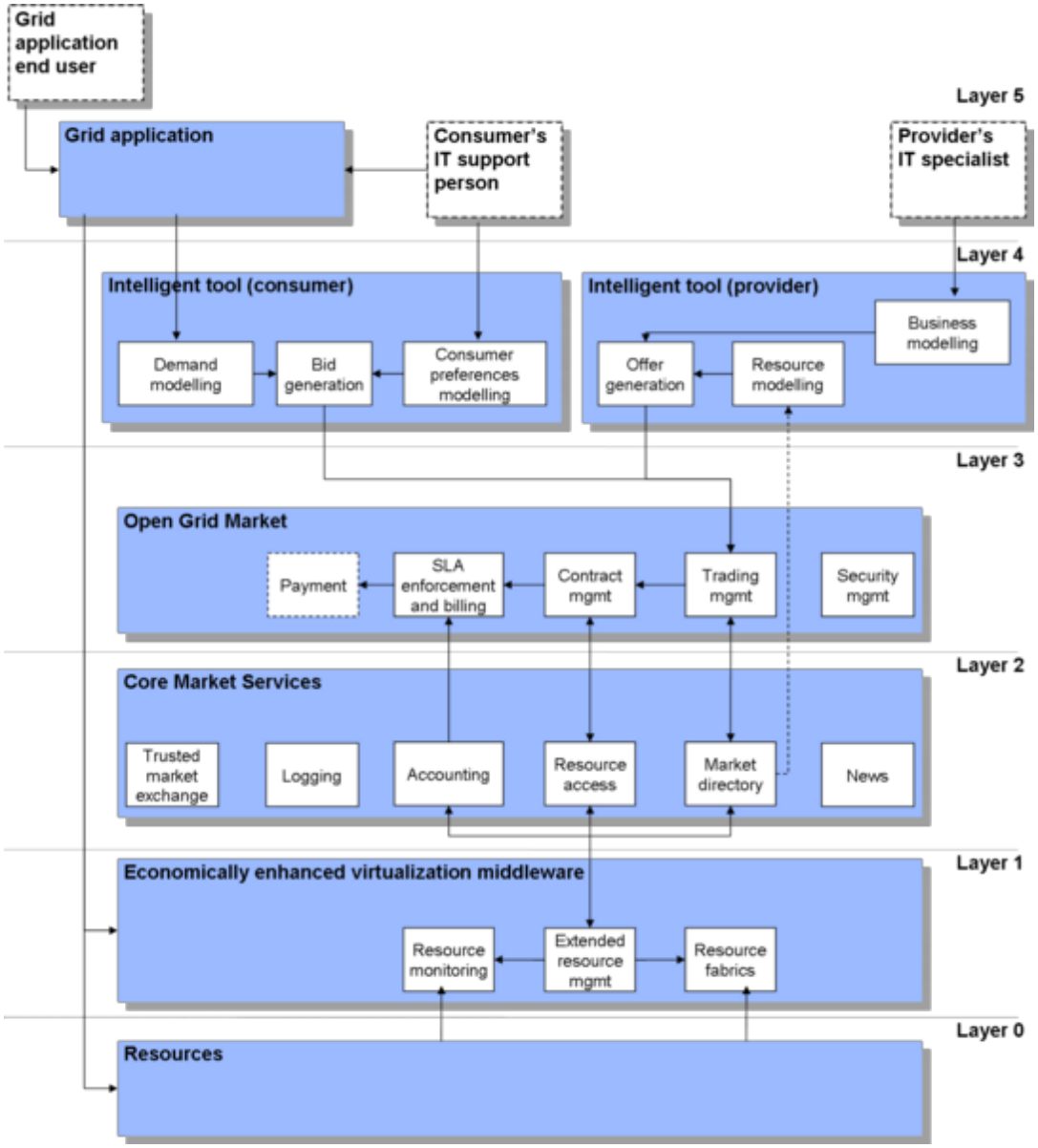


Figure 3.1: SORMA architecture logical view. Arrows in the figure represent dependencies not data or control flow.

applications and for describing their economical preferences that will determine their bidding strategies on the Open Grid Market. Analogously, providers can also technically specify their offers and describe their business models to determine the generation of their offers on the Open Grid Market.

Layer 3: Open Grid Market Layer

This layer is in charge of the allocation of the resources to the clients' Grid applications. These assignments are carried out taking into account certain rules played into the market.

Layer 2: Core Market Services

This layer extends the standard Grid middleware from layer 1 with the necessary services for an open marketplace. It offers services to: communicate participants in a secure and reliable way, log all the transactions executed on the market, trade which resources are used by whom, interact with the lower layer and provide information about prices, resources' usage and so on. An association of some of these services is what is called Grid Market Middleware (GMM [13]) in the following chapters.

Layer 1: Economically Enhanced Virtualization Middleware

Within this layer EERM and resource fabrics components are placed. Those components conform the core of this PFC, so they are explained in further detail in the next section (3.1.2). Besides these components, a monitoring element is located in this layer. The lastest is in charge of measuring the state of the resources in terms of technical parameters.

Layer 0: Physichal Resource Layer

Layer of the plain technical resources traded on the SORMA market, such as virtual machines.

3.1.2 EERM

In a similar way as the previous section, this section is intended to explain the functionality of the EERM in terms of its architectural elements, which are showed in figure 3.2.

As explained in the Introduction chapter (see chapter 1), once a client has contracted a resource where to execute its tasks a SLA is supplied to the EERM through the GMM. Next, the responsabilities of EERM are exposed :

Economy Agent (EA): This EERM component is responsible of deciding if a task is technically and economically feasible and calculating its price based on the client's preference (e.g.

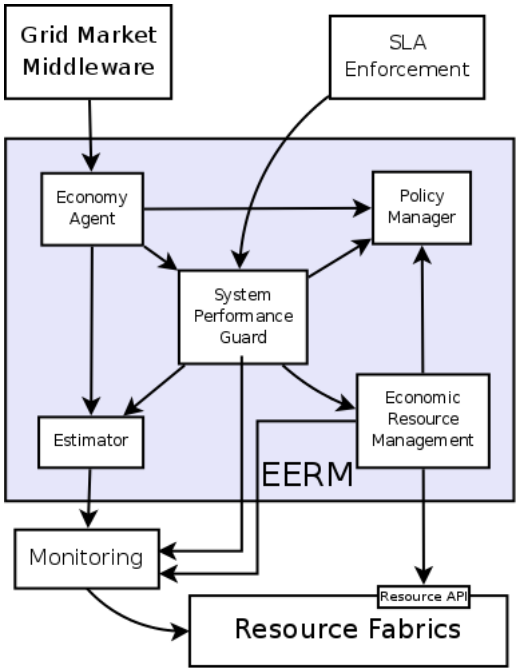


Figure 3.2: EERM architecture.

regular client, standard), economic policies and current and future estimated amounts of resources' supply and demand.

Estimator Component (EC): EC estimates the expected impact of executing a task on the utilization of resources. Thus, the system is able to prevent losing performance due to an overload of these resources.

System Performance Guard (SPG): This component is in charge of the fulfilment of the SLAs. In case of resource performance problems it is notified by the monitoring component. Thereafter SPG might take decisions about cancelling, migrating tasks, etc. with the objective of accomplishing more important SLAs, i.e. those SLAs which maximize the overall revenue. Those decisions are made in terms of the policies stored in the Policy Manager.

Policy Manager (PM): This component stores and manages policies concerning to client classification and task cancellation or suspension, etc. This is a very important component of EERM –almost all the components interact with it– because it allows to adapt EERM's behaviour at runtime.

Economic Resource Manager (ERM): The Economic Resource Manager has the onus of ensuring an efficient use of local resources. For this purpose, it is in charge of the communication with local resource managers and influence them to achieve a more efficient global resource usage.

3.2 Functional requirements

In this section, functional requirements will be explained. These are classified in terms of the architecture component within which they belong. This separation in components (Resource Fabrics, Economic Resource Manager and Policy Manager) is maintained in following chapters when possible.

3.2.1 Resource Fabrics requirements

The main functionalities of this component are enumerated and briefly described next:

- *Must provide mechanisms for the customized creation of virtual machines over heterogeneous resources.* The resource fabrics component must be able to create customized virtual machines. It means, that the user can decide which kernel version, which distribution and with which applications the virtual machine will be created.
- *Must provide mechanisms for the management of a virtual machine.* This functionality include start and stop VM and its services.
- *Job management.* This component must provide functions for the management of the jobs that will be executed by virtual machines (e.g. to send a job to a VM).
- *Must maintain a resource register.* This component has the onus of maintaining information about available physical resources.

3.2.2 Economic Resource Manager requirements

Next, the requirements of the ERM component are listed:

- *Must maintain the necessary information about VMs.* ERM must maintain a registry of the available VMs and their characteristics (e.g. which applications each one has installed, its reservation timetable, etc.).
- *Must be able to contact with Resource Fabrics for the managing of VMs.* That is, it could call create, start and shutdown mechanisms offered by the RF component.
- *Must be able to send jobs to VMs.* It can contact with the VMs and send jobs to execute on them.

- *Must be able to manage reservations for each VM.* It must be able to accommodate new reservations (agreement by which a client reserves in advance a VM) taking into account the current state of the reservation schedule.

3.2.3 Policy Manager requirements

Requirements of the Policy Manager are the following:

- *Must provide an API for resolving conflicts to the users of PM.* It consists of a software infrastructure that permits users of the PM to query it about decisions they have to take.

3.3 Non-functional requirements

Non-functional requirements are characteristics or constraints related to the quality that software must provide. In our prototype, these are:

- *Must work on heterogeneous resources.* A provider could offer resources from different processor architectures.
- *Must work on Debian based system.* For the purpose of this prototype, it is not necessary that the software have to be distribution independent. But it could be interesting that the software could be easily extensible to other distributions.
- *Resource Fabrics must be remotely accessible.* The resource fabrics component must be remotely accessible through web services inside the LAN of the provider.
- *Must be extensible.* This is a very important requirement because the software that we are developing is just a prototype that will be included in the SORMA's prototype, not a final commercial software. For this reason many changes are expected in the prototype.
- *The documentation must be clear.* The documentation, specially in the APIs of the components developed in the context of this PFC must be clear for the same reason as the previously point, for an easy extensibility.
- *Rule Engine independent.* Changing the Rule Engine must be smooth.
- *Must use the following software:*
  - Java 1.5: The generated software must be JAVA API 1.5 compliant.

- JESS: Java Expert System Shell. The PM component has to use this software as its rule engine to store and manage policies. We have chosen this rule engine because of its interoperability with Java.
- Xen: The hypervisor used for the virtualization. Xen is an open source virtualization software with a great amount of functions for managing virtual machines.
- Tycho: Communication platform used in SORMA to send jobs to VMs.
- Axis 2: SOAP engine used to build web services.
- XMLBeans: Web service data binding provided by Apache. It creates Java types based on XML schema.

Chapter 4

Specification

In this chapter we will introduce the API of the software components of the project as well as the actors that will use these elements. We provide the Resource Fabrics API as it is a meaningful component by itself ready to be used in other scenarios/projects. Anyway, this component is meant to be used by the ERM component within the context of this PFC.

For each component functionality, we provide the following information: function name, brief description, input and output parameters description and kind of exception that the function might through.

Most of the specification of the data types for the input and output parameters can be found in the appendix B. The rest is explained in detail in the Design chapter 6. This separation is due to the fact that those data types included in the appendix follows an XML schema and deserve an special attention as it is intended to be as standard as possible.

4.1 Actors

In this section we specify the actors which will interact with the project's components. As it is part of a bigger project, it is expected that all the actors will be software elements instead of humans:

EERM : SORMA's software component. It is explained briefly in chapter 1 and in further detail in chapter 3. We can find the following actors according to the EERM's subcomponent that they represent:

SPG : System Performance Guard. EERM software component that monitors the state of resources and take decisions consequently. It can act on behalf of the resource

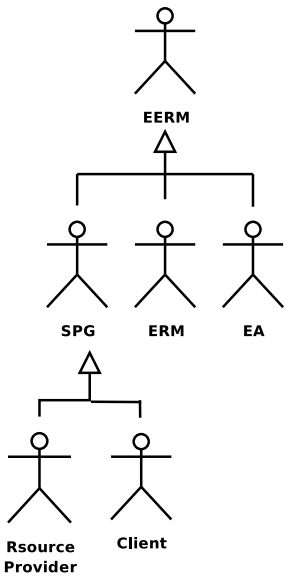


Figure 4.1: Actors hierarchy.

provider or on behalf of the client (resource consummer). Therefore, it is specialized in two different actors:

SPG Resource Provider : SPG EERM software component that may act on behalf of the resource provider.

SPG Client : SPG EERM software component that is acting on behalf of the final client.

ERM : Economic Resource Manager. EERM software component. (See section 3.1.2 for a component description).

EA : Economy Agent. EERM software component. (See section 3.1.2 for a component description).

4.2 Resource Fabrics specification

For the scope of this prototype, we only provide three functionalities for the Resource Fabrics component, the lowest layer that communicates with physical resources. These are: create a new VM and the necessary operations to manage them. At the moment, these management functions are start and shutdown VMs. However, in a commercial product based on this project this layer should also offer functions for suspending and resuming VMs, migrate VMs from one host to another, change the resources assigned to each VM (amount of memory, number of VCPUs, disk

space, etcetera), and functionalities for the management of jobs (cancelling tasks, suspending, resuming them and so on).

createMachine	
Description	Choose an available resource according to the processor architecture specified inside parameter *MachineDescription* and creates there a new virtual machine accomplishing with the other requirements also specified in the parameter *MachineDescription*.
Actors	ERM
Input	*MachineDescription*: specification of the requirements that the new machine must serve. See appendix B.
Output	*MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B.
Exception	*NoResourceAvailable*: there is no resource of the required architecture. *InvalidRequirement*: one of the specified requirements cannot be accomplished. *CommunicationFailed*: it is impossible to contact or transfer data to the target resource.

startMachine	
Description	Start the machine identified by *MachineIdentifier*.
Actors	ERM
Input	*MachineIdentifier*: list of the necessary information to manage and access to the VM. See appendix B.
Exception	*UnableToStart*: an error has occurred while starting the machine and it is impossible to start it. *MachineNotFound*: there is no machine identified by *MachineIdentifier* in this resource.

shutdownMachine	
Description	Shutdown the machine *MachineIdentifier*.
Actors	ERM
Input	*MachineIdentifier*: list of the necessary information to manage and access to the VM. See appendix B.
Exception	*UnableToShutdown*: an error has occurred while stopping the machine. *MachineNotFound*: there is no machine identified by *MachineIdentifier* in this resource.

4.3 Economic Resource Manager specification

In this section we introduce the ERM specification to help other EERM components to interact with it. This layer aims to provide functions for managing resources, more concretely functions to handle VMs such as create, start and shutdown a VM; methods for managing an efficient use of VMs such as handle its related information and schedule reservations; and lastly, functions for sending jobs to those VMS. However, in a real product, this API would include a bigger set of functions to manage VMs and the jobs that will be executed on them, for instance: pause job, migrate job, pausing/resume VM, etcetera.

createMachine	
Description	Choose an available resource according to the architecture required and creates there a new virtual machine accomplishing with the requirements specified in the parameters *HardwareDescription* and *ImageDescription*. This function also provides a unique name to the VM.
Actors	SPG Resource Provider
Input	*HardwareDescription*: description of the hardware required for the new VM. See appendix B. *ImageDescription*: description of the requirements for the construction of a disk image for the new VM. See appendix B.
Output	*MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B.
Exception	*NoResourceAvailable*: there is no resource specified for the required architecture. *InvalidRequirement*: one of the specified requirements cannot be accomplished. *CommunicationFailed*: it is impossible to contact or transfer data to the target resource.

startMachine	
Description	Start the machine identified by *MachineIdentifier*.
Actors	SPG Resource Provider
Input	*MachineIdentifier*: list of the necessary information to manage and access to the VM. See appendix B.
Output	*boolean*: whether machine is started properly or not.
Exception	*UnableToStart*: an error has occurred while starting the machine and it is impossible to start it. *MachineNotFound*: there is no machine identified by *MachineIdentifier* in the target resource. *CommunicationFailed*: it is impossible to contact or transfer data to the target resource. *VMDoesNotExist*: there is no VM identified by *MachineIdentifier*.

shutdownMachine	
Description	Shutdown the machine identified by *MachineIdentifier*.
Actors	SPG Resource Provider
Input	*MachineIdentifier*: list of the necessary information to manage and access to the VM. See appendix B.
Output	*boolean*: whether machine is stopped properly or not.
Exception	*UnableToShutdown*: an error has occurred while starting the machine and it is impossible to start it. *MachineNotFound*: there is no machine identified by *MachineIdentifier* in the target resource. *CommunicationFailed*: it is impossible to contact or transfer data to the target resource. *VMDoesNotExist*: there is no VM identified by *MachineIdentifier*.

getResources	
Description	Returns the set of provider's VMs.
Actors	SPG
Output	*Set(MachineIdentifier)*: a list of MachineIdentifiers of all the available VMs.

getResourceInfo	
Description	Request for information about the VM identified by *MachineIdentifier*.
Actors	SPG
Input	*MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B.
Output	*MachineInfo*: information related to a VM, its hardware, the software it has installed and its reservations plan.
Exception	*VMDoesNotExist*: there is no VM identified by *MachineIdentifier*.

reserveResource	
Description	Reserves the resource identified by *MachineIdentifier* fullfiling the characteristics described in *Reservation* (e.g. period of time).
Actors	SPG Client
Input	*MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B. *Reservation*: information about the reservation.
Output	*boolean*: whether the reservation can be planned properly or not.
Exception	*VMDoesNotExist*: there is no VM identified by *MachineIdentifier*.

getReservations	
Description	Request for the list of reservation of the resource identified by *MachineIdentifier*.
Actors	SPG
Input	*MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B.
Output	*ReservationSet*: register of all the reservations of a resource.
Exception	*VMDoesNotExist*: there is no VM identified by *MachineIdentifier*.

sendJob	
Description	This method sends the job specified in the *JSDL* parameter by client *Client* to the VM identified by *MachineIdentifier*.
Actors	SPG Client
Input	*Client:* system consumer. *MachineIdentifier*: list of the necessary information to manage and access to the new machine. See appendix B. *JSDL:* job description. (See [14]).
Output	*SentJobId*: job identifier.
Exception	*VMDoesNotExist*: there is no VM identified by *MachineIdentifier*. *NoClientReservation*: there is no reservation of this client on this VM. *CommunicationFailed*: it is impossible to contact or transfer data to the target resource.

4.4 Policy Manager specification

This component is in charge of managing policies to help EERM to take decisions. We provide these functions to fullfil with the requirement of being RE independent (see Requirements chapter 3). The functionalities listed below are enough for accomplishing our necessities in this prototype. Nevertheless, a complete software should provide operations for managing modules, facts, etc.

executePolicy	
Description	Executes a policy which helps to take some decision or to resolve a conflict.
Actors	SPG, EA, ERM.
Input	*Policy*: policy for a conflict resolution. See Design chapter 6.
Exception	*RuleEngineNotFound*: the expert system is not specified properly in the configuration file. *PoliciesNotFound*: the policies location is not specified properly in the configuration file. *RuleEngineException*: any fault related to the rule engine.

Chapter 5

Architecture

In this chapter, we present the scope of the architecture for this project. Concretely, we explain the logical architecture of the Resource Fabrics, Economic Resource Manager and Policy Manager components. To shed light on this chapter, every section comes with a figure showing the relations and dependencies between elements.

As in previous chapters, we present a bottom-up structure. As depicted in figure 5.1 and as we known for previous chapters, the ERM component is the central component which receives requests for managing resources and forwards them to resource fabrics layer when necessary, and in case of any foreseen conflict asks PM to resolve it.

Figure 5.1: Architecture.

5.1 Resource Fabrics architecture

Resource Fabrics is the component that provides access to and control of the resources inside a resource provider's LAN.

As shown in figure 5.2 Resource Fabrics component is formed by two main subcomponents: the Resource Coordinator and the Local Virtualization Manager which are explained in detail in the next sections. On one hand, **Resource Coordinator** chooses and helps to prepare a resource for creating new virtual machines. On the other hand, the **Local Virtualization Manager** is placed in every resource to offer functionalities for the management of its virtual machines.

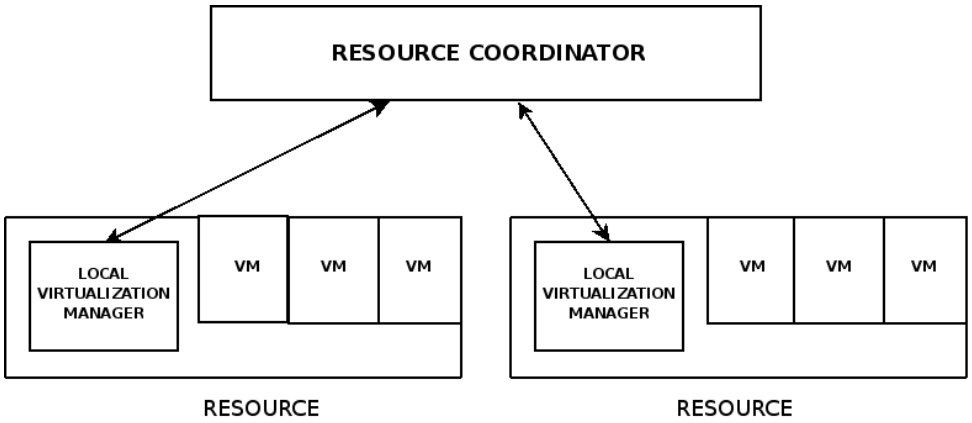


Figure 5.2: Resource Fabrics architecture.

5.1.1 Resource Coordinator

The Resource Coordinator is built upon the following elements (see figure 5.3).

The **Virtual Machine Creator** is in charge of the correct creation of virtual machines. For this purpose, it contacts the Resource Registry, Repository Image and Transfer Manager components. It also contacts with the Local Virtualization Manager explained in the next subsection.

The **Resource Registry** component maintains a list of the available resources and theirs characteristics, such as the architecture, default user and so on.

The **Repository Image** is formed by two elements: the **Store** in which we cache base systems, kernels and software packages for the disks images, and the **Image Builder** which is responsible of maintaining and filling the Store.

Finally, the **Transfer Manager** is the component who transfers disk images created by Repository Image to the Local Virtualization Manager explained in the next subsection.

The latest three components, Resource Registry, Repository Image and Transfer Manager are coordinated by the first one, Virtual Machine Creator.

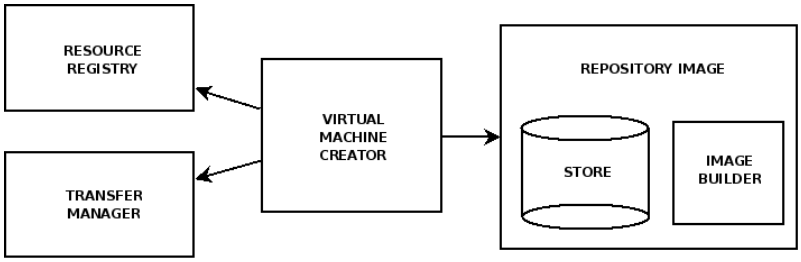


Figure 5.3: Resource Coordinator architecture.

5.1.2 Local Virtualization Manager

As in the previous section, we will define the responsibilities and capabilities of the Local Virtualization Manager in terms of the components that conform it.

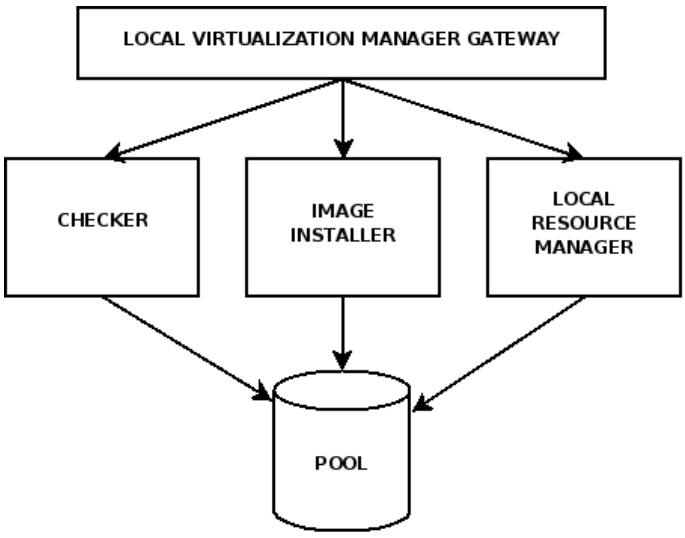


Figure 5.4: Local Virtualization Manager architecture.

The **Local Virtualization Manager Gateway** is the access point to the local services. Every incoming request has to pass through this component. It do not perform any action, just acts as a gateway.

The role of the **Pool** in the Local Virtualization Manager is similar to the Store module in the Resource Coordinator component. Within the pool, we store the most recently used kernels and base systems as well as all the necessary information for the correct operation of VMs, i.e. , its

disk images and configurations files.

The **Checker** component is accessed during the creation of a new machine. It checks if the machine creation requirements are available in the Pool cache. That is, whether the required kernel and base system are cached locally or not, and reserve a space in the Pool for the new machine.

The **Image Installer** element has the onus of installing the base system, kernel, and create the necessary ramdisk for a VM creation.

Finally, the **Local Resource Manager** is the component in charge of managing the VMs. For this prototype it just offer create, start and stop capabilities, but in a final prototype or in a commercial product it is supposed to provide functions for migrating and redimensioning VMs.

5.2 Economic Resource Manager architecture

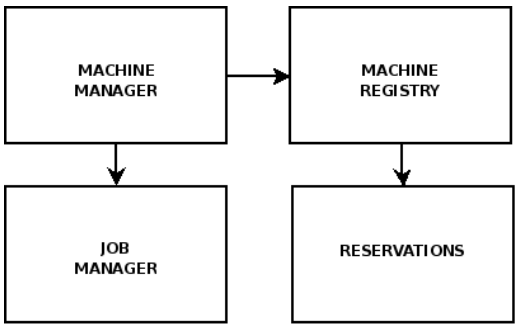


Figure 5.5: Economic Resource Manager architecture.

This component is in charge of the efficient use of the set of virtual machines created in the hardware resources offered by a provider.

As depicted in figure 5.5, the central architectural element of ERM is the **Machine Manager** component. This, has the onus to orchestrate the rest of the elements and it is also in charge of assigning unique names to virtual machines and to request the creation and destruction of them.

The **Machine Registry** maintains a register with the available virtual machines and information related to them. Some of this information is: the mechanism to contact the VM, the software installed on it, and its reservations. A user must reserve a VM before he could execute a job on it.

The **Reservations** component registers the reservations of each machine. It is the responsible for deciding whether a reservation is feasible or not and schedule it in the timetable associated to each machine.

Lastly, the **Job Manager** contacts VMs to send them jobs and to manage these jobs.

5.3 Policy Manager architecture

Finally, the Policy Manager architecture is presented. It consists of a **Rule Engine** and a **Policy Manager Frontend** to operate with the Rule Engine in a transparent way.

The architecture of this component naturally arise to satisfy the requirement of being Rule Engine independent (see chapter 3). That is, the Rule Engine might be changed by another one easily. To achieve this requirement the front-end isolates the Rule Engine from the way to access to it.

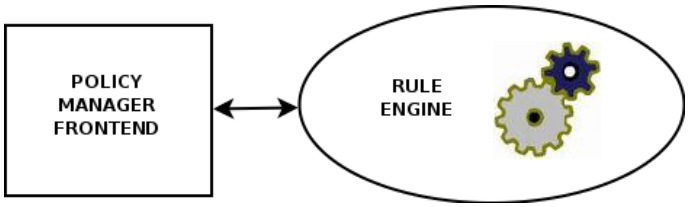


Figure 5.6: Policy Manager architecture.

Chapter 6

Design

This chapter presents the main design decisions related to project's components (RF, ERM, PM). It is divided in two parts. On one side, we describe the design solution for each component in terms of class diagrams including the design patterns that we have applied. In the second part, we explain our design in terms of the most interesting use cases to help the reader to grasp the interactions among components.

During the design phase of the top level components, both ERM and PM, we always had in mind that they are EERM's components. Therefore, they must be designed to help EERM to achieve its goals, maximizing provider's revenue and client's and provider's satisfaction.

6.1 Resource Fabrics design

The Resource Fabrics component is the closest element to physical resources. One of its responsabilities consists of transferring software packages from the Resource Coordinator component to the target Local Virtualization Manager. In this section, we explain how this transfer is designed. The rest of components, specially those placed inside the Local Virtualization Manager, are explained directly in the implementation chapter (see chapter 7) as most of them are not object oriented components and their designs are very tightly coupled to implementation decisions.

6.1.1 Image Transfer

This section corresponds to the explanation of the design of the Transfer Manager architectural component placed inside the Resource Coordinator component. (See Architecture chapter 5).

In this project, the provider of the bundle of resources could specify which data transfer mechanism (i.e. which protocol: SCP, FTP, etcetera) will use for transferring disk images to each resource (see System Evaluation chapter 8 for a protocol comparison discussion). The resource provider specifies the protocol in the *resources* XML file, detailed in the appendix section A.3.2. Thus, the application is able to know about the transfer protocol at runtime.

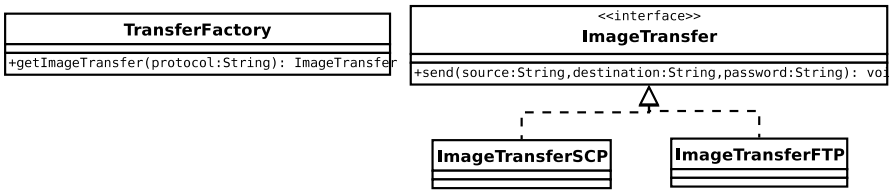


Figure 6.1: Image Transfer factory diagram.

This capability lead us to use the factory design pattern to resolve the problem of which image transfer class we instantiate for the image transfer at runtime. Figure 6.1 shows how we apply this pattern. In this figure we show a *Tranfer Factory* class with a static method *getImageTransfer(String protocol)* and two implementations of the interface *ImageTransfer* used for transferring files from a component to another. Thereafter, if we want to use the SCP protocol for the transfer, we just only have to call *TransferFactory.getImageTransfer("SCP")* and this will return us an interface implemented with the SCP mechanism.

6.2 Economic Resource Manager design

The EERM component, as well as its subcomponent ERM have to guarantee an efficient use of the marketable resources. To achieve this goal we propose an extensible and policy-driven design for reserving these resources.

6.2.1 Virtual machines reservations

First of all, we have to introduce the definitions of reservation and planned reservation. We define a reservation as the client's intention to reserve a VM for a period of time. We will say that this reservation is planned/scheduled to the agreement by which the client reserves in advance a VM to use it from a period of time, i.e. the reservation is scheduled in the reservation timetable of a VM without overlapping in time with any other reservation. Therefore, a client must had made a reservation for a VM before he could send a job to it. In this section, we propose an extensible design for the reservation mechanism. It allows the clients to specify reservations in several ways depending on the inherent characteristics of their resource necessities.

For this prototype we have defined two kinds of reservations depending on the way they can be scheduled:

Fixed: a reservation specified in terms of the beginning date-time and the ending date-time. That is, the reservation is feasible if it is scheduled within those dates. For instance, a client may ask for a reservation from Monday, April 21 2008 at 11:40 AM to Monday, April 28 2008 at 11:40 AM.

Interval: a reservation specified by means of its duration time and a lower and upper bound where it is feasible to plan it. That is, the booking can be performed if it is scheduled in the timetable after the lowerbound and before the upperbound for a *duration* period of time. For example, a client might want a reservation for a 20 hours period between Monday, April 21 2008 at 11:40 AM and Saturday, April 26 2008 at 12:00 AM, but no matter when this booking will be done exactly. Thus, we said that this kind of reservation is *mobile*, because we can reschedule it.

Scheduling process restrictions

In the project's framework, we have imposed some restrictions to the timetable scheduling process to simplify its complexity because fitting reservations in a timetable could be a computational complex problem. Thus, when we are trying to plan a reservation R in a virtual machine timetable, we have to take care about possible conflicts and restrictions:

[1] If R is overlapping with an already started reservation (this is, the beginning reservation's date is in the past), R cannot be planned.

[2] If R is overlapping with more than one scheduled reservations, R cannot be planned. We could have a policy specifying the maximum number of overlapped reservations as is suggested in Conclusions and Future Work chapter 11.

[3] If R is overlapping with just one reservation S:

  [3.1] if S is of type Fixed, the Policy Manager is called to resolve the conflict between R and S. This decision will be taken in terms of which client, R's or S' client, is the most preferential. In case of equality between client's preferences the reservation S will rest planned. We should understand client preference's as the grade in which the provider wants to encourage or maintain client's loyalty.

  [3.2] if S is mobile, such as an Interval reservation, we will try to reschedule it inside its defined-range. If it could not be rescheduled, we call the PM to resolve the conflict

as in the case described before. For the sake of simplicity, we only consider one reservation rescheduling for a given reservation overlap.

To sum up, we are encouraging to build client's loyalty rather than anything else (in case of conflict between reservations we choose to plan that one owned by the most preferential client). Although this policy might seem to be opposite to the EERM goal of maximizing revenue, it is not because a client's loyalty could bring us higher revenue on the long-term. Even though, in the Conclusions and Future Work chapter 11 we discuss about taking into account the current reservation revenue besides the client's preference grade when deciding about a conflict resolution.

Design

The design of these reservations is shown in figure 6.2. We can see that each machine has its timetable of booked reservations. As each reservation has its own mechanism to be scheduled on a timetable, our design approach consists of an abstract class *Reservation* where the common part is defined and two subclasses for the different types of reservations. The *schedule* function is implemented into the specialized reservation classes. As it can be seen, this solution is based on the *template pattern*. In our case, we call the method *reserve* when we want to plan a reservation. This function is in charge of doing the first common part (if the timetable is empty we append the reservation without taking any other consideration) and call to specialized function *schedule*. It trys to schedule the reservation in the timetable and calls to method *resolve* of the superclass (that function is the hook of the template) to resolve a reservation conflict when necessary.

Reservations are planned in the first empty timetable's gap when possible. Other approaches could be taken into account, as schedule reservations in the hole that they fits better or in the biggest hole and so on. The study and evaluation of which approach could be better for this system is out of the project's scope even so, we suggest some alternatives to this decision in the Conclusions and Future Work chapter 11.

In the figure 6.2, a Client class is depicted. This is a simplification used by this project because the onus of representing clients belongs to another SORMA's partner.

This design permits ERM to manage resources more efficiently than if it just have had reservations specified with a beginning and ending date. This improvement is quantified in the System Evaluation chapter 8. It also allows to be extended easily in case we want to specify bookings in any other way. For example, a client might want 3 hours of duration as soon as possible or as last as possible, etc.

We refer the reader to Conclusions and Future Work chapter 11 for an explanation of different

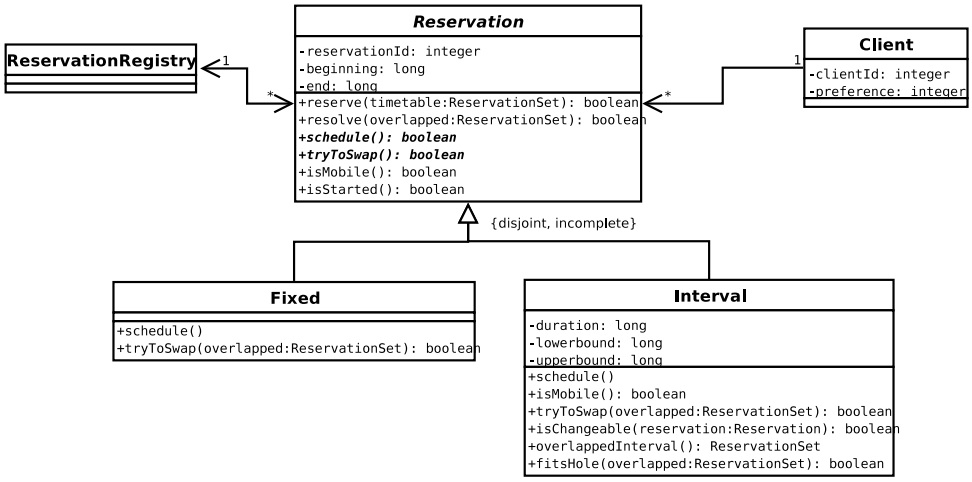


Figure 6.2: Reservations mechanism design. Application of the template pattern in reservations. Getters and setters methods are omitted from the figure.

alternatives to plan reservations that should be evaluated in the future. Besides, there can be found several policies examples in addition the one presented in the following section 6.3. Later on this chapter, it is presented the use case *Reserve Resource* for a complete comprehension of the reservation mechanism.

6.3 Policy Manager design

This component is the one that interacts with the RE. For this reason and to fullfil with the requirement (see chapter 3) of being RE independent, we designed the PM component as depicted in figure 6.3.

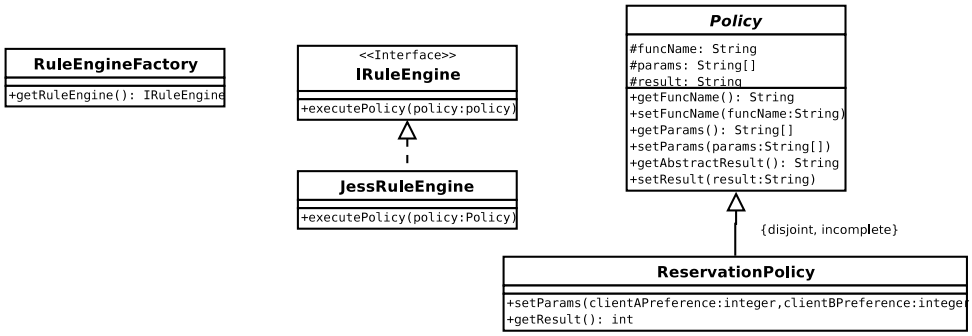


Figure 6.3: Policy Manager[1] diagram.

We have an interface *IRuleEngine* that acts as a connector to the RE. For each different RE we want to use for the system, we have to provide an implementation of this interface. Therefore,

we use again the *factory pattern* in our proposal. We think that could be useful to allow changing from one RE to another without compiling the source code in a development context, although we consider that almost never it will be necessary to change between Rule Engine implementations on runtime in a real scenario.

On the one hand, we have a *RuleEngineFactory* which return us an interface through which we can work with the RE implementation. This class is in charge of controlling that a unique instance of a *IRuleEngine* implementation exists in the system. To decide which implementation of the interface we want to instantiate we query it to a configuration file (see appendix A), in a different way to what we have done in the case of Image Transfer factory explained previously in this chapter.

Finally, we have the *Policy* abstract class that represents each provider's policy specified in the RE language. The objective was to provide an infrastructure through which the rest of EERM components could use PM, not to do policies themselves. Thus, we just provide a simple example of policy introduced in the previous section. It consists of resolving a conflict between two reservations when planning them by means of the clients' reservations preference. In the next chapter we show the policy implementation (see chapter 7) and throughout the Conclusions and Future Work chapter 11 we propose several policies that may be applied to EERM.

To sum up, if we want to change the current RE by another implementation of RE we just have to implement the *IRuleEngine* interface. And for each policy that we add to the RE we have to create its wrappered class extending the abstract class *Policy*.

6.4 Use cases

To clarify the functionalities provided by our prototype, we explain the most representative use cases, those which are more complex regarding the number of interactions between classes and more interesting from the functional point of view.

6.4.1 Create Machine

The use case *Create Machine* is used when creating a new virtualized resource is required (see specification chapter 4 for the details of this functionality). The architectural components involved in this use case are ERM and RF.

We present a simplified sequence diagram of this use case in figures 6.4 and 6.5. These are simplified due to the omission of parameters types, the omission of singleton patterns as in the case of *ResourceRegistry*, *PoolLock*, *StoreLock*, etc. and the omission of error cases. Although,

the parameter's types could be deduced from the parameters name, the reader can refer to appendix B for a specification of the most important parameter types.

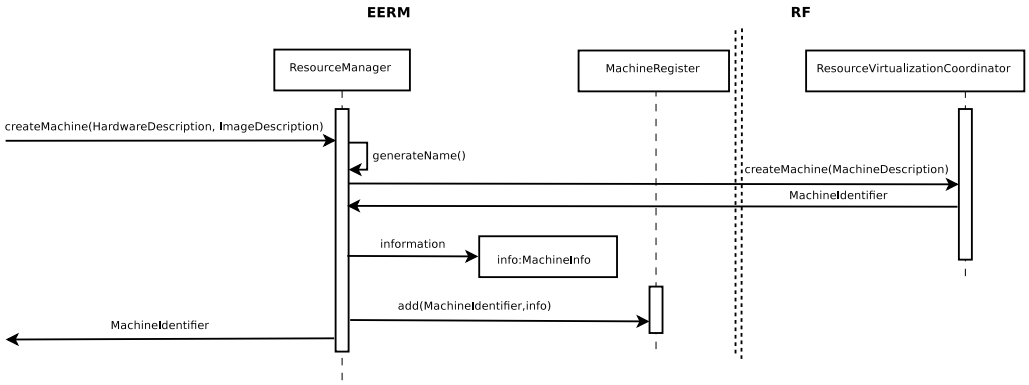


Figure 6.4: Create Machine use case sequence diagram. Economic Resource Manager fragment.

Once a create machine request arrives to the ERM component, the *ResourceManager* class generates a unique name for the new virtual machine and forwards the call to the *ResourceVirtualizationCoordinator* of the Resource Fabric component. As soon as the RF component returns, the *ResourceManager* creates an instance of *MachineInfo* and fills it with information about the characteristics and reservations of the new virtual machine. Finally it registers the new machine in the *MahineRegister*. This process is depicted in figure 6.4.

What is happening inside RF once a request has arrived is shown in figure 6.5. The *ResourceVirtualizationCoordinator* class contacts with the ResourceRegistry for a resource where to create the new virtual machine (if no resource exists an error message is returned). Then, it forwards the call to the *ImageCoordinator*. It calls the *ImagePoolManager* from the Local Virtualization Manager of the target resource to find out which requirements it can accomplish without downloading anything from Internet. Then if there exists a requirement that cannot be fullfilled by the target resource, it asks for the software required to build an image to the *ImageStoreManager* in mutual exclusion to ensure no interference of concurrent requests. After that, it transfers that software to the target resource, in this example, using the SCP protocol. Then it asks, in mutual exclusion, to the *ImagePoolManager* to install the disk image and finally it asks, again in mutual exclusion, to the *VirtualizationManager* for the virtual machine creation strictly speaking. In case the Local Virtualization Manager could satisfy all the requirements, the calls to methods *buildImage, getImageTransfer* and *send* are skipped.

The interaction between the previous explained classes and the Image Builder, Checker, Image Installer and Local Resource Manager components is explained in the next chapter 7 because these components follow a non object oriented design.

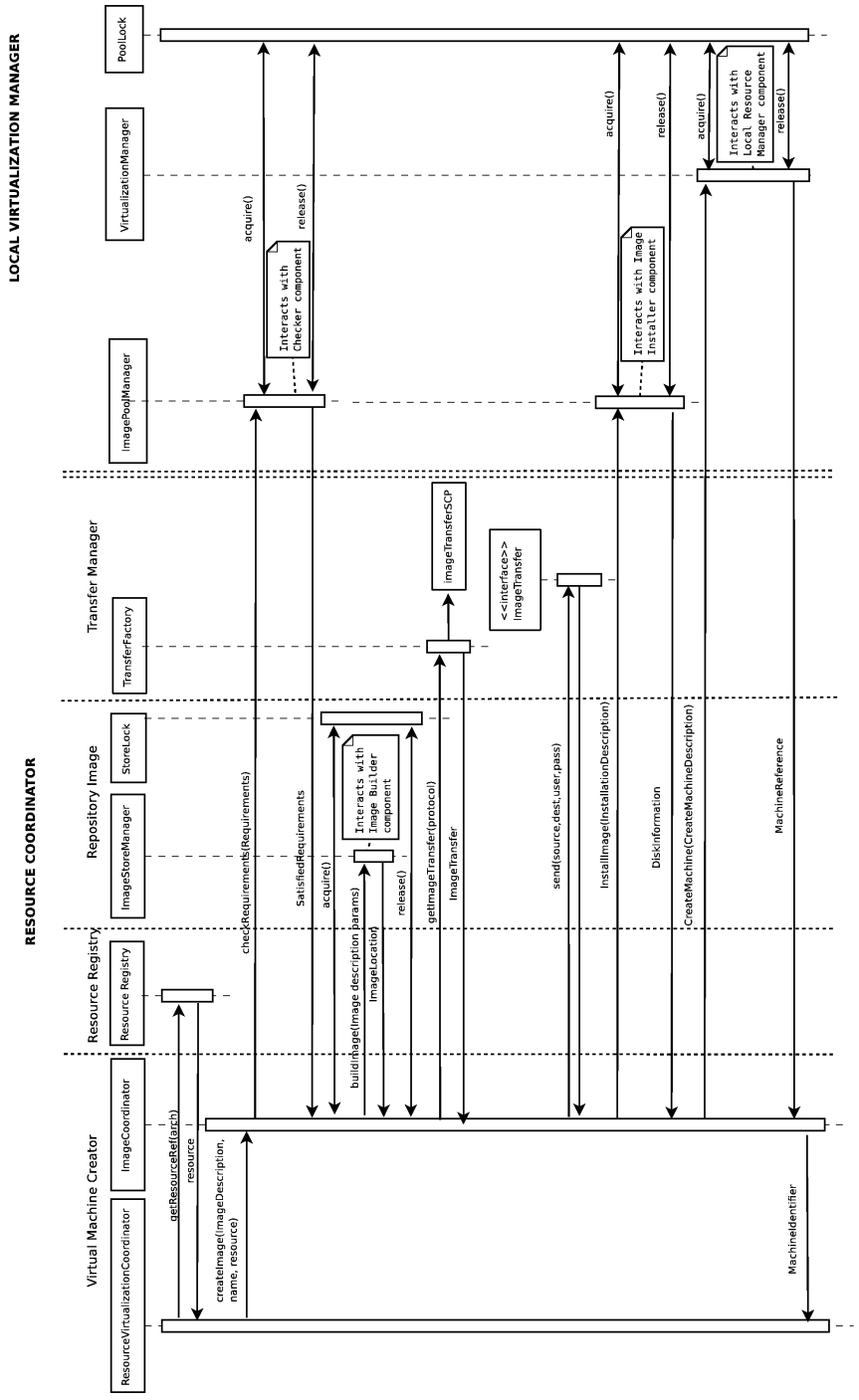


Figure 6.5: Create Machine use case sequence diagram. Resource Coordinator fragment.

6.4.2 Reserve Resource

This section presents the Reserve Resource use case as shown in figures 6.6, 6.7 and 6.8.

The use case starts when a *reserveResource* request arrives at *ResourceManager*. This request specifies the VM where to perform the reservation and the reservation going to be scheduled. If this VM does not exist, it is not registered in the *MachineRegister*, an error message is returned. Otherwise, the *ResourceManager* gets its plan (that is, the reservation timetable) and calls the *reserve* function. If the plan is empty this reservation is planned immediately. Otherwise, the method *schedule* is called to plan the reservation. This *schedule* method returns whether the reservation has been planned or not.

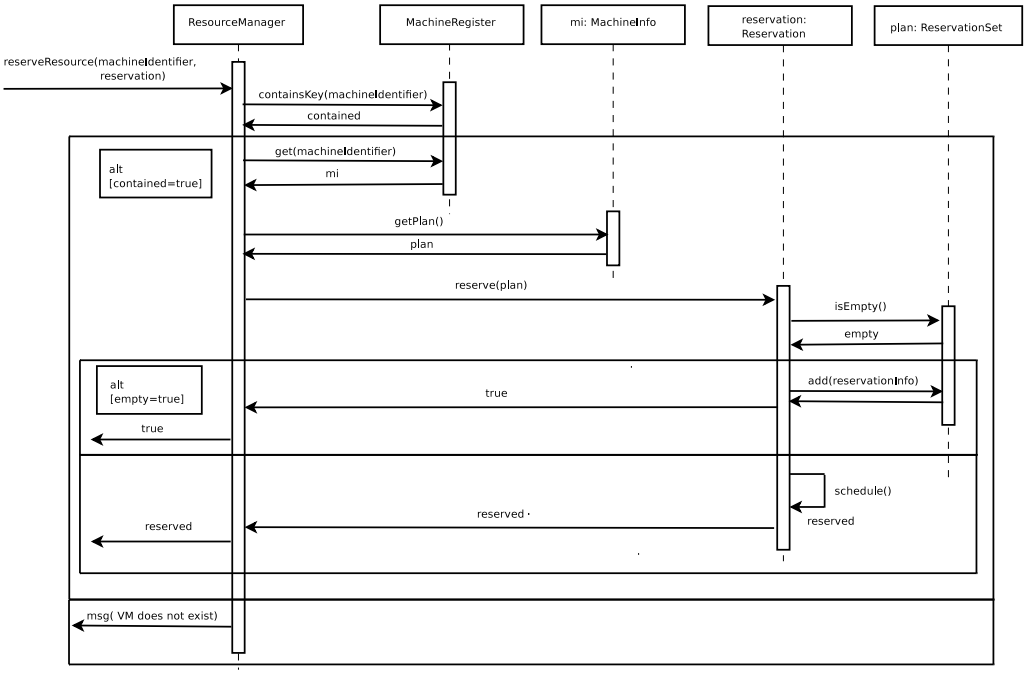
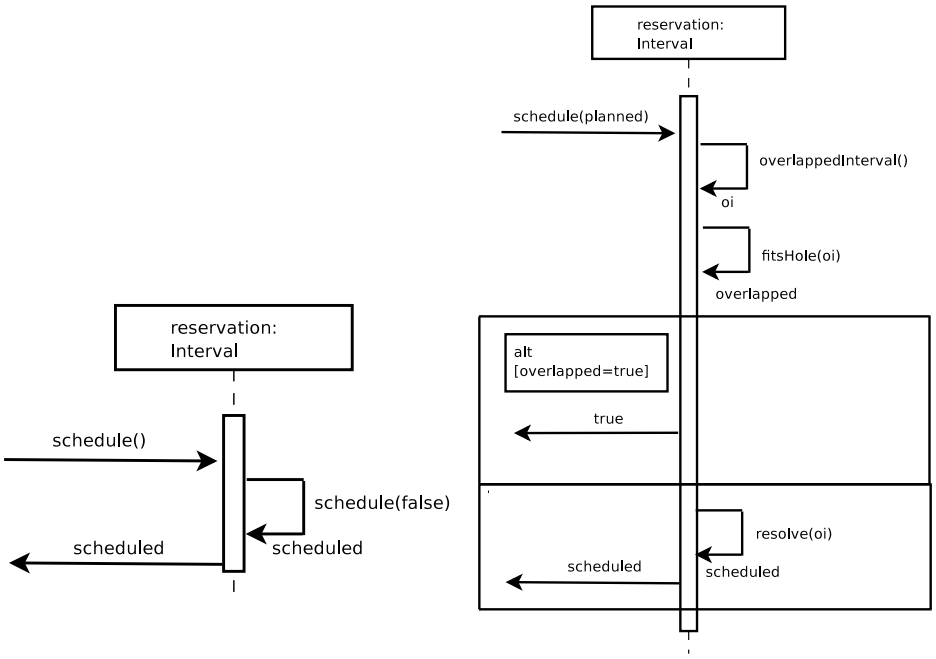


Figure 6.6: Reserve Resource use case sequence diagram. Method reserve.

Figure 6.7 shows the behaviour of method schedule for the case that we are trying to plan an Interval reservation. We depict this case because it is more complex than the fixed one. In 6.7(a) we call to the private schedule method rendered in 6.7(b) advertising that the implicit reservation is not planned (we will call this function with a true value in case we want to reschedule a reservation). Then, the Interval reservation calls to method *overlappedInterval* to calculate with which reservation its range overlaps, i.e. which reservations are already planned between its lower and upperbound. After this, the *fitsHole* function is called to plan the reservation in any empty gap between the planned reservations. In the case that the *fitsHole* function could schedule the reservation, this *schedule* function will return true, otherwise that means that it is necessary to call to

method *resolve* for deciding between this implicit reservation and the ones already planned.



(a) Method schedule (b) Private method schedule

Figure 6.7: Reserve Resource use case sequence diagram. Method schedule.

Finally, the method *resolve* is shown in figure 6.8. Firstly, it calls to method *tryToSwap* trying to reschedule any reservation inside its feasible range as well as plan the implicit reservation and keep the planned ones. If it is not possible, for each reservation that could be changed by that one we are trying to plan, Policy Manager is called (*executePolicy* method) to resolve between them. When PM will resolve the conflict in favour of the implicit reservation this method will swap the reservations and return true. If the implicit reservation cannot be changed for any planned reservation, false is returned.

6.4.3 Send Job

This section shows the interactions among ERM's classes when it is required to send a job to a VM. Once a client has made a reservation over a VM he can send a job there. Figure 6.9 illustrates this use case. First of all, the *Resource Manager* class from the ERM component asks to the *Machine Register* whether the required VM exists or not. If it does not exist an error message is returned, otherwise the Resource Manager queries for the reservation plan of the VM and asks to it whether this client has a reservation for it or not. If the client has no reservation for the VM an error message is returned, otherwise the job is sent to the target VM through the *Tycho*[2] classes.

[2] *TychoManager* and *TychoResourceWrapper* classes have been already done in SORMA's project.

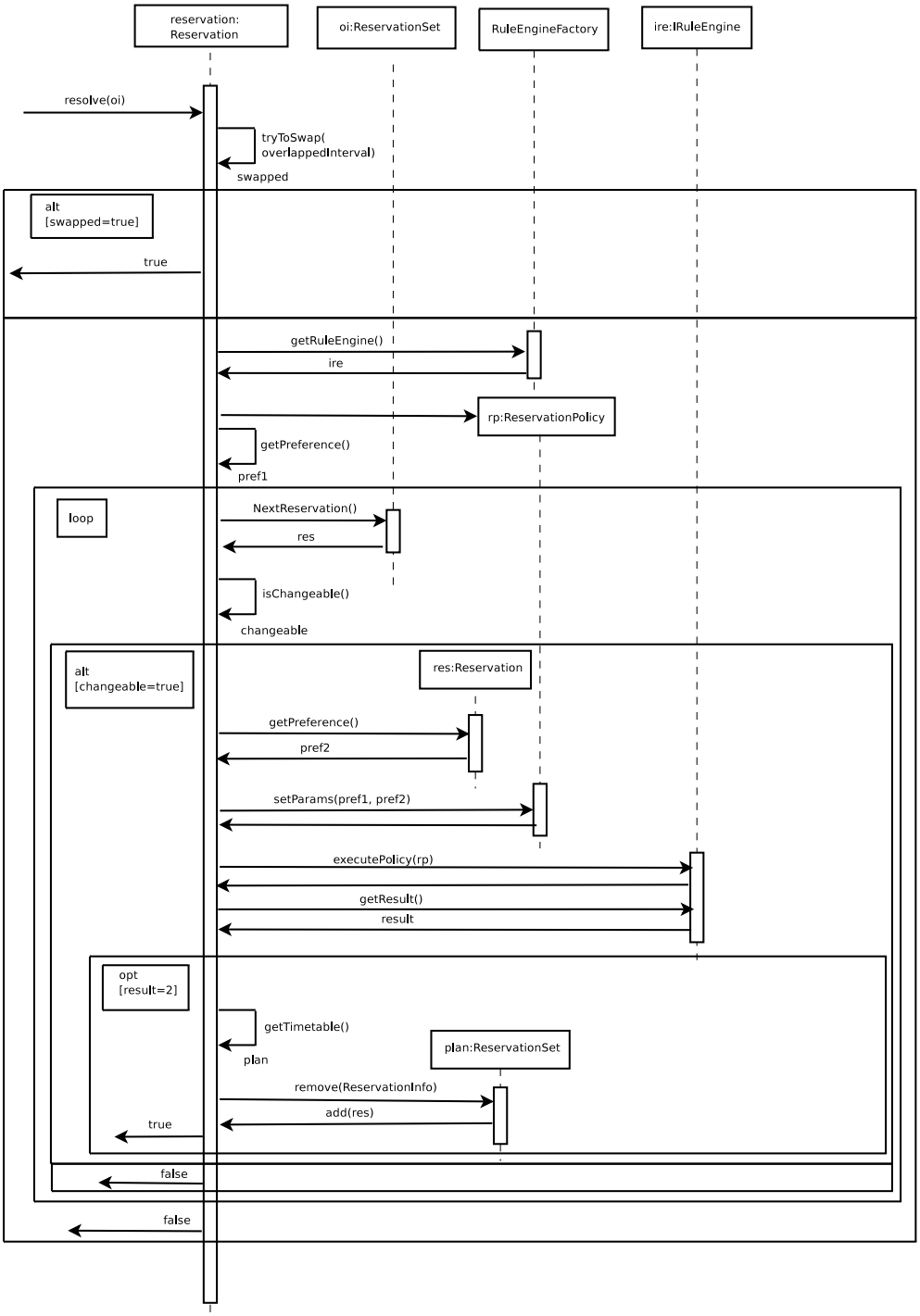


Figure 6.8: Reserve Resource use case sequence diagram. Method resolve.

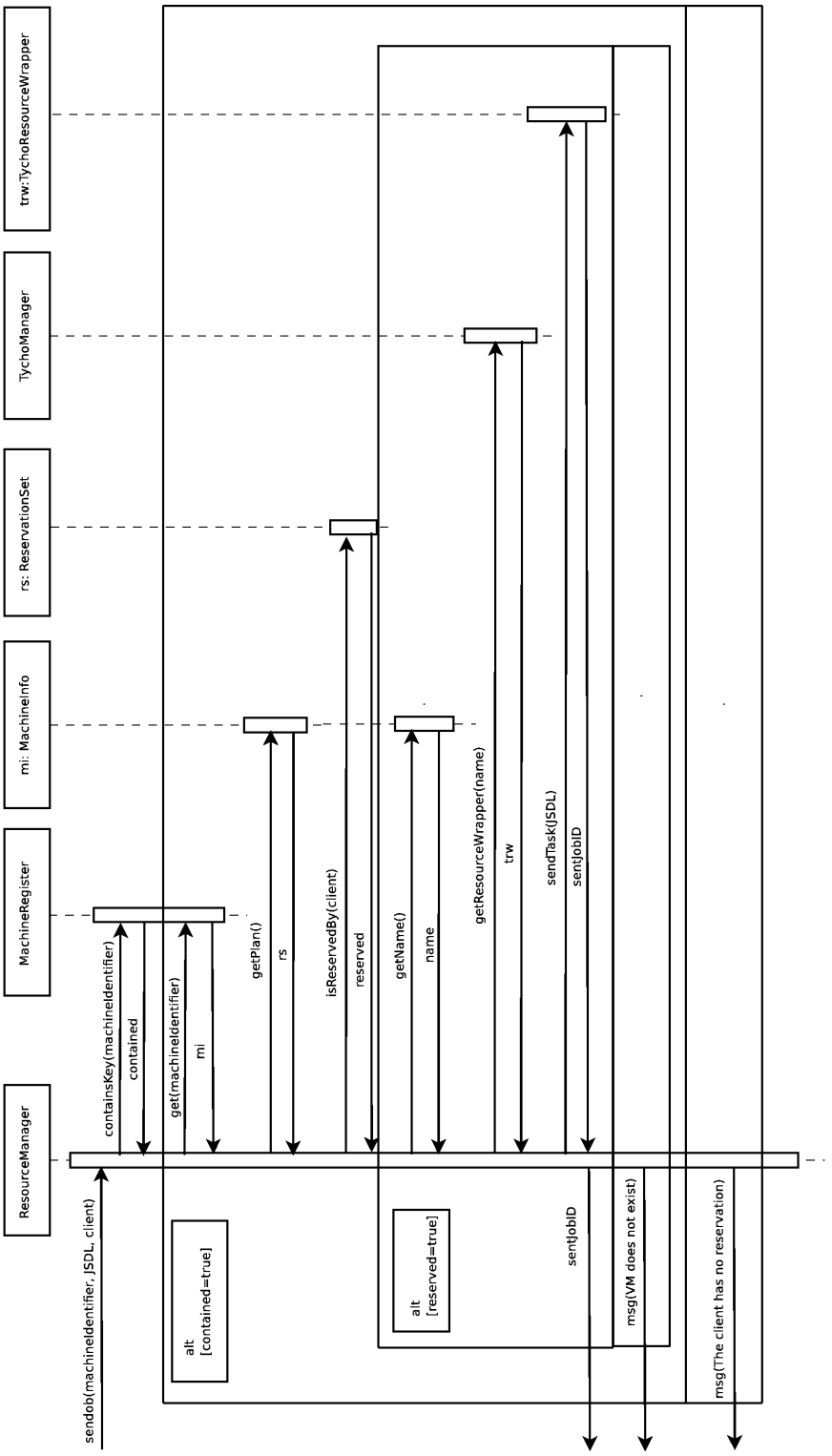


Figure 6.9: Send Job use case sequence diagram.

Chapter 7

Implementation

So far, we have explained the overall project without concerning about the technologies involved in it. In this chapter, we present these technologies and the most relevant implementation details related to them. This chapter is structured as the previous ones, in terms of project components. However, first of all, we will explain those characteristics that are present in more than one component.

7.1 Common issues

In these section we introduce all those implementation decisions that affect more than one project component.

First of all, we have decided to implement the project components in Java 1.5 as this was agreed in the context of SORMA's project. Nevertheless, some subcomponents of RF are written in bash script due to its proximity to the OS (see section 7.2).

Secondly, we have implemented synchronous stateless Web Services (WS) for the communication among remote components, those that may be deployed in separate servers (see chapter A). These are generated with Axis2 [15] which is a toolkit from Apache Software Foundation [16] for developing and deploying WSs. Concretely, we specify these WSs using WSDL (Web Service Description Language) and then, we use the Axis utility wsdl2Java to construct the client stubs and server skeletons for the marshalling and unmarshalling of the service parameters. These parameters, are those specified by XML schemas in the appendix B. On the other hand, we use XMLBeans [17] from Apache Software Foundation as data binding because it translates XML schemas to Java types, so within the Java code we can manipulate these parameters without deal-

ing directly with the XML data representation. We use the synchronous version of these WSs because the requests need to be blocking.

We have implemented several registers in this project: the Resource Registry which registers hardware resources within Resource Coordinator RF component and Machine Registry inside ERM component which registers those VMs created in the resources. Those are implemented transient, just as Java HashMaps. Although, in future versions of this prototype, more sophisticated registers will be required, leastways persistents because it is indispensable that the registers will be persistent to avoid data loses in case of failure or service maintenance.

Finally, as we want to construct a high configurable software, we use a Java properties file to parameterize some decisions. This configuration file is shown in appendix A.

7.2 Resource Fabrics implementation

In this section, we will focus our attention in the Resource Fabrics layer. This is the component in charge of creating VM using Xen virtualization [2]. Xen is an open source hypervisor that offers low overhead even though without hardware support for virtualization.

Firstly, we are going to explain how the provider can communicate its available resources to the software components. Then we will center the discussion in the scripts deployed within RF component, including the following components: Checker, ImageBuilder, ImageInstaller and Local Resource Manager[1]. Then, we comment how we implement the software transfer between the Store and Pool elements and finally, we list the characteristics that a VM will offer.

Advertising resources

The provider specifies its resources in a XML file to advertise them (see appendix A). This way, the Resource Registry can be fullfiled with the information provided in this file. This register is implemented in Java and uses JDOM, a Java representation of an XML document [19], to manipulate the XML encoded data.

[1] A previous version of the Local Resource Manager was already done in the context of BREIN project [18]. This has been used as a starting point to do the current Local Resource Manager.

The scripts

The corpus of RF is formed by the Checker, ImageBuilder, ImageInstaller and Local Resource Manager components as far as they do almost all the component's logic. These components perform low level actions close to the operating system such as: operations over the file system, creating an empty file to accommodate a new file system for a new VM, etc. For that reason, we decided to do these actions in a scripting language, in this case bash, because it has more functionalities than Java to deal with low level OS details. Next, we will explain how we communicate these elements with the ones implemented in Java (the Local Virtualization Manager Gateway and the Virtual Machine Creator).

To achieve this communication, Java supplies us with a singleton Runtime instance, through which we can execute anything over the operating system and treat it as a process. When a process is about to die it can send an exit code to his father. We use this code, as most of the GNU-commands do, to notify whether a certain command has finished successfully (return code 0) or not (return code different from 0). The inconvenient of this kind of communication is that the code is an integer number belonging into the range [0,255]. That allows us to define 255 error messages but only one message in case of success. Because of this limitation, we use another kind of communication, using the standard channel. This way we can send any string which can be defined at execution time.

To sum up, we use different mechanisms to get the results from scripts. On the one hand, the exit code to notify whether the script is successfully executed or not and on the other hand, the standard output to inform about the localization of the data in case of success or the failure message in case of error. For the sake of clarity, the following example pretend to show how this communication can be used:

```
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec( "test.sh", "parameter");
proc.waitFor();
int ret = proc.exitValue();
byte[] mesg = new byte[100];
proc.getInputStream().read(mesg);
String smesg = new String(mesg);
System.out.println("script exit message "+smesg);
System.out.println("script exit value "+ret);
```

If the *test.sh* script is like this:

```
#!/bin/bash
echo $1
exit 111
```

The output of executing this program will be:

```
script exit message parameter
script exit value 111
```

The reader can refer to the Architecture chapter 5 for a description of the scripts functionalities or to appendix C for the precise specification of the actions that can perform each script. Here, we explain these components from an implementation point of view. Next, there are listed several tools that have been used through the implementation of our script solution:

debootstrap: It is used to create a Debian base system from scratch. It does this by downloading .deb files from a mirror site, and carefully unpacking them into a directory which can be chrooted into (i.e. change the root directory of the file system). [20]

wget: It is an utility to retrieve files from the Web using HTTP(s) and FTP protocols. This utility is used by debootstrap to download software packages. We do not use it directly because its use is very tightly coupled to the Linux repository structure. [21]

dpkg: It is a package that contains the low-level commands for handling the installation and removal of packages on the system. It is used for installing the kernel and its modules properly. [22]

apt*: This is Debian's front-end for the dpkg package manager. It provides the apt-get utility and APT dselect method that provides a simpler, safer way to install and upgrade packages. It is used for downloading those packages required by the Client. [23]

Those utilities are provided in any basic Debian-based distribution, except wget that is provided in any Linux basic system. Nevertheless, they can be installed over a non Debian based distribution. Therefore, our scripts are not coupled to Debian, although we recommend to use a Debian base system in the provider's deployment because they use packages developed by Debian's community. Besides, over a Debian OS we can install typical utilities from other distribution. For instance, if we want to construct an rpm-based system we can use rpmstrap (similar to debootstrap) and yum (similar to apt) utilities running over a Debian system. Thereafter, to permit our scripts to be extended in this sense, we use what is often termed a hook to wrap these utilities (see appendix C for the hook's description).

Storage

Two storage components exist inside RF, namely Pool and Store, that have to maintain a cache with kernels, releases, software and so on. The Checker, ImageBuilder, ImageInstaller and Local Resource Manager component access to these physic elements in mutual exclusion to avoid inconsistencies and because the tools that they use (such as apt*) cannot be accessed concurrently. Here is an example of the current structure of these storage elements:

```
/store/
  |-/kernels/
  |  |-linux-image-2.6.18-6-xen-amd64_2.6.18.dfsg.1-18etch1_amd64.deb
  |  |-linux-modules-2.6.18-6-xen-amd64_2.6.18.dfsg.1-18etch1_amd64.deb
  |
  |-/packages/
  |     |-/etch/
  |        |-fortune-mod_1%3a1.99.1-3_amd64.deb
  |
  |-/releases/
  |     |-/etch/
  |     |  |-base-deb-etch-amd64.tgz
  |     |  |-base-deb-etch-i386.tgz
  |     |
  |     |-/sarge/
  |        |-base-deb-sarge-amd64.tgz
  |
  |-/archives/
  |  |-/30173/
  |  |-image-30173.tar.gz
  |
  |-log
  |-error_log
```

The Store is intended to be a central repository for every provider's resource. Thus, it has to maintain a great variety of realeases, kernels, etc. We have introduced this cache in our solution because transfer data within a LAN is faster than downloading it from Internet. In the System evaluation chapter 8 the improvement caused by this cache is quantitatively evaluated. Although caching is a good practice, we have to take into account the size of the Store (because there is a

hardware resource constraint) as well as maintaining softwares packages up-to-date. Through the concluding chapter 11, we propose a solution to carry out these issues.

```
/pool/
  |-/kernels/
  |  |-linux-image-2.6.18-6-xen-amd64_2.6.18.dfsg.1-18etch1_amd64.deb
  |  |-linux-modules-2.6.18-6-xen-amd64_2.6.18.dfsg.1-18etch1_amd64.deb
  |
  |-/releases/
  |     |-base-deb-etch-amd64.tgz
  |     |-base-deb-sarge-amd64.tgz
  |
  |-/swap/
  |     |-swap.img
  |
  |-/aplic/
  |     |-/tycho/
  |     |  |-/lib/
  |     |  |-resourceFabrics
  |     |  |-resourceFabrics.jar
  |     |
  |     |-jdk5.tgz
  |
  |-/archives/
  |     |-/VM0/
  |        |-VM0.cfg
  |        |-vmlinuz-2.6.18-6-xen-amd64
  |        |-initrd.img-2.6.18-6-xen-amd64
  |        |disk-VM0
  |
  |-log
  |-error_log
```

Regarding the Pool element, it has a similar structure than the Store component, regardless it is supposed to be smaller because it is intended to be a cache for the resource itself. To maintain the Pool updated and small (the disk space is supposed to be used by the VMs, so we do not want to waste too much space) we have implemented a Least Recently Used (LRU) policy. We keep in

the cache the two least recently used kernels and releases. We have taken this decision because we assume that clients will use the same kernels and realeases (the newest ones), therefore producing a low miss rate. The improvement caused by the introduction of this cache is quantified in the System Evaluation chapter 8. This improvement is pretty smaller than the one got with the Store caching. Thus, a higher miss rate does not overhead our solution.

Data transfer

We have decided that the data stored in the Store will be shipped to the target resources' Pool using the SCP protocol with RSA based authentication. We have taken this decision because it facilitates the deployment phase. It is so because SCP works over SSH and installing and configuring a SSH server is as easy as installing one single package and leave it with the default configuration. In the System Evaluation chapter 8 we compare the performance of this protocol with the one obtained by the FTP protocol, showing a better performance in the case of SSH-based protocol. This way, it reasserts our decision of using the SCP.

VM characteristics

Finally, the VM created with these scripts will have the following characteristics:

- Its file system will be stored in a file without gaps.
- The software packages that the client had required will be installed properly, (i.e. configuration files under /etc/ and so on).
- Every VM will have Java 1.5 installed by default, This Java software corresponds with the jdk5.tgz file under *pool/aplic/* diretory.
- VMs will have installed and running a Java application (*pool/aplic/tycho/resourceFabrics.jar*) in charge of publishing the state of the resource and the reception of jobs to be executed on the VM (see section 7.3 for the job sending description). In order to start up and stop this software when the VM is boot up and shutdown respectively, we have built an script (*pool/aplic/tycho/resourceFabrics*) stored in */etc/init.d* directory (of the VM) and softlinked inside the suitable run levels directories.
- VMs will incorporate a SSH server. It is specified using the *MANDATORYPACKAGES* variable in the configuration file (see appendix A).

7.3 Economic Resource Manager implementation

With respect to the technologies used inside ERM component we have to outstand Tycho software [24] used to send jobs to the VMs. Tycho is *A Resource Discovery Framework and Messaging System for Distributed Applications* as they advertise in their web page. Thus, in this project we do not have to worry about how we send jobs to VMs. We just have installed the Java application mentioned at the end of the previous section within every VM from the RF side and from the side of ERM we interact with Tycho Java classes as detailed in previous chapter (see Design chapter 6).

7.4 Policy Manager implementation

The decision of which Rule Engine is meant to be used, is not part of the project. However, we decided to use JESS because it is available for free for academic research, it is highly interoperable with Java and it is used by other partners of the SORMA project.

For this prototype, we have only implemented one rule. This is used when a reservation conflict occurs. Given two numbers representing clients' preference, it returns the most preferential client between them:

```
(deffunction clientConflict (?c1 ?c2)
    (if (> ?c1 ?c2) then
        (return 1)
    else (if (< ?c1 ?c2) then
            (return 2)
         else
            (return 0)
         )
    )
)
```

This is just a very simple example of what we could do using a RE. Of course, if we would had done it in Java it would be easier to implement, but more sophisticated rules and functions are expected in the future. Besides, this way the rule could be smoothly changed. Creating an accurate ontology and adding facts to the base fact on runtime, converts PM into a high tool for adapting behaviour at execution time. However, this ontology is out of the scope for this prototype as well as SORMA's prototype due to its complex design.

Chapter 8

System evaluation

In this chapter, we present the experiments done to evaluate the performace of our system. The aim of these tests is to obtain results that will reassert our design decisions. Related to ERM and PM we evaluate the improvement of having Interval reservations with respect to Fixed ones and how the PM achieves provider's goals (section 8.1). And related to RF we show the improvements due to the introduction of Store and Pool caches and a comparison between data transfer protocols (see sections 8.2 and 8.3 respectively).

We use the *commons math* library 1.2 under the Apache Software License 2.0 to calculate the statistics presented in this chapter.

8.1 Policy Manager and Reservations

The aim of this section is twofold. On one hand, we will show that the design of the reservations helps to achieve an efficient use of the virtualized resources, in terms of increasing the number of planned reservations. And, on the other hand, we will demonstrate empirically that using the Policy Manager we are able to accomplish provider's objectives. In the case of this prototype, it means prioritizing reservations of the most preferred clients taking into account the current state of the system. To a complete description of the reservations and the Policy Manager rules refer to chapters 6 and 7.

Experimental setup

The experiments presented in this section are based on the following setup:

$$Number\ of\ experiment\ iterations\ =\ 20 \tag{8.1}$$

The number of test iterations is set to 20 times.

$$Number\ of\ reservation\ requests\ =\ 200\ reservations\ per\ timetable \tag{8.2}$$

The 200 reservations in equation 8.2 are the same for each timetable. It is so, to have comparable results.

$$1h\ <=\ reservation\ duration\ <=\ 10h \tag{8.3}$$

$$Timetable\ length\ =\ 168h \tag{8.4}$$

We will plan the reservations into a timetable of a week of duration (168h).

$$0h\ <=\ lowerbound\ <=\ 167h \tag{8.5}$$

The inequation above represents that the reservations can start at any time of the timetable length.

$$lowerbound\ +\ duration\ h\ <=\ upperbound\ <=\ 167\ +\ duration\ h \tag{8.6}$$

The reservations may finish at the end of timetable plus its duration. It is so, because a reservation of duration 10 can be planned to start at the hour 167 of the timetable so, it will finish at 167 + 10 h.

$$Timetable\ extralength\ =\ 167\ +\ 10\ h\ =\ 177h \tag{8.7}$$

We set the timetable length to a week long (168h), but as we want to avoid cutting reservations if their end exceeds timetable's length, we permit reservations to finish when they need. 10 is the maximum duration of a reservation and 167 is the latest a reservation can start.

$$Percentage\ of\ reservations\ of\ gold\ clients\ =\ 30\% \tag{8.8}$$

A 30% of the requested reservations will be done by gold clients.

We generate these values using an uniform distribution. Tests IV-VII demonstrates the validity of this setup by changing the range value of each parameter.

Fixed vs Interval reservations

First of all, we just had Fixed reservations, these are the reservations specified by means of a beginning and end date time for a VM booking. But this reservation representation is very poor because, its an innefficient time slot representation (e.g. imagine that we have two huge reservations that are overlapped only by a unit of time, we only can plan one of them into our VM timetable). That is the reason for which we had designed Interval reservations. These are the ones which are specified by means of the booking duration and a time interval where is feasible to plan them.

The improvement achieved by introducing Interval reservations is intuitive. In the worst case, an Interval reservation can be thought of as a Fixed reservation, when the upperbound of the interval is equal to the lowerbound plus the duration. Otherwise, we have the freedom to move the planned reservations, within their interval, to try to fit a new reservation. Therefore, having Interval reservations might increase the number of planned reservations with respect to having just Fixed Reservations. In the current prototype, we have implemented both of them and they can be mixed.

To test the improvement introduced by Interval reservations with respect to Fixed reservations, we try to schedule 200 Interval reservations on a timetable and the same 200 reservations converted into Fixed reservations (changing equation 8.6 for this: $upperbound = lowerbound + duration\ h$) on another timetable.

Table 8.1 shows the number of planned reservations in both cases, notice that with Interval reservations we gain a mean of 7.2 planned reservations.

	Mean	**Stdev**	**Min**	**Max**
# Reservations Fix	**34.1**	3.386	29	40
# Reservations Interval	**41.3**	4.231	35	53

Table 8.1: Fixed vs Interval reservations.

On average, we always earn several planned reservations but theoretically some worst cases exists, as shown in figure 8.1. On the left side of the figure, we show some requests of Fixed reservations and under the horizontal line we show how the timetable will look like after these requests. We

had to reject the second reservation because it overlaps with the first one already planned. On the right side, we show the same reservations but now with an interval of time where to plan them. As a result of this range, after planning the first reservation we can reschedule it, move it later until its ending interval. This way, we can accept the second reservation but we have to reject the third and forth ones. Thus, with Fixed reservations we have planned 3 reservations. Conversely with Interval reservations, we have planned just 2 reservations. This is a very particular case, hidden in the tests because of its low probability of happening. Nevertheless, we are still maximizing the number of planned reservations online, that is just considering the already planned reservations and the reservation we are about plan. Notice that after the second reservation, we have planned only one reservation, in the case of having Fixed reservations (left side of the figure) against the two planned reservations in the case of Interval reservations.

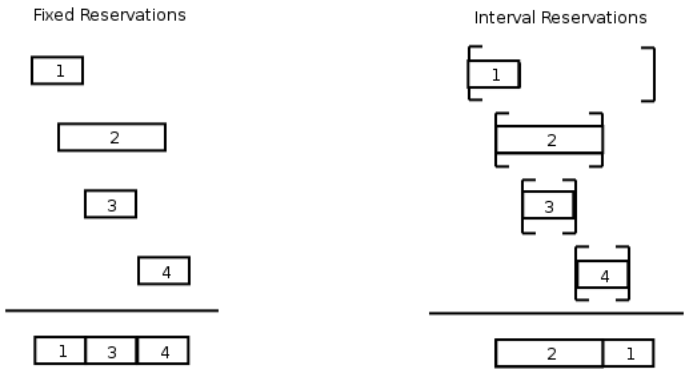


Figure 8.1: Interval improvement worst case.

Policy Manager improvement

The second improvement that we have introduced to achieve a great satisfaction of providers' goals is the use of Policy Manager rules to decide which reservation must to rest planned in the case of conflict between a reservation about to be planned and an already planned reservation. In our prototype, the resource provider wants to prioritize reservations of clients with a higher loyalty (preference). To demonstrate that the use of Policy Manager helps the provider to get its objectives we have designed tests I-VII.

Test I:

For this test, we use two timetables where to plan Interval reservations. We will try to plan the same reservations in each one. The difference between them is that in one of them, we have activated and follow the advises of the Policy Manager component and in the other we do not take

into account PM advises.

The results of the test are collected in table 8.2. We show the number of reservations planned (with and without PM) and then we breakdown them into reservations of standard clients and those of gold clients. The number of gold reservations has significatively increased with the activation of PM. We have pass from having about 12 gold reservations to 31. Moreover, we can observe that the number of reservations using PM has increased in 12.45 on average, that is a 29% higher than without using PM. This growth is due to the fact that when the timetable is quite plenty the PM only can decide to change one reservation for another in the case that the new one will be smaller than the planned one, otherwise the new one will not fit in the timetable. Thereby, we can plan more reservations because they are smaller

		Mean	**Stdev**	**Min**	**Max**
# Reservations	**without PM**	42.95	3.90	35	49
	with PM	55.40	4.50	46	64
# Standard clients	**without PM**	30.90	4.69	23	39
	with PM	23.7	5.57	14	34
# Gold clients	**without PM**	12.05	3.14	8	20
	with PM	31.70	4.19	26	39
Reservations growth		12.45	3.50	7	19
Gold clients growth		19.65	2.66	16	26

Reservations improvement	**1.29x**
Gold clients improvement	**2.63x**

Table 8.2: Policy Manager test I.

Both improvements, the reservations' growth and the gold reservations' growth are obtained in most of the scenarios, but there are some scenarios where no improvement exists. Figure 8.2 illustrates this case. On the left side, we show an scenario without using PM. There, when the second reservation request arrives we have to reject it because it overlaps with the first one. Conversely, on the right side, the second request is accepted and the first one is removed using PM. Therefore, we are maximizing the number of gold reservations at each time, just taking into account the current state of the timetable (after the arrival of the second reservation, the timetable on the left has 0 gold reservations while the timetable that uses PM has 1 gold reservation), although we are not maximazing this at the end of the experiment.

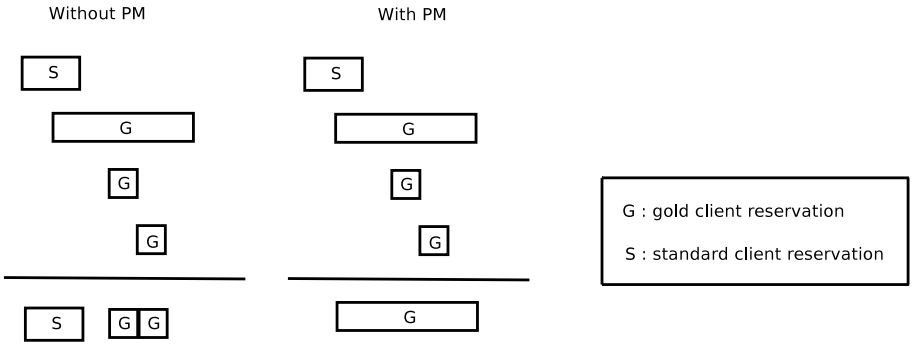


Figure 8.2: PM improvement worst case.

Test II:

This test will demonstrate that Policy Manager also works when we only have Fixed reservations, table 8.3 quantifies this utility. The setup for this test is the same as the previous section, but changing equation 8.6 for this:

$$upperbound = lowerbound + duration\ h$$

		Mean	Stdev	Min	Max
# Reservations	**without PM**	34.1	3.386	29	40
	with PM	34.55	2.781	29	40
# Standard clients	**without PM**	22.7	2.577	18	27
	with PM	12.65	2.834	6	18
# Gold clients	**without PM**	11.4	2.722	7	18
	with PM	21.9	2.426	18	29

Table 8.3: Policy Manager test II. Fixed reservations.

The results show that with PM we can plan 1.92 times more gold reservations than without using PM, so in this case PM also works adequately.

Test III:

This test will illustrate that client classification could be as sophisticated as wanted, having different levels of preferential clients. In this experiment, we have introduced the diamond clients, more preferred than the gold ones. For their setup, we change the percentage of gold clients' reservations (shown in equation 8.8) for these two percentages:

$$Percentage\ of\ reservations\ of\ gold\ clients\ =\ 28\%$$

$$Percentage\ of\ reservations\ of\ diamond\ clients\ =\ 2\%$$

Table 8.4 shows the results of this test, we can conclude that using PM we could achieve the provider's desire of prioritizing reservations of most preferred clients.

		Mean	Stdev	Min	Max
# Reservations	**without PM**	43.3	4.52	35	56
	with PM	55.75	4.22	47	63
# Standard clients	**without PM**	29.85	4.28	22	41
	with PM	24.15	3.27	20	32
# Gold clients	**without PM**	11.5	2.54	6	15
	with PM	26.45	3.24	22	32
# Diamond clients	**without PM**	1.95	1.36	0	5
	with PM	5.15	2.60	2	11
Reservation growth		12.45	4.84	4	20
Gold clients growth		14.95	3.52	8	20
Diamond clients growth		3.2	1.96	0	8

Reservations improvement	**1.29x**
Gold clients improvement	**2.3x**
Diamond clients improvement	**2.64x**

Table 8.4: Policy Manager test III. 3 Levels of clients' loyalty

Test IV:

In this test, we vary the number of generated reservations (equation 8.2) to observe the behaviour of our system in case of high contention. Figure 8.3 shows the results of increasing the number of requests (x axis). We conclude that the number of reservations and the number of gold clients reservations planned increase as the number of reservation's requests increase (the scheduler has more reservations variability to choose that ones that fit better in the timetable). It also shows that, without using PM, the number of planned reservations is independent of the number of requests.

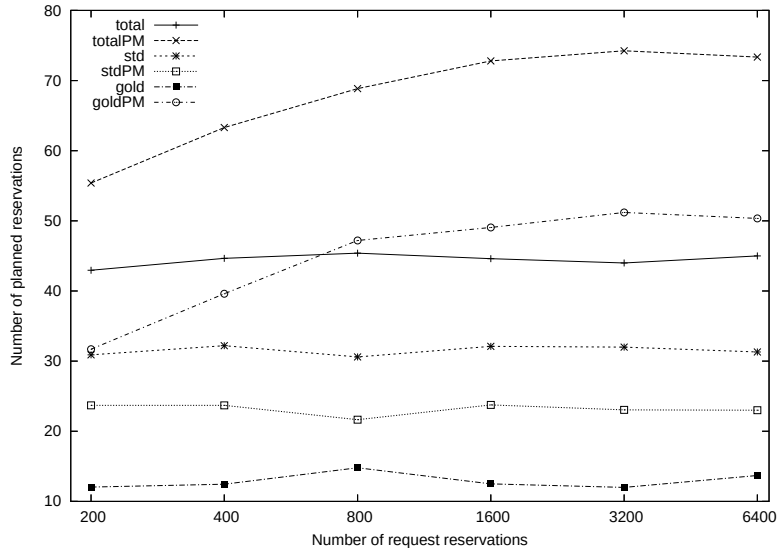


Figure 8.3: Test IV. Number of planned reservations vs requested ones.

Test V:

In this test, we vary another parameter, the percentage of gold reservations (equation 8.8) to observe the consequences of changing it. Figure 8.4 illustrates the results. We can observe that the amount of gold reservations increase as the amount of requests of gold reservations increase. This provokes a decrease in the number of standard clients as their percentage decrease. Therefore, we conclude that the amount of planned reservations is independent of their preference.

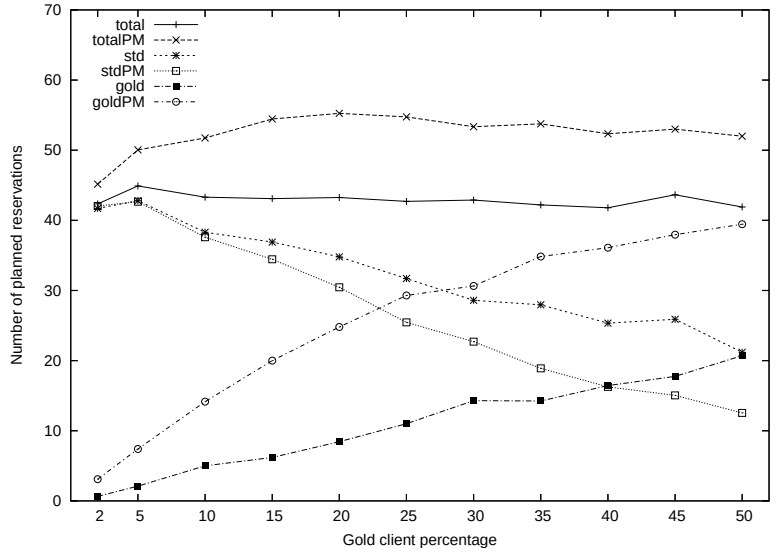
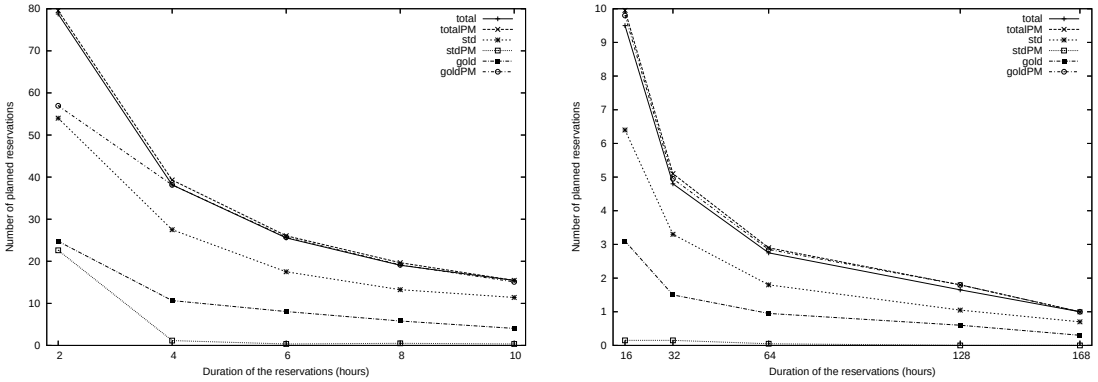


Figure 8.4: Test V. Number of planned reservations vs gold clients' percentage.

Test VI:

This test infers the reaction of our system as we vary the length of the reservations (equation 8.3). We plot the results in figure 8.5. In subfigure 8.5(a), we vary the duration length from 1 to 10 hours, that corresponds to our default setup (equation 8.3). In subfigure 8.5(b), we vary the duration from 16 to 168 hours (notice that our timetable is 168 h length plus an extra length quantified in 8.7, so with reservations of 85 h or more the timetable would contain two reservation at most). We conclude that as the reservation duration increase the number of planned reservations decrease, approximately each time we duplicate the length of the reservations, the number of planned reservations is divided by two.



(a) Short reservations (b) Long reservations

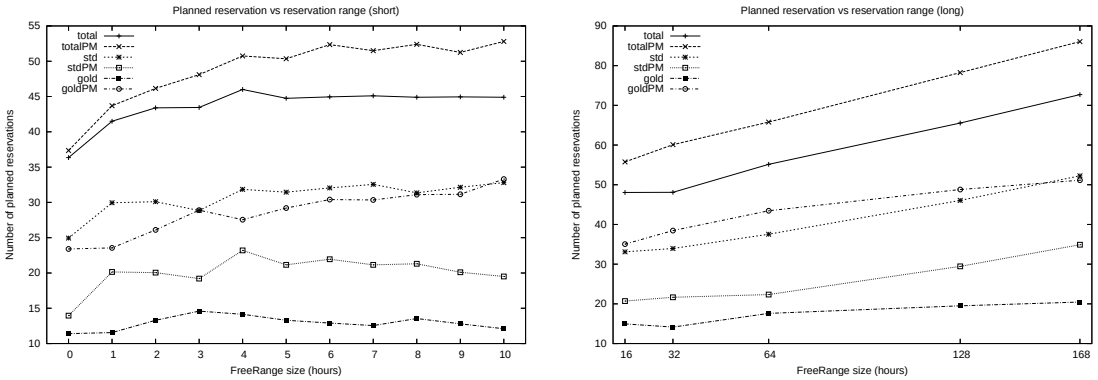
Figure 8.5: Test VI. Number of planned reservations vs reservations length.

Test VII:

In this last test, we characterize the performace of the system with respect to the length of its interval. We change equation 8.6 for this:

$$upperbound = lowerbound + duration + freeRange$$

Where *freeRange* is the difference between the interval length and the duration of the reservation. Figure 8.6 depicts the number of reservations planned as function of the freeRange variable. These results show that the bigger the freeRange, the bigger the amount of planned reservations.



(a) Small freeRange (b) Big freeRange

Figure 8.6: Test VII. Number of planned reservations vs freeRange length.

8.2 Create Machine improvement

In this section, we show the improvement due to the introduction of Pool and Store caches in the VM creation process. To do this test, we deploy our system in a cluster, as shown in figure 8.7. We use one machine (pcaudet) to execute EERM, and a machine with Xen hypervisor (pccasals) to execute the LVM component. Furthermore, we use a VM to host the Resource Coordinator component. We use a VM instead of a physical machine to host RC because there were not any other machine in the cluster with the same processor architecture of pccasals. That suppose a problem because RC is in charge of downloading the software that will be used in the VMs using apt-get tool and although apt-get is supposed to be prepared to download packages from any architecture available the truth is that it does not do so.

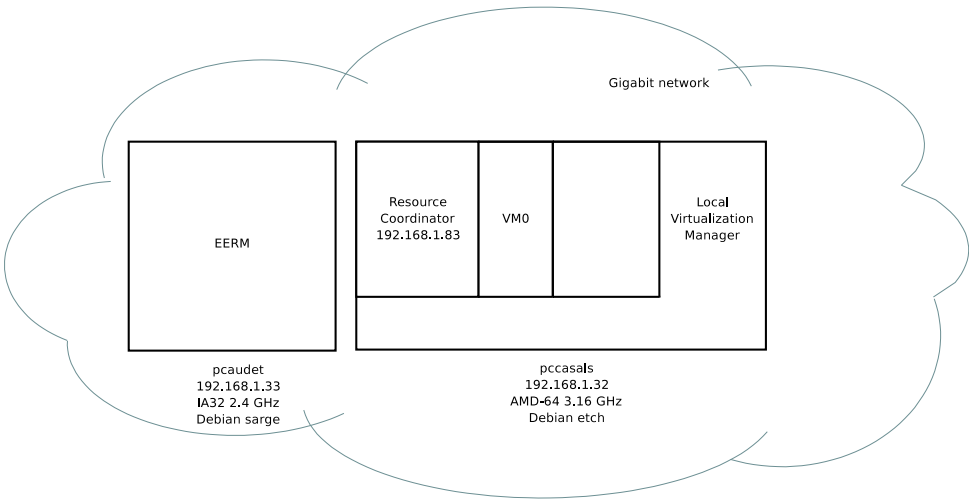


Figure 8.7: Test scenario.

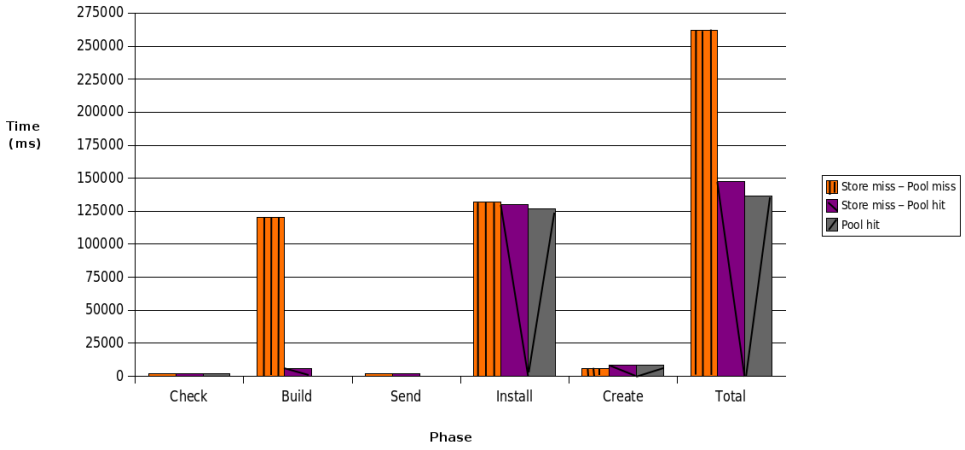


Figure 8.8: Create Machine Improvement.

We consider three different scenarios: the first one, when the software requirements for creating the VM cannot be satisfied by neither the Pool cache nor the Store cache; the second one is when those requirements cannot be satisfied by the Pool cache but can be satisfied by the Store cache; and the third case is when the requirements can be satisfied by the Pool cache.

Figure 8.8 shows in which phase of the process of creating a VM the time is spent in each scenario. The *install* phase (installing release packages) takes roughly the same time in any case. In chapter 11 Conclusions and Future Work there are some suggestions to reduce this time. In case of a miss in both caches, we expend a great amount of time (120493 ms) downloading the requirements from Internet (*build* phase) and packaging them to be sent in the *send* phase. In the case of a miss in the Pool and a hit in the Store, we can observe that the *build* phase time has been reduced to 5868 ms, which is more than 20 times smaller than in the previous case. This time is spent in preparing what is going to be sent to the Pool. And in the last case, if we hit in the Pool, we can skip the *build* and *send* phases, although comparing with the previous case it just supposes 7534ms of reduction.

Therefore, we conclude that the introduction of the Store cache has a great impact in the system performance as we reduce almost half of the time for creating a machine. The introduction of the Pool is not so noteworthy because it is intended to avoid the transfer time from the Store to the Pool. Although this time is negligible in our LAN, it could be more significant in slower LANs. Besides, the overhead introduced by this design is minimum if we compare it with an oblivious solution in which there is no caching. Thus, the cost of creating a VM in our solution when the caches are empty is similar to creating a VM in the oblivious solution, but for the *check* and *send* phases (which are the smallest phases with regard to time). Once the Store and Pool caches are initialized, the results are greatly improved compared with the oblivious solution even considering the overhead introduced by the *check* and *send* phases.

8.3 Transfer protocols comparison

In this section we compare two transfer protocols to decide which is the most suitable for our system. For this test, we have installed in one cluster machine *wu-ftpd* Debian package to act as a FTP server and we use already installed *open-ssh* package as a SSH server. We also install package *expect* which is used to automate interactive applications to do the FTP test.

Our test consists of computing the time spent in sending packages from one cluster node to another node (these two nodes are depicted in figure 8.7) in slots of 10 repetitions for each protocol and data package (connection establishment time + data transfer time). We use RSA authentication in the case of SCP test. Table 8.3 shows the results. We can observe that SCP is faster than FTP if the size of the data is small, less than 100MB. Contrary, the FTP protocol is faster for bigger data . We deduce that the time to establish a connection is bigger in the case of FTP (as it have to establish two TCP connections, one for control and one for data), but it transfers data faster than SCP. In our system we have implemented SCP as the data that we use to transfer is about 50MB (base release + kernel).

	FTP	**SCP**
Base system (44MB)	2.7371 s	1.7151 s
Kernel (4MB)	2.1857 s	0.4721 s
200 MB	6.0678 s	6.651 s
100 MB	3.5164 s	3.7144 s
50 MB	3.1258 s	1.9563 s

Table 8.5: Data transfer protocols comparison.

Chapter 9

Related work

This chapter present the related work for this project: a set of projects, that are currently being developed and have been used as ideas source for the design of the project. In the following sections, we introduce each system and we compare the main aspects that they have in common with our project and what is useful for our design.

9.1 The XenoServer Open Platform

The XenoServer Open Platform is a public infrastructure for global-scale service deployment, where XenoServers are spread around the world and available for any member of the public. This project covers so many topics: resource management, resource discovery, authentication, privacy, charging, billing, payment and auditing. It has been developed in the Computer laboratory of the University of Cambridge.

In this project, clients can submit tasks to a VM hosted in a XenoServer, all the transactions are done through XenoCorp, a trusted third party. The virtualization software used in this project is Xen.

This platform provides clients with several immutable template images for operating system kernels and file-systems. They store these images locally in the hosts where the virtual machines will be created, a.k.a the XenoServers (but there is not guarantee that all the images will be persistently cached). Clients define *a tailored image*, in terms of modifications to these templates, called an *overlay*. Although the templates are cached locally at the XenoServers, client overlays are remote stored in the XenoStore (a trusted distributed storage service) and are accessed trough a NFS server (see figure 9.1). This approach reduces the amount of data shipped by the network dur-

ing deployment. Since the overlay is remote it may be shared between multiple virtual machines running on a set of XenoServers.

We have decided to install the software packages as proposed by Unix recommendations. Configuration files in /etc, and so on. This is opposed to XenoServer's design as they have its software installations under a unique directory. Their solution has a potential drawback when accessing remote software as it might be very slow. They also propose Pasta [25], a distributed storage system based on a DHT, as an alternative to XenoStore, but does not fit our goals (having a DHT has no sense in a LAN context).

See [10] for an overview of their project and [26] for the explanation of their proposal.

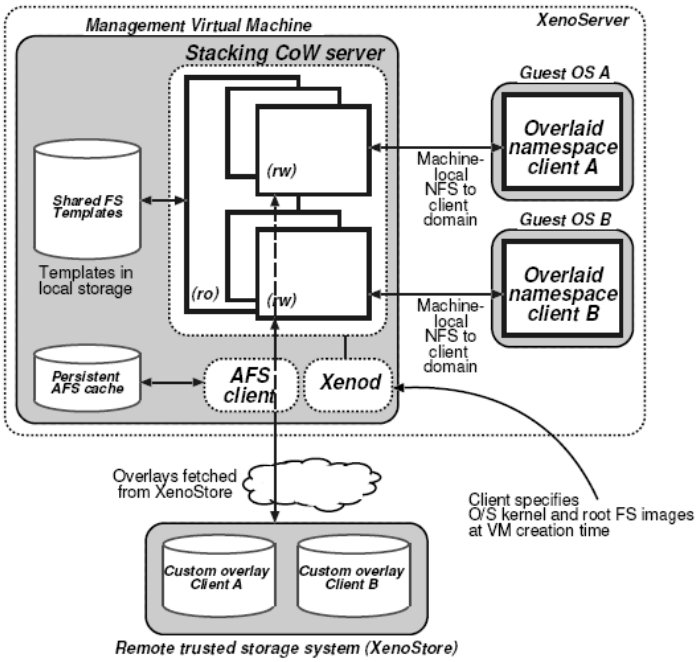


Figure 9.1: The XenoServer platform. Deploying a Guest OS in the XenoStore model. Figure extracted from [26].

9.2 BREIN project

Brein European project[18] (**B**usiness objective driven **RE**liable and **I**ntelligent grids for real busi**N**ess) aims to provide an infrastructure that foster the collaborations among grid's Virtual Organizations. To achieve this objective they use the Grid, semantic web, multi-agents and virtualization technologies.

Inside this project, the VtM component (Virtualization Manager) is in charge of creating and efficiently-managing VMs. Its current implementation of the disk image creation for a VM is different from our design in these aspects:

- They do not have a central repository for caching releases and kernels, they only cache one release locally in the servers that will run the VMs.
- The VMs that can be created through VtM, are not customized. They will run the same kernel as the host and will use the default release.
- The applications that will be installed in the VMs are installed in another disk partition mounted under /aplic. Furthermore, all the VMs will have the same software at creation time as there is no system actor in charge of describing VMs software.

The VtM component, concretely its Resource Manager subcomponent, has been used by this project as an starting point for our implementation.

9.3 In-VIGO project

The In-VIGO project [27] (**I**n **V**irtual **I**nformation **G**rid **O**rganizations) is another approach to Grid-computing which aims to decouple user environments from Grid resources.

In-VIGO provides users with tools to automate the creation of application services by describing how its services work. It also provides each user with a persistent private virtual workspace for launching and developing applications and using and managing private data. This is achieved by means of virtualizing all resources (see figure 9.2): machines, networks, applications and data.

They have designed VMPlant [28], a Grid service that automates the creation of VMs. Once a machine is created and configured to meet application needs, it can be copied and instantiated to provide homogeneous execution environments scattered across Grid resources. For automate this process, they maintain a cache of VMs' images. The design is as following: a client can express his requirements by means of a direct acyclic graph (DAG) specification of a virtual machine. Nodes in the DAG configuration may be associated with actions to be performed within a virtual machine's guest. The DAG enforces a partial order between the actions (e.g, first install red hat linux 8.0, secondly install Web File Manager, start file manager, etc.). This DAG aids the virtual machine created process by supporting partial matches of cached VM images. The process of cloning and configuring is thus based on: firstly, the copy of a machine's state from its original

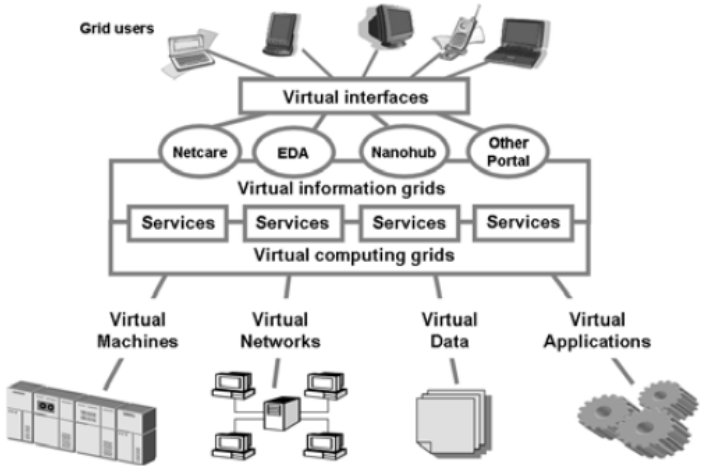


Figure 9.2: High-level view of the In-VIGO project approach. Figure extracted from [27].

image to the cloned image. Thereafter, the instantiation of the machine and finally, the execution of configuration actions.

The In-VIGO proposal differs in our requirements in the sense that they store virtual machines resumed and saved and we only want cache disk images. Furthermore, they consider quite homogeneous images. Otherwise, the cache space required might be unmanageable. Conversely, we have preferred to have a fine-grain control over the software installed in our VMs.

9.4 SoftUDC

SoftUDC [29] is a software-based utility data center that virtualizes server, network, and storage resources. It is being developed in the HP laboratories.

Their approach abstracts a bundle of hosts as a single node which may contain several VMs on it. It uses Xen as the virtualization software, although it incorporates some modifications for managing services and controlling the I/O network traffic. It uses SmartFrog [30] to automate the process of creating and deploying VMs and to control online maintenance operations.

SmartFrog is a framework with a language for describing and activating services (see figure 9.3). It is implemented in Java by the HP laboratories as well and released under LGPL license.

This proposal do not have a central repository like ours to cache packages and base systems. They install application packages and instantiate requested applications and services using a daemon running in Domain 0 of each host.

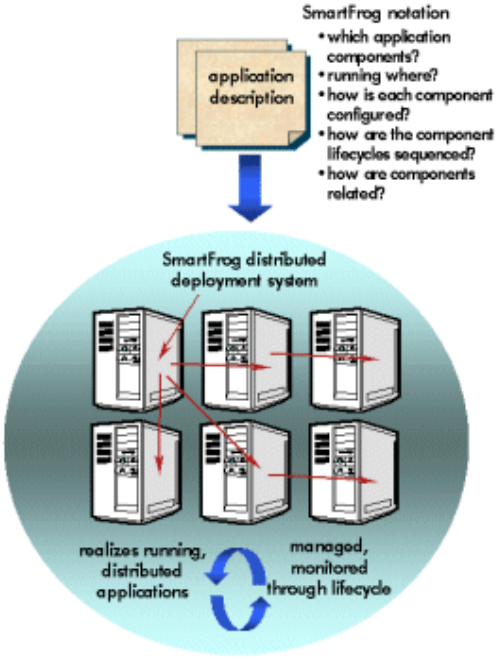


Figure 9.3: SmartFrog framework. Figure extracted from [30].

9.5 Globus Virtual Workspace

The Globus Virtual Workspace [31] is a project from the Globus Alliance.

They define a workspace as *an execution environment that can be made dynamically available to authorized clients by using well-defined protocols*. These workspaces are implemented as VMs for the current infrastructure.

This project provides users with functions based on WSRF (Web Service Resource Framework) protocols to manage VM (as depicted in figure 9.4). Therefore, clients can deploy their workspaces, manage them and control the resources allocated to them. We use Web Service instead of WSRF because there is no need of maintaining any state.

In their project's web page [31], they offer several workspaces already prepared to be deployed. However, they do not provide a tool to automate the process of creating virtual workspaces. Contrary, we have focus our project in the automation of building customized workspaces (i.e. virtual machines).

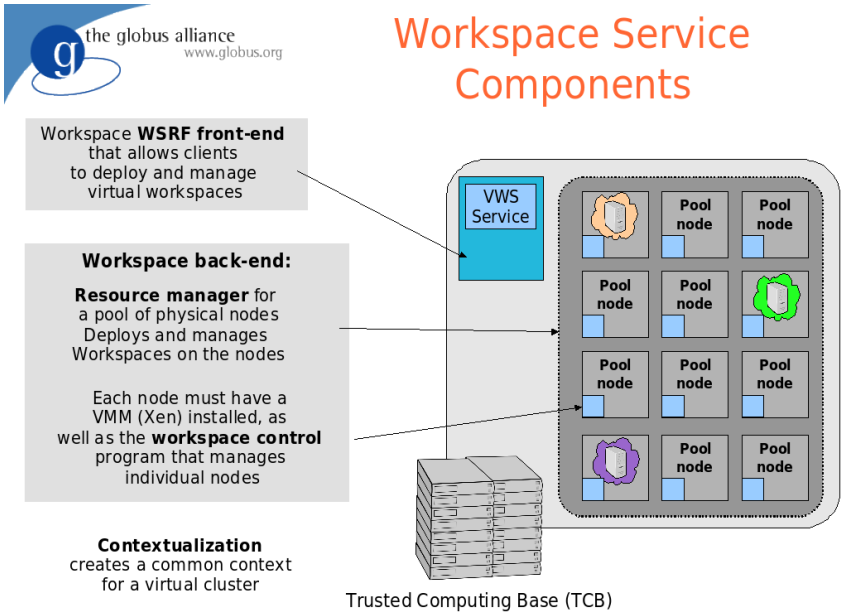


Figure 9.4: Globus Virtual Workspace. Figure extracted from [32].

9.6 SODA

SODA [33] is a **S**ervice-**O**n-**D**emand **A**rchitecture which focus on the hosting of application services.

SODA uses User-Mode-Linux (UML) as a virtualization technique. UML runs without requiring any modification of the host user space. Processes within an UML (guest OS) can be executed like on a real Linux machine. Conversely, our proposal requires that guest OSs have been ported to Xen.

To create services automatically, SODA downloads the service from the Application Service Provider (location specified by the customer). These services must be packaged using RPM an organized into a file system with a single root.

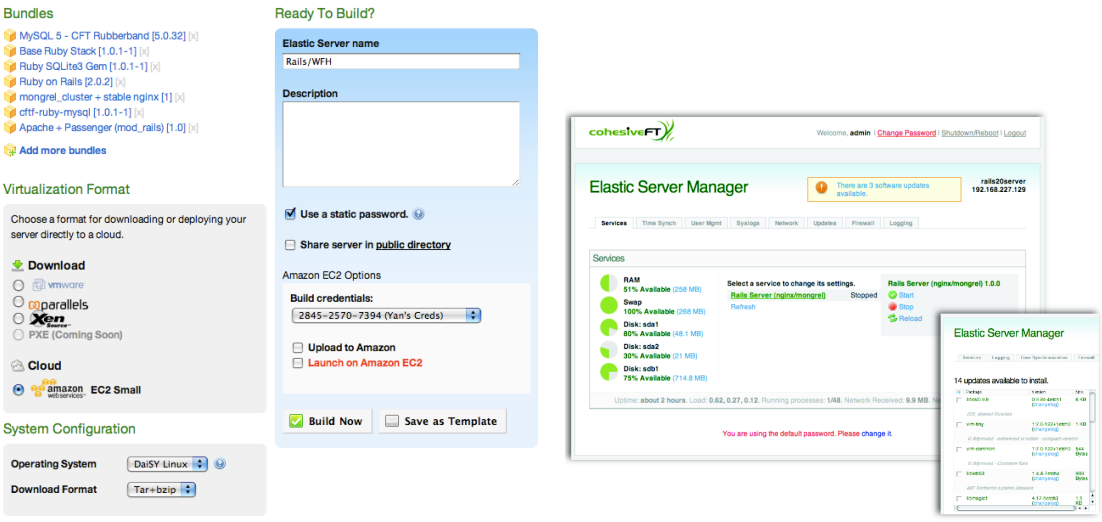
SODA is a quite old project and it uses the old version of UML that does not provide neither CPU nor bandwith isolation. Therefore, they have to implement as part of their project some enhancements inside the host OS in order to isolate these resources.

9.7 Amazon Elastic Computing Cloud

Amazon Elastic Compute Cloud [34] (Amazon EC2) is a commercial product that offers compute capacity in the Amazon's Cloud. Cloud computing (promoted by Amazon) differs from Grid computing. In the first one, the customers buy an infrastructure (a VM) where they can deploy or execute their own productivity applications. However, in a Grid, the customer submit jobs and gather results. Nevertheless, it is part of our related work as in both computational paradigms we might use virtualization.

Amazon EC2 offers a web service interface for users to create Xen VMs. Clients can only decide among three types of available VMs (small, large and extra large) as Amazon does not offer a more fine grain customization process for creating VMs. Furthermore, these VMs only differ in the amount of resources allocated to them and VMs only support their own Xen-enabled Linux kernel. This homogeneity in the VMs offered by Amazon differs from our system requirements. Anyway, Amazon EC2 enjoys a great popularity, but we do not know more details about its implementation as it is a private product.

9.8 Elastic Server



(a) Elastic Server. (b) Elastic Server Manager.

Figure 9.5: Elastic Server screenshots.

Elastic Server [35] offers a way to create VMs and to manage them independently of the cloud where they will be deployed. Its main objective is the interoperability among potentially clouds infrastructures.

Figure 9.5(a) depicts how a client can build a VM. He can specify which software has to be installed on the VM (coarse-grain customization, bundles in the figure), the virtualization format (Xen, Vmware, etc.), the system configuration (memory MB, hard drive MB, OS, etc.) and in which cloud the VM must be deployed (nowadays, only Amazon EC2 exists).

Once a VM is deployed in a cloud, the Elastic Server Manager (depicted in figure 9.5(b)) allows clients to manage its VMs and enables dynamic system reconfiguration.

Chapter 10

Project plan and economic evaluation

This chapter presents the project development plan carried out during the last months. Besides, we show the project costs including all the costs related to the project development: human salaries, software required licenses and hardware prices.

10.1 Plan and human cost

Through the Introduction chapter 1, we have described the project tasks and showed the initial plan, a tentative plan done during the first week of February of the current year that looked ahead what was going to be done. Now, we present the development plan post mortem, that is, what we have done. This final plan is a simplified plan as we have considered a full-time working day (8 hours per day). That is a simplification in the sense of that the real working day has been variable, but on average it has been around 40 hours per week. Besides, we do not have considered Easter week holidays, an other holiday days, but those should be considered in a real plan (e.g. in an enterprise context).

Figure 10.1 shows how the tasks have been scheduled along the time. Here we include two tasks that do not appear in the initial plan because they are previous to the initial plan task. These are: *Background and Related Work* and *Requirements specification and initial plan*. This figure also shows the tasks costs. Although the project has been developed by a single person, we have divided each task in subtasks to assign them to an IT stuff role. Therefore, we can provided a more realistic project cost. The salaries for role that we have considered are shown in table 10.1.

Comparing this plan with the initial one presented in the introductory chapter (see chapter 1) we can observe that some tasks were underestimated, which led to the procrastination of several tasks,

Role	Alias	€ h
Project Manager	PM	58
System Analyst	SA	45
Programmer	P	28

Table 10.1: Costs by roles

WBS	Name	Start	Finish	Duration	Cost	Assigned to
1	Background & Related Work	Jan 28	Feb 5	6d 2h	2,900	PM
2	Requirements specification & initial plan	Feb 5	Feb 8	3d 6h	1,740	PM
3	Progress report	Mar 24	Mar 24	6h	348	PM
4	**Software**	**Feb 11**	**May 9**	**65d**	**17,875**	
4.1	**Image Respository**	**Feb 11**	**Apr 4**	**40d**	**9,980**	
4.1.1	Specification & design	Feb 11	Feb 20	7d 4h	2,700	SA
4.1.2	Implementation & test	Feb 20	Apr 4	32d 4h	7,280	P
4.2	**Resource Fabrics**	**Apr 7**	**Apr 16**	**7d 4h**	**1,905**	
4.2.1	Specification & design	Apr 7	Apr 7	5h	225	SA
4.2.2	Implementation & test	Apr 7	Apr 16	7d 4h	1,680	P
4.3	**Policy Manager**	**Apr 16**	**Apr 17**	**1d 2h**	**550**	
4.3.1	Specification & design	Apr 16	Apr 17	6h	270	SA
4.3.2	Implementation & test	Apr 16	Apr 17	1d 2h	280	P
4.4	**Economic Resource Management**	**Apr 17**	**Apr 29**	**7d 4h**	**3,480**	
4.4.1	Specification & design	Apr 17	Apr 24	5d	1,800	SA
4.4.2	Implementation & test	Apr 17	Apr 29	7d 4h	1,680	P
4.5	Prototyping	Apr 29	May 1	2d 4h	560	P
4.6	Testing	May 1	May 9	6d 2h	1,400	P
5	Final report	May 12	Jun 12	23d 6h	8,645	P, PM, SA
6	Presentation	Jun 19	Jun 25	5d	2,320	PM

Figure 10.1: Final plan.

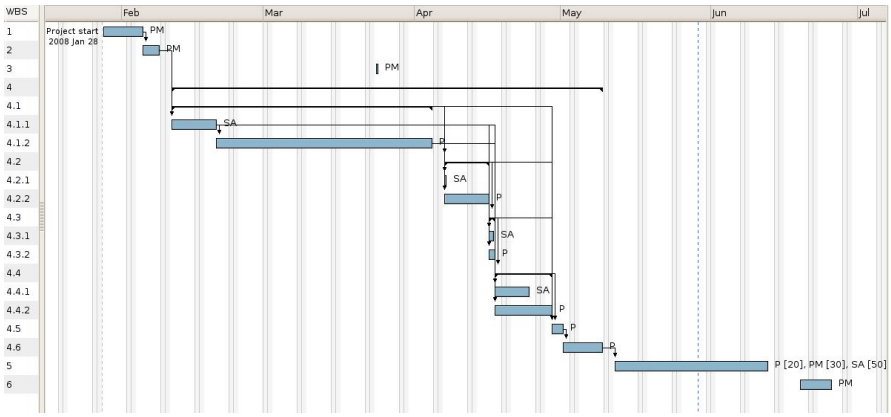


Figure 10.2: Final gantt diagram.

but never in excess. Contrary, the development of Policy Manager that was estimated in 24 hours, was reduced to 16 hours. To conclude, the project had a duration of 5 months (887 hours) and the cost related to human salaries was 35,788 €. Thus, its duration has exceed the initial one in 2 weeks (80 hours).

10.2 Software cost

Regarding the software used in this project we have to mention:

Linux: The OS Linux and its tools debootstrap, etc, are under GNU-GPL License, thus they not involve any monetary cost.

Tomcat: The application server is available under Apache License, i.e. with no cost.

Axis: This toolkit is also distributed under Apache License.

JESS: Jess is provided with no cost for academic research projects.

Xen: Xen is an open-source virtualization software under GNU-GPL License.

Java: Java is distributed with no cost.

Libraries: The libraries used during the project were under a free License as GNU-GPL or APACHE License.

Therefore, the software used in this project does not increase its cost.

10.3 Hardware cost

The software developed as part of this project and this report have completely been done in a single laptop, *a Samsung R40, Intel Core 2 Duo at 1.66GHz and 1 GB of Memory*, which costs 1,200 euros.Besides, the prototype was tested in the eDragon research group's cluster with no additional cost to charge on this project. However, in a real environment we should take this cost into account.

Chapter 11

Conclusions and Future Work

We end this report presenting our conclusions. Firstly, explaining several considerations about the system that we have built and secondly, we suggest some future trends to keep on working on the main topic of this project: the resource management. To close this chapter, I express some personal comments.

11.1 Project outcomes

We have presented a Policy-Driven Resource Management system to help providers to achieve their objectives as well as render them with a mechanism to create virtual customized execution environments on-demand.

Our final results derived from the initially proposed objectives includes:

- Construct a system for the customized creation of VMs on demand. This includes a hierarchy of caches to reduce this creation time.

- Design and implement a reservation mechanism that maximize the utilization of the VMs. Therefore, it maximizes provider's satisfaction as its resources are potentially fully utilized and maximize client's satisfaction by means of accommodating its reservations for VMs in advance.

- Build the infrastructure of Policy Manager for interpreting provider's rules and enabling EERM components to query it for recommendations to take decisions.

This system has been prototyped and tested proving that it works properly helping providers to accomplish their objectives.

We conclude that we have achieved with the objectives established for this project.

11.2 Future work

11.2.1 Economic Resource Registry

The Resource Registry introduced in the architecture chapter 5 within RF element is the component in charge of selecting a resource each time that creating a machine is required.

The current implementation of this registry is not taking into account the utilization of each resource. It just takes care about the required architecture for the new machine. Besides, the allocation of the VM on a resource is done randomly.

An interesting issue to work around in the future could be, how to apply economic models to resource registry to improve its behaviour. That means, how to avoid the overload in a resource whilst other resources are underutilized in terms of the number of virtual machines over each resource and their utilization.

11.2.2 Migration of VMs

The SORMA prototype will include the capability of migrating VMs from one host to another with the intention of managing these resources more efficiently (e.g. suppose that a VM on a host demands more resources (memory, disk, etc.) and that host cannot allocate them to the VM, but in the bundle exists a host that can offer them. Thus, we could migrate this VM to that host.).

This migration can be done by pausing and saving the VM, transfer it through the LAN to the new host and resuming it there. This is the easiest way of migrating VMs. However, any server or any process running on this VM must be paused. Therefore, it will be unavailable for a period of time and in so many cases this will not be allowed. Alternatively, live migration permits migrating VMs just stopping them about 300ms [36].

In order to reach an easy live migration of VMs among hosts, some changes should be done with respect to the current implementation. On the one hand, we will need an storage system shared among VMs as Xen's virtualization do not support disk migration. We can study the possibility of having locally scratch zones to write partial results which can be synchronized with the remote

disk periodically. Alternatively, we can use that shared storage only for customers' data and we can have cached in the Store and in the Pool read only images with several operating systems.

11.2.3 Reservations

In this section, we discuss about several issues related to the reservation scheduling process. First, we enumerate several ways of scheduling reservations. Secondly, we suggest how to improve the Policy Manager rules for resolving conflicts among reservations in a way that the provider's revenue and the client's and provider's satisfaction will be improved.

On the one hand, when trying to plan a reservation, we first try to schedule it in an empty timetable's gap. In the current implementation of both Fixed and Interval reservations, we plan a reservation in the first empty gap, a.k.a. *first fit*. Maybe, that is not the best choice because it might lead to the timetable's fragmentation. The same problem occurs in other areas, e.g. in the heap space of Java or in the Linux memory management to supply different sizes of contiguous memory, etc. Some solutions might be:

best fit : schedule the new reservation in the smallest gap that is bigger or equal to the reservation duration. However, this way we might have a lot of fragmentation (lots of tiny gaps).

worst fit : plan the new reservation in the biggest hole. This way we will have gaps as big as possible, but we will have a shortage of big gaps.

We have implemented the *first fit* policy because we are dealing with a timetable (as the time goes, it is not advisable to have gaps at the beginning of the timetable). Besides, the complexity of this solution is linear in the worst case (with respect to the number of already planned reservations). Conversely, in *best fit* as well as in *worst fit* cases the cost is linear, if we have not no more extra data structures (e.g. the Linux kernel uses the Buddy System Algorithm to allocate contiguous memory. It maintains lists of 1 contiguous block, 2, 4, 8,...1024).

On the other hand, we can introduce request reservation prediction per client. That is saving statistics per client of when they use to request a reservation, its duration, etc. Thereby, we can *pre-reserve* VMs. It will be useful if we have to pay a fine to a client for cancelling his reservation. For instance, if we know that our most preferential client use to request a machine reservation each Wednesday for the next Monday, we could pre-reserve each Monday (block reservations of other clients on that day) until that preferred client performs its reservation request or until Thursday morning, whatever comes first. Thereby, we potentially avoid cancelling reservations as the request of that preferential client does not overlap with any planned reservation.

Finally, when we are trying to plan a reservation and it overlaps with a planned reservation, we call Policy Manager to resolve that conflict. It sorts out the situation by choosing the most preferential reservation. However, this policy is extremely simple because it does not concern about monetary factors. A more accurate rule should consider at least: the reservations' duration, the price per hour, the client's preference and the cost of breaking the SLA of the planned reservation in case it has to unplan it. Thereby, if we provide PM with all this information, the provider might be able to elaborate more sophisticated policies (e.g. suppose that the provider wants to prioritize large reservation with high revenue. Then, he should define a rule taking into account the reservations' duration, and the price per hour).

11.2.4 Store maintenance

This section suggests a solution for maintaining up to date and size-limited the Store. This is the repository that caches kernels, releases and software packages to be deployed to resources.

The size of kernel and releases directories is quite limited. We can estimate it by multiplying the number of different processor architectures a provider has in its bundle by the number of available releases/Xen kernels (which is small). The problem arise when dealing with software packages, because the default behaviour of the Store is to search if it had the required package in its cache without consulting it through an official repository.

To maintain the Store up-to-date, we propose to create a background process that performs an update (to download the most up-to-date package list), scans the cached packages and removes the older versions. New packages will be downloaded on demand in a lazy manner. This process can be executed periodically using the *cron* application (to automate its execution).

11.3 Personal remarks

I also have seen accomplished my personal goals through the realization of this project.

The work done in this PFC has supposed a personal challenge to overcome and a good chance to know about some research projects that are currently being partially done in the Barcelona Supercomputing Center (BSC) such as SORMA [1] and BREIN [18]. Moreover, I have learnt about some topics of my interest such as Operating Systems, Virtualization and Resource Management.

I have enjoyed collaborating with the Computer Architecture Department (DAC) within the eDragon research group at the Technical University of Catalonia (UPC).

Appendix A

Deployment

This appendix is meant to be a quick guide to deploy and install our system. Throughout it, we explain where to place each component and how to install and configure them to work properly. We assume that our deployment is done over a Debian Linux OS.

A.1 Components placement

Figure A.1 shows a possible scenario where to deploy our prototype. We need one machine for the EERM components (ERM i PM), one for the RC component and several machines where to deploy LVM component. We call each server with the name of the component that must be installed on it prefixed by *pc_* (e.g. the computer where we place the RC component will be called pc_RC for the rest of the manual).

Notice that this deployment represents the most distributed configuration. However, a provider can put together all these components in the same server machine.

A.2 Software

We require to install and/or configure some extra software for a proper setup:

Tomcat : We use Apache Tomcat version 6.0.14 as the application server. We copy our services (.aar files) into *$TOMCAT_HOME/webapps/axis2/WEB-INF/services* and configure Tomcat to listen on port 8080. We have to install Tomcat in pc_RC and in pc_LVM.

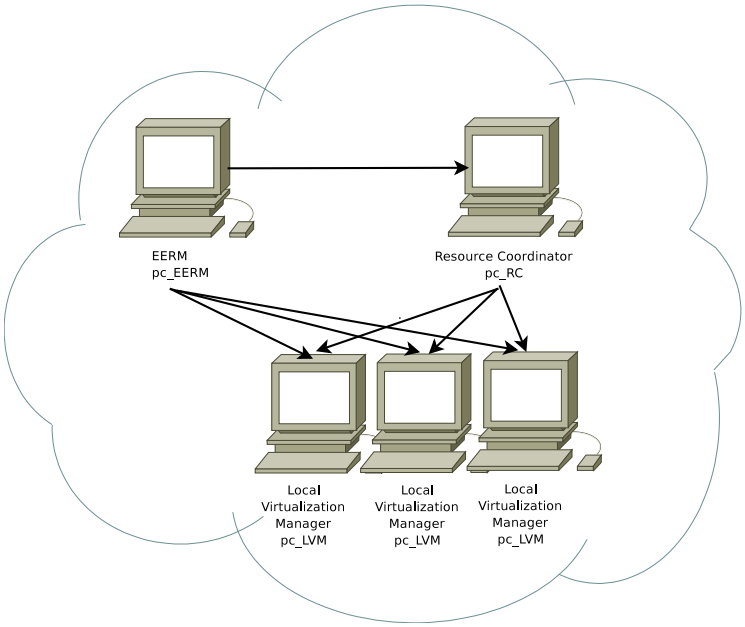


Figure A.1: Deployment scenario example.

JESS : Jess is the Rule Engine. It is necessary to copy *jess.jar* into *$JAVA_HOME/jre/lib/ext/* directory or set this jar into the *CLASSPATH* variable.

debootstrap : It must be installed in pc_RC and in pc_LVM (it can be installed by typing in a terminal as root: *apt-get install debootstrap*).

SSH : The system needs a SSH client in pc_RC and a SSH server in pc_LVM.

apt : Apt use to be already installed in Debian-based systems. Nevertheless, it is necessary to configure it. Modify the file */etc/apt/sources.list* from the pc_RC to point to the required repositories for the different architectures of the pc_LVM.

A.3 Configuration files

In this section, we explain the configuration files by means of complete examples. These configuration files must be stored on a directory named PDRM placed in /etc.

A.3.1 Configuration properties

The file *config.properties* is used to parametrize those components implemented in Java. This is a set of properties:

```
RESOURCES /etc/PDRM/resources.xml
```

This property points to where the file that describes the bundle of physical resources is placed. This property must be set on pc_RC

```
BUILDER /usr/share/PDRM/ImageBuilder.sh
CHECKER /usr/share/PDRM/Checker.sh
INSTALLER /usr/share/PDRM/ImageInstaller.sh
VTM /usr/share/PDRM/ResourceManager.sh
```

Those properties must point to where the scripts elements are placed. Builder property must to be set on pc_RC the rest in pc_LVM.

```
TIMEOUT 1000000000
```

This property specifies the timeout of a web service client. If no timeout is specified a default value is set. This property must be set on pc_EERM and pc_RC.

```
RESOURCOORDINATOR 192.168.1.83:8080
```

This is the Location of the Resource Coordinator services. This must be set on pc_EERM.

```
RULEENGINE eu.sormaproject.eerm.policymanager.JessRuleEngine
```

This property determines which RE the PM should use. This must be set on pc_EERM.

```
POLICYDIR /usr/share/PDRM/policy/
```

This is used to specify the directory where the policies that the RE should use are placed. This must be set on pc_EERM.

A.3.2 Advertising resources

The file *resources.xml* is used by the provider to publish its resources. This must be placed on the pc_RC. The provider must specify for each resource, its address, port where to find the services, processor architecture of the resource and a host user. It can also specify a protocol for the data transfer and a password if it is required for the transfer protocol.

```
<provider-resources>
    <resource>
        <url>192.168.3.2</url>
        <port>8080</port>
        <arch>i386</arch>
        <user>greig</user>
        <protocol>scp</protocol>
    </resource>
    <resource>
        <url>192.168.3.3</url>
        <port>8080</port>
        <arch>amd64</arch>
        <user>greig</user>
        <password>password</password>
        <protocol>ftp</protocol>
    </resource>
</provider-resources>
```

A.3.3 Store Environment

This is the configuration file for the Store component (the file must be named: *storeEnv.cfg* and placed in pc_RC):

```
BASEDIR=/usr/share/PDRM   #base directory, where the scripts are placed

#Store organization

STORE=/aplic/greig/store  #store location
DRELEASES=$STORE/releases #releases directory
DKERNELS=$STORE/kernels   #kernels directory
DPACKAGES=$STORE/packages #archives directory
ARCHIVES=$STORE/archives  #archives directory
LOG=$STORE/log            #log file
ERRORLOG=$STORE/error_log #error log file

PACKAGEFORMAT=deb         #default package format
RELEASE=etch              #default release
```

```
dKERNEL=2.6.18            #default kernel version
MANDATORYPACKAGES="ssh"   #not basic packages which will be included in
                          #every base images
MIRROR=http://ftp.rediris.es/debian
MOUNT=mnt                 #mount point directory
KERNELPIN=atleast         #default constraint for kernels
                          #[atleast|newest|exactly]
```

A.3.4 Pool Environment

This is the configuration file for the Pool component (the file must be named:*poolEnv.cfg* and placed in pc_LVM):

```
BASEDIR=/usr/share/PDRM   #base directory, where the scripts are placed

#Pool organization

POOL=/aplic/greig/pool    #pool location
DRELEASES=$POOL/releases  #releases directory
DKERNELS=$POOL/kernels    #kernels directory
ARCHIVES=$POOL/archives   #archives directory
SWAP=$POOL/swap           #swap directory
LOG=$POOL/log             #log file
ERRORLOG=$POOL/error_log  #error log file
KERNELPIN=atleast         #default constraint for kernels

BS="1024k"                #default disk size
COUNT=768                 #default number of disk blocks
IMAGEFILE="image.tar.gz"  #name of the package sent by the Store
MANDATORYPACKAGES="ssh"   #not basic packages which will be included in
                          #every base images
MIRROR=http://ftp.rediris.es/debian
PACKAGEFORMAT=deb         #default package format

# Domains defaults settings

MEMORY=1024               #MB of memory
```

```
CPU=1                      #number of CPUs
GATEWAY=192.168.1.30
NETMASK=255.255.255.0
BROADCAST=192.168.1.255
NETWORK=192.168.1.0
BRIDGE=brein0              #network bridge
DEFAULTSWAPSIZE=128        #default swap size (MB)
MOUNT=$POOL/mnt            #mount point directory
```

A.4 Security issues

This section describes how to avoid sending passwords through the network when executing a web service and when transferring data through SCP.

Sudoers file

We should adapt the sudoers file to allow the application server user to execute some commands without be prompted for passwords. The process described below should be done in pc_RC and pc_LVM.

In order to access to the sudoers file in mutual exclusion, as root execute:

```
visudo -f /etc/sudoers
```

and add the following:

```
# Cmnd alias specification
Cmnd_Alias      XENUTILS =      /usr/sbin/xm
Cmnd_Alias      DEBUTILS =      /usr/bin/dpkg, \
                                /usr/bin/apt-get, \
                                /usr/sbin/debootstrap
Cmnd_Alias      FSUTILS =       /sbin/mkfs.ext3, \
                                /bin/mount, \
                                /bin/umount, \
                                /usr/sbin/chroot, \
                                /bin/cp, \
                                /bin/cat, \
```

```
                                    /bin/echo, \
                                    /bin/rm, \
                                    /bin/mv, \
                                    /bin/mkdir, \
                                    /sbin/MAKEDEV, \
                                    /bin/chown, \
                                    /bin/sh, \
                                    /bin/tar, \
                                    /bin/ln

# User privilege specification
greig   ALL= NOPASSWD: DEBUTILS, FSUTILS, XENUTILS
```

SSH communication

To install SSH open a console an type as root:

```
apt-get install openssh-client openssh-server
```

To use SSH from pc_RC to pc_LVM without passwords, using RSA keys, follow these steps:

```
ssh-keygen
```

answer the question or accept default values. Then you should have *id_rsa* and *id_rsa.pub* files into your */.ssh* directory. Copy the id_rsa.pub file to remote machines (pc_LVM)

```
scp id_rsa.pub greig@pc_LVM:.ssh/.
```

Then, ssh to pc_LVM and append the contents of your public key into the authorized_keys file. If this file does not exist it will be created.

```
cat id_rsa.pub >> authorized_keys
```

Make sure that the permissions for .ssh directory are set to 700. Otherwise, change them by typing:

```
chmod 700 .ssh
```

Finally, the permissions of the authorized_keys file must be 644. Otherwise:

```
chmod 644 authorized_keys
```

Appendix B

Resource Fabrics Language specification

The aim of this appendix is to formalize the XML schemas used in the project for the creation and management of virtual machines. These schemas conform the Resource Fabrics Language (RFL) and its namespace is: *http://www.ac.upc.edu/rfl*. Throughout this documentation we present the first approach to the RFL (version 0.1), thus, it should not be considered as an stable version. The RFL elements are listed in the next page.

For each schema element, we present a brief definition including the type of the element and the multiplicity and type of the subelements that it contains. Also, we show a pseudo schema for an easy understanding of the described elements. These pseudo schemas use BNF-style conventions for the elements notation. That is, * denotes zero o more multiplicity, + for one or more occurrences and ? for zero or one occurrences.

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119. [38]

Table of contents

B.1 Communication elements

In this section we enumerate and explain the elements used in the communication between different components (refer to chapter Design 6 to see their interactions).

MachineDescription

Definition

This complex element describes the requirements that a new VM must accomplish. These are: how it will be named and which software and hardware characteristics it will have.

It MUST support the following elements:

- Name, with multiplicity one.
- ImageDescription, with multiplicity one.
- HardwareDescription, with multiplicity one.

Pseudo schema

```
<MachineDescription >
    <Name />
    <ImageDescription />
    <HardwareDescription />
</MachineDescription >
```

Example

```
<MachineDescription >
    <Name>VM-Lite</Name>
    <ImageDescription>
        <Architecture>i386</Architecture>
    </ImageDescription>
    <HardwareDescription>
    </HardwareDescription>
</MachineDescription >
```

ImageDescription

Definition

This element is used to describe the requirements that MUST be accomplished by the disk image of a virtual machine: it MAY specify a kernel to run in the VM; it MUST specify the architecture of the host machine; it MAY specify a set of software packages to be installed on the VM; it MAY specify a mirror from which download the necessary software; it MAY specify the size of the VM disk; it MAY specify the package format required for the VM and it MAY specify the release required for the VM.

It MUST support the following elements:

- Kernel, with multiplicity zero or one.
- Architecture, with multiplicity one.
- PackageList, with multiplicity zero or one.
- Mirror, with multiplicity zero or one.
- DiskSize, with multiplicity zero or one.
- Release, with multiplicity zero or one.

Pseudo schema

```
<ImageDescription>
    <Kernel />?
    <Architecture />
    <PackagesList />?
    <Mirror />?
    <DiskSize />?
    <Release />?
</ImageDescription>
```

Example

```
<ImageDescription>
    <Kernel>
        <Constraint>newest</Constraint>
```

```
    </Kernel>
    <Architecture>i386</Architecture>
    <Mirror>http://ftp.rediris.es/debian</Mirror>
    <DiskSize>1024k</DiskSize>
</ImageDescription>
```

HardwareDescription

Definition

This element is intended to describe the initial configuration of the VM. We are evaluating the possibility of replace this element by using JSDL [14].

It MUST support the following elements:

- Memory, with multiplicity zero or one.
- NumberOfCPU, with multiplicity zero or one.
- SwapSize, with multiplicity zero or one.

Pseudo schema

```
<HardwareDescription>
    <Memory />?
    <NumberOfCPU />?
    <SwapSize />?
</HardwareDescription>
```

Example

```
<HardwareDescription>
    <Memory>1024</Memory>
    <NumberOfVCPU>2</NumberOfVCPU>
    <SwapSize>512</SwapSize>
</HardwareDescription>
```

Requirements

Definition

This element describes the kernel and base system required for a VM identified by its name.

It MUST support the following elements:

- Name, with multiplicity one.
- Kernel, with multiplicity zero or one.
- Release, with multiplicity zero or one.

Pseudo schema

```
<Requirements>
    <Name>
    <Kernel>?
    <Release>?
</Requirements>
```

Example

```
<Requirements>
    <Name>VirtualMachineName</Name>
</Requirements>
```

SatisfiedRequirements

Definition

This is a complex type used to list the requirements that a host could satisfy without downloading anything from Internet.

It MUST support the following elements:

- Name, with multiplicity one.
- MachineHome, with multiplicity one.
- Kernel, with multiplicity zero or one.

- Release, with multiplicity zero or one.

Pseudo schema

```
<SatisfiedRequirements>
    <Name>
    <MachineHome>?
    <Kernel>?
    <Release>?
</SatisfiedRequirements>
```

Example

```
<SatisfiedRequirements>
    <Name>VirtualMachineName</Name>
    <MachineHome>/pool/VirtualMachineName/</MachineHome>
</SatisfiedRequirements>
```

InstallationDescription

Definition

This element specifies what is needed to install and how to perform this action.

It MUST support the following elements:

- Name, with multiplicity one.
- PackageList, with multiplicity zero or one.
- Release, with multiplicity one.
- Mirror, with multiplicity zero or one.
- DiskSize, with multiplicity zero or one.

Pseudo schema

```
<InstallationDescription>
    <Name/>
```

```
    <PackagesList/>?
    <Release/>
    <Mirror/>?
    <DiskSize/>?
</InstallationDescription>
```

Example

```
<InstallationDescription>
    <Name>VirtualMachineName</Name>
</InstallationDescription>
```

DiskInformation

Definition

This element describes the necessary information related to those locations of disk elements to create and run a VM (where the VM home is, where the kernel to run is, where the ramdisk is and where the file system is).

It MUST support the following elements:

- KernelLocation, with multiplicity one.
- KernelLocation, with multiplicity one.
- InitrdLocation, with multiplicity zero or one.
- DiskLocation, with multiplicity one.

Pseudo schema

```
<DiskInformation>/
    <MachineHome />
    <KernelLocation />
    <InitrdLocation />?
    <DiskLocation />
</DiskInformation>
```

Example

```
<DiskInformation>/
    <MachineHome>/pool/Machine/</MachineHome>
    <KernelLocation>/pool/archives/VM/vmlinuz-2.6.18-xen</KernelLocation>
    <InitrdLocation>/pool/archives/VM/initrd.img-2.6.18-i686</InitrdLocation>
    <DiskLocation>/pool/archives/VM/disk-VM</DiskLocation>
</DiskInformation>
```

CreateMachineDescription

Definition

This element describes the necessary information to create and run a VM: how it is named, where its disk is and its hardware characteristics.

It MUST support the following elements:

- Name, with multiplicity one.
- DiskInformation, with multiplicity one.
- HardwareDescription, with multiplicity one.

Pseudo schema

```
<CreateMachineDescription>
    <Name/>
    <DiskInformation/>
    <HardwareDescription>
</CreateMachineDescription>
```

Example

```
<CreateMachineDescription>
    <Name/>
    <DiskInformation>
        <KernelLocation>/pool/archives/VM/vmlinuz-2.6.18-xen</KernelLocation>
        <InitrdLocation>/pool/archives/VM/initrd.img-2.6.18-i686</InitrdLocation>
```

```
        <DiskLocation>/pool/archives/VM/disk-VM</DiskLocation>
    </DiskInformation>
    <HardwareDescription>
    </HardwareDescription>
</CreateMachineDescription>
```

MachineReference

Definition

This complex element has the necessary information to contact with a VM.

It MUST support the following elements:

- MachineIP, with multiplicity one.

Pseudo schema

```
<MachineReference>
    <MachineIP/>
</MachineReference>
```

Example

```
<MachineReference>
    <MachineIP>147.83.83.83<MachineIP/>
</MachineReference>
```

MachineIdentifier

Definition

This complex element has the necessary information to identify and contact with a VM.

It MUST support the following elements:

- Name, with multiplicity one.
- HostIP, with multiplicity one.

- HostPort, with multiplicity one.
- MachineIP, with multiplicity one.

Pseudo schema

```
<MachineIdentifier>
    <Name/>
    <HostIP/>
    <HostPort/>
    <MachineIP/>
</MachineIdentifier>
```

Example

```
<MachineIdentifier>
    <Name>SORMAVM<Name>
    <HostIP>147.83.23.24<HostIP/>
    <HostPort>8080<HostPort/>
    <MachineIP>147.83.83.83<MachineIP/>
</MachineIdentifier>
```

B.2 Basic complex elements

This section explains those complex elements that are subelements of the communication elements presented in the previous section (see section B.1).

PackageList

Definition

This element is a complex type specifying a set of Linux packages that the virtual machine MUST have installed. It also MUST have installed the dependencies of each package without needing any further specification.

It MUST support the following elements:

- Package, with multiplicity zero or more.

Pseudo schema

```
<PackagesList >
    <Package />*
</PackagesList>
```

Example The packages pdftk and vim and their dependencies must to be installed in the VM.

```
<PackageList>
    <Package>
        <PackageName>pdftk</PackageName>
    </Package>
    <Package>
        <PackageName>vim</PackageName>
    </Package>
</PackageList>
```

Package

Definition
This element is a complex type specifying a Linux package by its name.

It MUST support the following elements:

- PackageName, with multiplicity one.

Pseudo schema

```
<Package>
    <PackageName />
</Package>
```

Example Specification of package pdftk.

```
<Package>
    <PackageName>pdftk</PackageName>
</Package>
```

Kernel

Definition
This complex element describes the kernel that virtual machines SHOULD run. It is described in terms of an optional kernel version number and an optional modifier. Notice that it has no sense to specify the kernel version if we use the constraint *newest*.

It MUST support the following elements:

- KernelVersion, with multiplicity zero or one.
- Constraint, with multiplicity zero or one.

Pseudo schema

```
<Kernel>
    <KernelVersion />?
    </Constraint>?
</Kernel>
```

Example A kernel newest or equal than the kernel 2.6 MUST be provided.

```
<Kernel>
    <KernelVersion>2.6</KernelVersion>
    <Constraint>atleast</Constraint>
</Kernel>
```

Release

Definition
The release element is useful for specifying which release MUST be installed in the virtual machine. Alternatively we MAY specify just a package format.

Notice that, it has no sense to specify a release based on a format type in the field codename and a different format in the field PackageFormat.

It MUST support the following elements:

- Codename, with multiplicity zero or one.
- PackageFormat, with multiplicity one.

Pseudo schema

```
<Release>
    <Codename />?
    </PackageFormat>
</Release>
```

Example

```
<Release>
    <Codename>etch</Codename>
    <PackageFormat>deb</PackageFormat>
</Release>
```

B.3 Basic simple elements

This section explains those simple elements that are subelements of the communication elements (see section B.1).

Name

Definition
This element specifies a name for the VM. It SHOULD be unique within the provider's LAN to facilitate the VM migration between hosts.

The type of this element is xsd:string.

Pseudo schema

```
<Name>xsd:string</Name>
```

Example VirtualMachineName identifies a VM.

```
<Name>VirtualMachineName</Name>
```

Architecture

Definition
This element specifies the processor architecture required by the VM.

The type of this element is xsd:string.

Pseudo schema

```
<Architecture>xsd:string</Architecture>
```

Example The VM MUST run over an amd64 processor.

```
<Architecture>amd64</Architecture>
```

KernelVersion

Definition
This is a simple type for identifying the family version of a Linux kernel.

The type of this element is xsd:string.

Pseudo schema

```
<KernelVersion>xsd:string</KernelVersion>
```

Example

```
<KernelVersion>2.6.18</KernelVersion>
```

Constraint

Definition
This constraint element specifies a requirement that the kernel of the VM must accomplish.

The type of this element is rfl:ConstraintType.

Pseudo schema

```
<Constraint>rfl:ConstraintType</Constraint>
```

Example The newest Linux kernel is required.

```
<Constraint>newest</Constraint>
```

Codename

Definition
This element specifies the name of the release that MUST be installed in the VM.

The type of this element is xsd:string.

Pseudo schema

```
<Codename>xsd:string</Codename>
```

Example

```
<Codename>dapper</Codename>
```

PackageFormat

Definition
This element identifies the file format of the software Linux packages that the VM MUST have installed.

The type of this element is rfl:PackageFormatType.

Pseudo schema

```
<PackageFormat>rfl:PackageFormatType</PackageFormat>
```

Example We want that the VM will be able to work on a debian-based system.

```
<PackageFormat>deb</PackageFormat>
```

PackageName

Definition
Name of a Linux package. It represents the package which is identified by package's name and its dependencies.

The type of this element is xsd:string.

Pseudo schema

```
<PackageName>xsd:string</PackageName>
```

Example The VM MUST have the pdftk program installed properly.

```
<PackageName>pdftk</PackageName>
```

Mirror

Definition
A Linux mirror address from where to download the software for the VMs.

The type of this element is xsd:string.

Pseudo schema

```
<Mirror>xsd:string</Mirror>
```

Example

```
<Mirror>http://ftp.rediris.es/debian</Mirror>
```

MachineHome

Definition
This element identifies the directory that MUST be used to store VM configuration file.

The type of this element is xsd:string.

Pseudo schema

```
<MachineHome>xsd:string</MachineHome>
```

Example

```
<MachineHome>/pool/Machine/</MachineHome>
```

KernelLocation

Definition

This element shows the path to the kernel that the VM MUST run.

The type of this element is xsd:string.

Pseudo schema

```
<KernelLocation>xsd:string</KernelLocation>
```

Example

```
<KernelLocation>/pool/archives/VM/vmlinuz-2.6.18-xen</KernelLocation>
```

InitrdLocation

Definition

This element shows the path to the kernel that the VM MAY use at boot time.

The type of this element is xsd:string.

Pseudo schema

```
<InitrdLocation>xsd:string</InitrdLocation>
```

Example

```
<InitrdLocation>/pool/archives/VM/initrd.img-2.6.18-i686</InitrdLocation>
```

DiskLocation

Definition

This element shows the path to the file system that the VM MUST use as a file system.

The type of this element is xsd:string.

Pseudo schema

```
<DiskLocation>xsd:string</DiskLocation>
```

Example

```
<DiskLocation>/pool/archives/VM/disk-VM</DiskLocation>
```

DiskSize

Definition
Size in bytes of the hard disk of the VM. Bytes can be followed by a multiplicative suffix: the suffix KB is equivalent to 1000 bytes, kB to 1024 bytes, MB to 1000*1000 bytes, M to 1024*1024 bytes, GB to 1000*1000*1000, G to 1024*1024*1024, and so on for T, P, E, Z, Y.

Pseudo schema

```
<DiskSize>xsd:string</DiskSize>
```

Example A disk of 1024 Kilobytes.

```
<DiskSize>1024k</DiskSize>
```

Memory

Definition
Memory in MB allocated to the VM.

The type of this element is xsd:string.

Pseudo schema

```
<Memory>xsd:string</Memory>
```

Example

```
<Memory>1024</Memory>
```

NumberOfVCPU

Definition
Number of virtual CPUs that the VM MAY use.

The type of this element is xsd:string.

Pseudo schema

```
<NumberOfVCPU>xsd:string</NumberOfVCPU>
```

Example

```
<NumberOfVCPU>2</NumberOfVCPU>
```

SwapSize

Definition
Size in bytes of the swap disk. Bytes can be followed by a multiplicative suffix: the suffix KB is equivalent to 1000 bytes, kB to 1024 bytes, MB to 1000*1000 bytes, M to 1024*1024 bytes, GB to 1000*1000*1000, G to 1024*1024*1024, and so on for T, P, E, Z, Y.

The type of this element is xsd:string.

Pseudo schema

```
<SwapSize>xsd:string</SwapSize>
```

Example

```
<SwapSize>1024k</SwapSize>
```

HostIP

Definition
IP address of the host where the VM is installed.

The type of this element is xsd:string.

Pseudo schema

```
<HostIP>xsd:string</HostIP>
```

Example

```
<HostIP>147.83.23.24<HostIP/>
```

HostPort

Definition
Port to contact with the host of the VM.

The type of this element is xsd:string.

Pseudo schema

```
<HostPort>xsd:string</HostPort>
```

Example

```
<HostPort>8080<HostPort/>
```

MachineIP

Definition
IP address of the VM.

The type of this element is xsd:string.

Pseudo schema

```
<MachineIP>xsd:string</MachineIP>
```

Example

```
<MachineIP>147.83.83.83<MachineIP/>
```

B.4 RFL types

This section specifies the enumeration types used within RFL.

ConstraintType

This type is used in the context of the kernel element specified in section B.2. These MAY provide modifiers for the kernel required by the user.

The following kernel modifiers MUST be supported.

Normative RFL Name	Definition
exactly	If a kernel is provided it MUST belongs to the family kernel version specified in the kernel version field
atleast	If a kernel is provided it MUST be at least as newest as the kernel specified in the kernel version field
newest	If a kernel is provided it MUST be the newest one available

Table B.1: Constraint Type specification

PackageFormatType

This type is used in the context of the release element specified in section B.2. This type enumerates the package format types that SHOULD be supported

The following package formats SHOULD be supported.

Normative RFL Name	Definition
deb	Debian's based system package format
rpm	Red Hat Package Manager. Package format from rpm based systems

Table B.2: Package Format Type specification

Appendix C

Scripts specification

The main purpose of this chapter is to expose the API specification of the lowest level components to help any future developer to use or extend these elements. These are: Checker, Image Installer and Local Resource Manager from Local Virtualization Manager component and Image Builder from the Resource Coordinator component, both within Resource Fabrics layer.

This chapter is structured by components. For each one, we specify its operations in terms of a concise description of its functionality, a list of the mandatory parameters that must be provided to use the operation properly, a list of the optional parameters used to modify the operation's behaviour and the operation's output. We also offer, for each element, a hook specification to extend easily these components. It is provided because these components have to call to release-specific functions to construct a system based on that release. Thereby, we encapsulate these functions in a hook script named as the name of the release (e.g. deb). The suitable hook will be loaded into our components at execution time.

C.1 Checker

This LVM component is in charge of the preparation of the VM home space and checking what requirements can be satisfied as a result of the Pool caching.

Operation: prepare

Description

This operation prepares a directory where to store the information related to the future VM named

by *name*.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Operation: check

Description

This operation is used to check if the Pool owns in the cache a requirement (kernel, release). In case that the checker requirements can be satisfied they will be copied to the VM *name* associated directory.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Optional parameters

−−**release [codename**]: checks whether a release is available locally or not. The optional codename parameter specifies which release we are checking for. If there is no codename specified, then any release will be fine. If a codename is provided then the format parameter is mandatory.

−−**format packageFormat** : indicates which is the type of the package format for the release. Therefore, it determines which hook must be used.

−−**kernel [version**]: checks whether a kernel is available locally or not. The optional version parameter specifies which kernel version we are checking for (e.g. −−kernel 2.6.18). If no version is provided any kernel will be fine.

−−**constraint {newest | atleast | exactly}** : this parameter depends on the previous one. We impose a constraint over the kernel. This constraint should be one of these three: *newest*, we are checking for the newest kernel version available; *exactly*, if we are checking if there is available any kernel of the same version family that the parameter *version* specified before; and *atleast*, if we require a kernel from a family at least as new as the specified in *version* parameter.

Output

codename : name of the available release which satisfies the checked requirements or blank if there is no one available.

packageFormat : name of the file format of the available release or blank if no one available.

version : version of the available kernel that satisfies what was checked or blank anyway else.

The script will answer us, only for what we had checked. For instance, if we have checked for a release, then *codename* and *packageFormat* are displayed. The output is blank if there is no release that could satisfy what is being checked. If we check for a kernel, then its *version* is outputted when the requirement could be satisfied, otherwise blank. If we ask for both, release and kernel, then the output is given in this order: *codename, packageFormat, version.*

Operation: remove

Description

This operation is used to remove the information related to a VM stored locally

Mandatory parameters

−−name name : uniquely identifier for a VM.

Operation: help

Description

This operation shows a short usage summary.

Checker hook

Function: search_package

Description

This function searches a package on all available package lists. For each match, it prints out the package name and a short description.

Parameters

[1] **package**: package name.

C.2 Image Builder

Operation: create-store

Description

This operation is used to create the Store and all of its directories. If a Store exists it will be removed.

Operation: create-image

Description

This operation is used to create a disk image for a new VM. It creates the Store if it does not exist.

Mandatory parameters

−−**arch [arch**]: processor architecture of the host machine.

Optional parameters

−−**base-system** : this parameter indicates that it is required to construct a base system.

−−**release codename** : release name.

−−**format packageFormat** : indicates which is the type of package format for the release (deb, rpm, etc.). Therefore, it determines which hook must be used.

−−**kernel [version** [−−constraint {newest | atleast | exactly}]]: it specifies the required kernel. We might provide a version for this kernel and impose a constraint over the kernel. This constraint should be one of these three: *newest*, for the newest kernel available; *exactly*, for any available kernel of the same *version* family; and *atleast*, if we require a kernel from a family at least as new as the specified in *version*. Any kernel will be provided if neither a version nor a constraint is specified.

−−**packages list** : comma separated list of software packages to add to the base system.

−−**mirror mirror** : URL from where to download any necessary software.

Output

output : absolute pathname to the required data and image identifier.

release : base system release provided.

packageFormat : format package type of the release.

Operation: remove-image

Description

To remove the information stored in the Store related to a VM.

Mandatory parameters

−−**identifier id** : identifier of an image. It is provided in the output of the *create-image* function.

Operation: help

Description

This operation shows a short usage summary.

Image Builder hook

Function: download

Description

Downloads a package and its dependencies.

Parameters

[1] **release**: codename of the package release.

[2] **output**: absolute path directory to store the package and its dependencies.

[3] **architecture**: processor architecture.

[4] **package**: package name.

Function: search_package_arch

Description

This function searches a package on the package lists of a target processor architecture. For each match, it prints out the package name and a short description.

Parameters

[1] **package**: package name.

[2] **architecture**: processor architecture.

Function: show

Description

It displays the package records for the package.

Parameters

[1] **package**: package name.

[2] **architecture**: processor architecture.

Function: depends

Description

Prints a list with every dependency a package has.

Parameters

[1] **package**: package name.

[2] **architecture**: processor architecture.

Function: bootstrap_tarball

Description

This function builds a tarball with all the necessary packages to construct a base system.

Parameters

[1] **architecture**: processor architecture.

[2] **output file**: output (tgz) file.

[3] **release**: codename of the release.

[4] **mount point**: mount point.

[5] **mirror**: URL from where to download any necessary software.

[6] **packages**: comma separated list of extra packages to include in the base system.

C.3 Image Installer

Operation: install

Description

Creates an image disk on a file and installs a system in it.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

−−**release codename** : base system release name.

−−**format packageFormat** : indicates which is the type of the package format for the release. Therefore, it determines which hook must be used.

Optional parameters

−−**packages list** : comma separated list of extra software packages to include in the base system.

−−**mirror mirror** : URL from where download any necessary software.

−−**disk-size size** : size in bytes of the VM's disk. Bytes can be followed by a multiplicative suffix: the suffix KB is equivalent to 1000 bytes, kB to 1024 bytes, MB to 1000*1000 bytes, M to 1024*1024 bytes, GB to 1000*1000*1000, G to 1024*1024*1024, and so on for T, P, E, Z, Y.

Output

disk : absolute path to VM disk.

kernel : absolute path to VM kernel.

ramdisk : absolute path to VM ramdisk.

Operation: remove

Description

This operation is used to remove the information related to a VM stored locally

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Operation: help

Description

This operation shows a short usage summary.

Image Installer hook

Function: bootstrap_unpack

Description

Installs the packages from a tarball file instead of downloading from a mirror site.

Parameters

[1] **file**: tgz file with the packages.

[2] **packages**: comma separated list of extra software packages to install.

[3] **release**: codename of the release.

[4] **mount point**: mount point.

Function: install_modules

Description

Performs the installation of a kernel and its modules for a file system.

Parameters

[1] **mount point**: path to the root of the VM's file system.

[2] **modules**: linux-modules package name.

[3] **kernel**: linux-image package name.

C.4 Local Resource Manager

Operation: construct

Description

This operation is used to construct the files needed for a new VM. These files are stored in the VM associated directory in the Pool.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

−−**kernel kernel** : path to VM kernel.

−−**disk disk** : path to VM disk.

Optional parameters

−−**mem memory** : amount of memory in MB allocated to the VM.

−−**cpu cpu** : number of CPUs available for the VM.

−−**ramdisk ramdisk** : path to VM ramdisk.

−−**swap size** : swap disk size in bytes. Bytes can be followed by a multiplicative suffix: the suffix KB is equivalent to 1000 bytes, kB to 1024 bytes, MB to 1000*1000 bytes, M to 1024*1024 bytes, GB to 1000*1000*1000, G to 1024*1024*1024, and so on for T, P, E, Z, Y.

Operation: run

Description

This operation runs a VM.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Operation: create

Description

This operation constructs and runs a VM.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

−−**kernel** : path to VM kernel.

−−**disk** : path to VM disk.

Optional parameters

−−**mem** : amount of memory in MB allocated to the VM.

−−**cpu** : number of CPUs available for the VM.

−−**ramdisk** : path to VM ramdisk.

−−**swap** : swap disk size in bytes. Bytes can be followed by a multiplicative suffix: the suffix KB is equivalent to 1000 bytes, kB to 1024 bytes, MB to 1000*1000 bytes, M to 1024*1024 bytes, GB to 1000*1000*1000, G to 1024*1024*1024, and so on for T, P, E, Z, Y.

Operation: shutdown

Description

Shutdowns a VM.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Operation: destroy

Description

Destroys a VM, including associated files.

Mandatory parameters

−−**name name** : uniquely identifier for a VM.

Operation: help

Description

This operation shows a short usage summary.

Appendix D

Glossary

ABI: Application Binary Interface. Interface which allows programs to access hardware through the Instruction Set Architecture (ISA) and the system call interface.

API: Application Programming Interface. High level interface of an application.

Axis: Framework from Apache for constructing SOAP processors such as clients, servers, gateways,etc.

BREIN: Business objective driven REliable and Intelligent grids for real busiNess. European research project.

Domain: Execution context that contains a running virtual machine.

Domain 0: Xen domain which is the responsible for managing the system.

EA: Economy Agent. EERM component responsible for deciding if a task is technically and economically feasible and calculating its price.

EC: Estimator Component. EERM component in charge of estimating the expected impact of executing a task on the utilization of resources.

EERM: Economically Enhanced Resource Manager. SORMA component used to separate the economic from the technical part in SORMA.

ERM: Economic Resource Management. EERM component with the onus of ensuring an efficient use of local resources.

FTP: File Transfer Protocol. Application protocol used to exchange files.

Full virtualization: An approach of virtualization which requires no modifications to the hosted operating system, providing the illusion of a complete system of real hardware devices.

GMM: Grid Market Middleware. Scalable middleware based on a peer-to-peer structured overlay network which offers core services for grid markets.

Grid: Distributed computing paradigm focused on large-scale resource sharing.

Guest: Process or system running on a VM.

Host: Underlying platform that supports a VM.

Hypervisor: Software that allows multiple virtual machines to be multiplexed on a single physical machine.

ICT: Information and Communication Technologies.

IT: Information Technologies.

ISA: Instruction Set Architecture. Instruction set which allows software to access to the hardware.

JESS: Java Expert System Shell. Rule engine for the Java platform.

JSDL: Job Submission Description Language. XML based specification from the Global Grid Forum used to describe the requirements of computational jobs for submission to resources.

LRU: Least Recently Used. Replacement policy used in caches in case of conflict.

LVM: Local Virtualization Manager. RF component placed in every resource to offer functionalities for the management of its virtual machines.

Paravirtualization: An approach to virtualization which requires operating system to be ported in order to run in a virtual machine.

Planned reservation: Agreement by which a client reserves in advance a VM.

PM: Policy Manager. EERM component that stores and manages policies to adapt EERM's behaviour at runtime.

RE: Rule Engine. Software that permits the isolation of business policies from the application design.

Reservation: Client's intention of reserving a VM in advance.

RC: Resource Coordinator. RF component that chooses and helps to prepare a resource for creating new virtual machines.

RF: Resource Fabrics. SORMA component that provides access to and control of the resources inside a resource provider's LAN.

RFL: Resource Fabrics Language. XML based specification for describing the interactions among RF components.

RPM: Red Hat Package Manager.

SCP: Secure copy. Protocol that works over SSH for transferring data.

SLA: Service Level Agreement. Agreement between clients and service providers. It defines the level of service with information about agreed priorities, responsibilities, guarantees, performance metrics, penalties to apply in case of SLA violation, etc.

SOAP: Simple Object Access Protocol. It is an XML-based communication protocol and encoding format for inter-application communication.

SORMA: Self-Organizing ICT Resource Management. European project to develop a platform that allows ICT resources trading on demand.

SPG: System Performance Guard. EERM component in charge of the fulfilment of the SLAs.

Tycho: Resource discovery framework and messaging system for distributed applications.

VM: Virtual Machine. Environment in which a hosted operating system runs, providing the abstraction of a dedicated machine.

VMM: Virtual Machine Monitor. See hypervisor.

WS: Web Service. Defined by the World Wide Web Consortium as *a software system designed to support interoperable Machine to Machine interaction over a network*.

Xen: open-source VMM for the x86 processor architecture.

Bibliography

[1] Sorma Web Page http://sorma-project.eu/.

[2] Xen web page. http://xen.org.

[3] I. Foster, "What is the grid: a three point check-list," tech. rep., 2002.

[4] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations," *Lecture Notes in Computer Science*, vol. 2150, pp. 1–??, 2001.

[5] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[6] Wikipedia The Free Encyclopedia http://en.wikipedia.org.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization.," in *SOSP* (M. L. Scott and L. L. Peterson, eds.), pp. 164–177, ACM, 2003.

[8] *Xen Users' Manual v3.0.*

[9] Inside Xen 3.0 A XenSource White Paper. http://xensource.com.

[10] S. Hand, T. Harris, E. Kotsovinos, and I. Pratt, "Controlling the xenoserver open platform," 2002. In Proceedings of the 6th IEEE Conference on Open Architectures and Network Programming (IEEE OPENARCH'03).

[11] Jess web page http://herzberg.ca.sandia.gov.

[12] C. L. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem," pp. 324–341, 1990.

[13] P. Chacín, X. León, R. Brunner, F. Freitag, and L. Navarro, "Core services for grid markets," in *CoreGrid Symposium*, 2008.

[14] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva, "Job submission description language (jsdl) specification, version 1.0," tech. rep., Global Grid Forum, 2005.

[15] Axis 2 web page http://ws.apache.org/axis2/.

[16] Apache Software Fundation http://www.apache.org/.

[17] XMLBeans web page http://xmlbeans.apache.org/.

[18] Brein web page. http://www.eu-brein.com/.

[19] JDOM web page http://www.jdom.org.

[20] Debootstrap web page. http://packages.debian.org/debootstrap.

[21] Wget web page. http://packages.debian.org/wget.

[22] Dpkg web page. http://packages.debian.org/dpkg.

[23] Apt web page. packages.debian.org/apt.

[24] Tycho web page http://www.acet.reading.ac.uk/projects/tycho/index.php.

[25] T. D. Moreton, I. A. Pratt, and T. L. Harris, "Storage, mutability and naming in pasta."

[26] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris, "Global-scale service deployment in the xenoserver platform."

[27] A. . Matsunaga, M. . Tsugawa, M. . Zhao, L. . Zhu, V. Sanjeepan, S. . Adabala, R. . Figueiredo, H. Lam, and J. . Fortes, *On the Use of Virtualization and Service Technologies to Enable Grid-Computing*, vol. 0. Berlin/Heidelberg: Springer, 2005.

[28] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. O. Figueiredo, "Vmplants: Providing and managing virtual machine execution environments for grid computing.," in *SC*, p. 7, IEEE Computer Society, 2004.

[29] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler, "Softudc: A software-based data center for utility computing," *Computer*, vol. 37, no. 11, pp. 38–46, 2004.

[30] Smart Frog http://www.hpl.hp.com/research/smartfrog/.

[31] Globus Virtual Workspaces web page http://workspace.globus.org.

[32] K. Keahey, "Globus Virtual Workspaces," Supercomputing 2007, Reno, NV. November 2007 http://workspace.globus.org/papers/SC07-keahey.ppt.

[33] X. Jiang and D. Xu, "Soda: a service-on-demand architecture for application service hosting utility platforms," 2003.

[34] Amazon Elastic Computer Cloud http://aws.amazon.com/ec2.

[35] Elastic Server http://elasticserver.com/.

[36] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines.," in *NSDI*, USENIX, 2005.

[37] D. Spence and T. Harris, "Xenosearch: Distributed resource discovery in the xenoserver open platform.," in *HPDC*, pp. 216–225, IEEE Computer Society, 2003.

[38] S. Bradner, *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. IETF, 1997.

[39] M. Macías, O. Rana, G.Smith, J. Guitart, and J. Torres, "Maximising revenue in grid markets using an economically enhanced resource manager," September 2007.

[40] P. R. Barham, B. Dragovic, K. A. Fraser, S. M. Hand, T. L. Harris, A. C. Ho, E. Kotsovinos, A. V. Madhavapeddy, R. Neugebauer, I. A. Pratt, and A. K. Warfield, "Xen 2002," Tech. Rep. UCAM-CL-TR-553, University of Cambridge, Computer Laboratory, Jan. 2003.

[41] T. Püschel, N. Borissov, M. Macías, D. Neumann, J. Guitart, and J. Torres, "Economically enhanced resource management for internet service utilities," *Lecture Notes on Computer Science (LNCS), Vol. 4831, pp. 335-348 8th International Conference on Web Information Systems Engineering (WISE'07)*.

[42] T. Püschel, N. Borissov, D. Neumann, M. Macías, J. Guitart, and J. Torres, "Extended resource management using client classification and economic enhancements," *Information and Communication Technologies and the Knowledge Economy, Volume 4 Expanding the Knowledge Economy; Issues, Applications, Case Studies; Part 1 17th eChallenges e-2007 Conference & Exhibition (e-2007)*.

[43] Sudoers manual. http://www.sudo.ws/sudo/man/sudoers.html.