Universidad Jaume I

Memoria del proyecto

Estrategia de ahorro energético y aprovechamiento de recursos para sistemas de Cloud Computing

Francisco José Clemente Castelló

SIE043 - Trabajo fin de máster

Máster en Sistemas Inteligentes

4 de septiembre de 2012

Proyecto dirigido por:

Rafael Mayo Gual

Resumen

El presente documento corresponde con el trabajo fin de máster "Estrategia de ahorro energético y aprovechamiento de recursos para sistemas de Cloud Computing" para la asignatura SIE043 del máster en Sistemas Inteligentes.

La energía, cada vez más, es un bien escaso que hay que conservar y utilizar de forma eficiente.

Una de las tendencias más recientes y actuales de tecnología TI para los modelos de negocio de las empresas es el Cloud Computing. El Cloud Computing es una tecnología consistente en la prestación de recursos y servicios computacionales a través de una red o de Internet. Pero, para desplegar una infraestructura de Cloud Computing se requiere una gran cantidad de clústers de computadores.

El aumento del número de nodos en *clusters* de computadores así como el incremento de la frecuencia de los procesadores en éstos, ha llevado a que el consumo de energía eléctrica en este tipo de plataformas se haya convertido en un factor muy importante a considerar. De hecho, importantes empresas e instituciones relacionadas con el campo de la computación de altas prestaciones (High Perfomance Computing) dedican gran parte de sus esfuerzos a desarrollar nuevos métodos de ahorro de energía en *clusters* de computadores. Además, en periodos de baja demanda se desperdicia parte de la capacidad de procesamiento de los clústers. Por tanto, es necesario idear una estrategia que permita aprovechar mejor los recursos físicos disponibles y ahorrar energía, todo ello sin perjudicar la calidad del servicio.

En este proyecto se presenta una pequeña aproximación a este fin, basada en la idea de virtualización, y del apagado y encendido remoto, selectivo y automático de nodos. Mediante el uso de la virtualización se ha desarrollado un software que permite agrupar máquinas virtuales de forma automática en función de la carga, permitiendo apagar o encender nodos según convenga y aprovechando mejor los recursos disponibles a la vez que se ahorra energía.

En este proyecto se ha utilizado el monitor de máquinas virtuales o hipervisor *Xen* como principal herramienta de virtualización. Una de las características destacadas de este hipervisor es la posibilidad de migrar máquinas virtuales en vivo, es decir, migrarlas sin ser apagadas ni tan siquiera pausadas. Esta característica permite que un servicio dado por una máquina virtual no sea interrumpido aunque se realice la migración de la máquina virtual a un nodo remoto.

Palabras clave

Cluster de computadores, Ahorro de energía, Virtualización, Cloud Computing, Xen

Índice general

Índice general I

Índice de figuras V

Índice de códigos fuente VII

1 Introducción **1**
1.1. Motivación del proyecto 1
1.2. Estado del arte 2
1.3. Objetivos del proyecto 3
1.4. Entorno y estado inicial 4
1.5. Organización de la memoria 5

2 Planificación y evaluación de recursos **7**
2.1. Planificación temporal 7
2.1.1. Identificación de tareas 7
2.1.2. Estimación de la duración de las tareas y relaciones de precedencia 14
2.1.3. Diagrama de planificación temporal 14
2.2. Estimación de recursos 14
2.3. Estimación de costes 17

3 Desarrollo y resultados del proyecto **19**
3.1. Virtualización 19
3.1.1. Tipos de vitualización 20
3.1.2. Monitor de máquinas virtuales o Hipervisor 21
3.1.3. Virtualización completa 22
Virtualización de plataforma 23
Procesamiento virtual 24
3.1.4. Virtualización Parcial 25
Memoria Virtual (Memoria de intercambio o Swap) . . 25
Almacenamiento virtual 25
Redes virtuales 25
Escritorios virtuales 26

3.2. Cloud Computing 27
3.2.1. Modelos de servicio 27
3.2.2. Tipos de nube 28
3.2.3. Características 29
3.3. Documentación de Xen 31
3.3.1. Migración en vivo de máquinas virtuales 32
3.3.2. API de *Xen* 33
3.3.3. Librería *Libvirt* 34
3.4. Preparación del *clúster* 34
3.4.1. Instalación de equipos 35
3.4.2. Instalación del sistema operativo en los equipos del clúster 35
3.4.3. Configuración de la red 36
3.4.4. Configuración SSH entre equipos 37
3.5. Gestión de máquinas virtuales 37
3.5.1. Creación de máquinas virtuales 37
3.5.2. Configuración de la red de las máquinas virtuales 39
3.6. Configuración para la migración con *Xen* 40
3.6.1. Configuración de *Xen* 41
3.6.2. Sistema de ficheros NFS 42
3.6.3. Configuración de puertos 42
3.7. Programación de las librerías intermedias 43
3.7.1. Librería *CPUinfo* 43
3.7.2. Librería *Servidor* 44
3.7.3. Librería *Nodo* 45
3.7.4. Librería *Virtual Machine* 47
3.8. Demonios para los nodos 48
3.8.1. Demonio *enersis-cpu* 49
3.8.2. Demonio *enersis-servidor* 49
3.8.3. Arrancador *enersis* 50
3.9. Demonio para el *frontend* 51
3.9.1. Demonio *enersis-frontend* 51
3.9.2. Arrancador *enersis* 54
3.10. Ficheros de configuración 54
3.10.1. Fichero *enersis-servidor.conf* 54
3.10.2. Fichero *enersis-frontend.conf* 55
3.11. Arquitectura del sistema 55

4 Desviaciones del proyecto **57**
4.1. Diagrama de planificación temporal realizado 57
4.2. Costes finales del proyecto 59

5 Conclusiones y trabajo futuro **61**
5.1. Conclusiones 61
5.2. Trabajo futuro 62

Bibliografía **63**

A Código fuente realizado **65**
A.1. Ficheros fuente para los nodos 65
A.1.1. Estructura de ficheros 65
A.1.2. Demonio *enersis-cpu* 68
A.1.3. Demonio *enersis-servidor* 70
A.1.4. Librería *CPUinfo.py* 72
A.1.5. Librería *servidor.py* 74
A.1.6. Fichero de configuración *enersis-servidor.conf* 76
A.1.7. Arrancador *enersis* 77
A.2. Ficheros fuente para el *frontend* 79
A.2.1. Estructura de ficheros 79
A.2.2. Demonio *enersis-frontend* 81
A.2.3. Librería *Nodo.py* 87
A.2.4. Librería *VirtualMachine.py* 89
A.2.5. Fichero de configuración *enersis-frontend.conf* 91
A.2.6. Arrancador *enersis* 92

Índice de figuras

2.1. Identificación de tareas. 9
2.2. Duración y precedencias de las tareas. 15
2.3. Diagrama de Gantt. 16

3.1. Esquema hipervisor Tipo I. Nativo 21
3.2. Esquema hipervisor Tipo II. Aplicación hipervisora 22
3.3. Esquema de procesamiento virtual 24
3.4. Logical Volume Management 26
3.5. Escritorio virtual. EyeOs 26
3.6. Modelos de Cloud Computing 28
3.7. Modelo tradicional 30
3.8. Modelo de Cloud Computing 30
3.9. Esquema de *Xen*. 32
3.10. API de Xen. 33
3.11. Arquitectura de Libvirt. 34
3.12. Montaje del *clúster*. 35
3.13. Conexión del cable Wake On LAN. 36
3.14. Interfaz de Virtual Machine Manager. 38
3.15. Configuración de la red con *Xen* mediante un puente. 40
3.16. Arquitectura Enersis. 56

4.1. Diagrama de Gantt de seguimiento 58

A.1. Estructura de ficheros Enersis para los nodos. 66
A.2. Estructura de ficheros Enersis para el *frontend*. 80

Índice de códigos fuente

A.1. Demonio *enersis-cpu* 68
A.2. Demonio *enersis-servidor* 70
A.3. Librería *CPUinfo.py* 72
A.4. Librería *servidor.py* 74
A.5. Fichero de configuración *enersis-servidor.conf* 76
A.6. Arrancador *enersis* 77
A.7. Demonio *enersis-frontend* 81
A.8. Librería *Nodo.py* 87
A.9. Librería *VirtualMachine.py* 89
A.10. Fichero de configuración *enersis-frontend.conf* 91
A.11. Arrancador *enersis* 92

CAPÍTULO 1

Introducción

Índice

1.1. Motivación del proyecto 1
1.2. Estado del arte 2
1.3. Objetivos del proyecto 3
1.4. Entorno y estado inicial 4
1.5. Organización de la memoria 5

1.1. Motivación del proyecto

La computación en nube o Cloud Computing es una nueva tecnología de temática muy actual que está teniendo un gran auge en la mayoría de empresas actuales. Consiste en la prestación de recursos y servicios computacionales a través de una red o Internet. Este nuevo paradigma de computación abre un gran abanico de posibilidades y nuevos modelos de negocio para las empresas. Mediante este nuevo paradigma, una empresa puede flexibilizar mucho más los recursos computacionales que necesite en cada instante, pagando únicamente por los que está utilizando.

Por otro lado, todo este nuevo concepto necesita detrás una gran infraestructura hardware para poder centralizar todos los recursos en las empresas que se dediquen a prestar los servicios de Cloud Computing. La mayoría de estas empresas, al igual que otros centros de procesamiento de datos de universidades e instituciones de investigación, utilizan una gran cantidad de *clusters* de computadores para ofrecer sus máquina virtuales, su capacidad de cálculo y

otros servicios de cómputo necesarios. Cada vez más, se necesitan mayor cantidad de *clusters* de computadores para satisfacer las necesidades del mercado actual, con lo que el consumo de energía aumenta de una forma considerable. Es por ello que deben surgir nuevas aplicaciones, estrategias y otras tecnologías para reducir el consumo excesivo de energía minimizando además el impacto medioambiental que ello conlleva.

Con frecuencia, en estas empresas e instituciones se alternan periodos de alta actividad, con grandes necesidades de uso, con periodos de baja demanda, en cuyo caso se desperdicia parte de la capacidad de procesamiento del *cluster*. Algunos servicios, como el correo electrónico, pasan parte del tiempo de servicio procesando muy poca información en horario nocturno, debido a las bajas peticiones de los usuarios en dicho horario. En estos servicios, se desperdicia parte de la capacidad de procesamiento del *cluster* que almacena el servicio, en este caso máquinas virtuales, por tanto se desperdicia energía. La tecnología fundamental en la que se sostiene el Cloud Computing es la virtualización. Por tanto, conviene investigar en este ámbito para obtener un ahorro de energía. Mediante la utilización de máquinas virtuales es posible ahorrar energía, agrupando varias máquinas virtuales en un mismo nodo, pudiendo de esta forma apagar los nodos que no se están utilizando.

En este proyecto, en primer lugar se estudia la migración de máquinas virtuales en vivo (migrar las máquinas virtuales sin ser apagadas ni pausadas) utilizando el hipervisor de máquinas virtuales *Xen* [1], para, en segundo lugar presentar una estrategia de ahorro de energía (denominada *Enersis*) en la cual, los nodos se enciendan o se apaguen remotamente en función de su carga de trabajo debido a las máquinas virtuales que contenga, migrando dichas máquinas virtuales entre los nodos según convenga. Esta estrategia es de gran interés para las empresas de Cloud Computing ya que permite un ahorro energético considerable sin perder la calidad del servicios que están ofreciendo. En este proyecto se pretende mostrar una visión de las posibilidades de ahorro energético que puede ofrecer la migración en vivo de máquinas virtuales, agrupando las máquinas virtuales en los nodos estrictamente necesarios mediante la estrategia desarrollada en este proyecto.

1.2. Estado del arte

El concepto de Cloud Computing es un concepto relativamente reciente aunque su origen fue ya anunciado desde la década de 1960. Con el auge de Internet y el aumento de la velocidad de la lineas de comunicación se hace posible la extensión de este concepto a través de redes como Internet. Empresas como Amazon, en 2006, lanzaron una de las primeras plataformas de prestación de servicios de cómputo para empresas invirtiendo en grandes centros de

datos. Actualmente, la temática del Cloud Computing se está empezando a aplicar a prácticamente todos los niveles del entorno de computación actual, de forma que la mayoría de empresas y usuarios están empezando a utilizar en su tareas cotidianas este nuevo concepto.

La temática acerca del ahorro de energía en centros de datos es de gran interés tanto para las empresas de prestación de servicios de Cloud Computing como para la comunidad científico-tecnológica actual: la frecuencia a la que operan los procesadores de los nodos y el aumento de la cantidad de los mismos, está llevando a que este tipo de plataformas consuma gran cantidad de recursos energéticos, lo que las convierte en opciones no deseadas a nivel económico para muchas aplicaciones. Por esta razón, muchas empresas e instituciones dedican gran parte de sus esfuerzos a buscar nuevas soluciones o alternativas para minimizar el consumo energético en los *clusters* de computadores.

Entre las técnicas más utilizadas actualmente para reducir el consumo energético de los *clusters* de computadores están la técnica DVFS (Dynamic Voltage and Frequency Scaling) [4], el encendido y apagado selectivo de nodos [5] y la virtualización [6].

1.3. Objetivos del proyecto

Los objetivos principales del proyecto son los siguientes:

- **Estudio de la migración de máquinas virtuales mediante el hipervisor *Xen*.**

  En este objetivo se pretende realizar un estudio de la virtualización de máquinas virtuales utilizando el hipervisor *Xen*, en concreto se debe estudiar con detalle la posibilidad de migrar máquinas virtuales en vivo a otros equipos remotos. La migración en vivo permite migrar una máquina virtual a otro nodo remoto sin ser apagada ni siquiera pausada, consiguiendo de esta forma que la migración entre nodos sea transparente al usuario final, manteniendo un servicio sin interrupción. Para ello, se debe realizar una preparación y configuración de los equipos que intervengan en dicho proceso de migración.

  En resumen, se deben realizar tareas de estudio, preparación, instalación y configuración de todos los elementos que intervengan en el proceso de migración de máquinas virtuales.

- **Implementación de una estrategia de ahorro de energía basada en la carga de trabajo**

  Para sacar partido a la migración en vivo de máquinas virtuales se pretende desarrollar una estrategia de ahorro de energía. Un *frontend* u

ordenador principal será el encargado de realizar un arbitraje de las máquinas virtuales que estén albergadas en los diferentes nodos del *cluster* o conjunto de ordenadores. Cuando un nodo tenga mucha carga de trabajo, el *frontend* se encargará de migrar alguna de las máquinas virtuales que contenga a otro nodo disponible. Por otra parte, si algún nodo tiene muy poca carga, el *frontend* migrará sus máquinas virtuales a otros nodos, apagando posteriormente dicho nodo.

1.4. Entorno y estado inicial

Para la realización de este proyecto se han proporcionado una serie de recursos y un espacio de trabajo descritos a continuación:

- **Laboratorio de investigación TI2127DL**

  En este laboratorio existe el entorno adecuado para la realización del proyecto. Está estructurado en tres espacios de trabajo, uno para realizar reuniones, otro donde están todos los equipos y *clusters* para la investigación, y otro para la realización de proyectos en el que hay todo el material de oficina y otras herramientas necesarias para el desarrollo de proyectos.

- ***Clúster* de computadores**

  La plataforma hardware sobre la que se debe desarrollar el proyecto es un *cluster* de computadores. En este caso, se han proporcionado tres equipos que se configurarán de forma que uno de ellos sea el *frontend*, y los otros dos formarán parte de éste como nodos del sistema.

- **El sistema operativo Ubuntu Server 12.04 LTS**

  El sistema operativo que se ha utilizado para el desarrollo del proyecto ha sido la distribución de linux Ubuntu 12.04 LTS. Se ha elegido esta distribución ya que ha apostado recientemente por el Cloud Computing y facilita mucho la instalación de todos los componentes de virtualización necesarios para su despliegue.

- **Encendido remoto (Wake On LAN)**

  Todos los nodos del *cluster* soportan y tienen configurada su BIOS para el encendido remoto Wake on LAN [7]. El funcionamiento de esta tecnología es el siguiente: la tarjeta de red se configura en la BIOS para que, aunque el nodo esté apagado, ésta quede en *stand-by*, a la espera de "paquetes mágicos". Un "paquete mágico" es una trama *broadcast* Ethernet en la que se incluye la dirección MAC del nodo que se desea encender, y es enviada por el equipo que quiere realizar el encendido remoto (en este caso sería el *frontend*). Cuando la tarjeta de red recibe una trama

de este tipo y, además, su dirección MAC coincide con la del paquete, la tarjeta envía una señal eléctrica a través del cable Wake on LAN, que llega a la placa base del nodo, encendiendo dicho nodo.

Esta característica del entorno es necesaria en el proyecto para realizar el encendido automático y remoto de los nodos del *cluster*.

- **Documentación del hipervisor Xen**

  Se dispone de documentación inicial sobre el hipervisor *Xen*, en la que existe información sobre la instalación y configuración de *Xen*, así como de manuales de usuario y otro tipo de documentación adicional.

En resumen, en la configuración inicial se parte de tres equipos independientes, en los que se instalará un sistema operativo Linux con soporte para virtualización en *Xen*.

1.5. Organización de la memoria

Esta memoria técnica está organizada en los siguientes capítulos:

- **Planificación y evaluación de recursos**

  En este capítulo se realiza toda la planificación del proyecto. En cada sección del capítulo se muestra con detalle tanto la planificación temporal del proyecto, como las estimaciones de recursos y estimaciones de costes del proyecto. Por tanto, se hace un análisis a priori de todo lo necesario a tener en cuenta para desarrollar el proyecto.

- **Desarrollo y resultados del proyecto**

  En este capítulo se realiza una descripción técnica de todas las partes del desarrollo del proyecto, mostrando en cada caso los resultados obtenidos y su aplicación práctica. Esta parte es la más extensa de la memoria.

- **Desviaciones del proyecto**

  En este capítulo se analizan y describen las desviaciones que ha tenido el proyecto durante su desarrollo. En él, se muestran detalles como los costes reales aproximados del proyecto y un diagrama de tiempos indicando la duración real del proyecto.

- **Conclusiones**

  En este capítulo se hace una recopilación de las conclusiones resultantes de la realización del proyecto, tratando de enfatizar los aspectos más relevantes, los conceptos aprendidos así como las conclusiones generales y personales obtenidas una vez finalizado el mismo.

- **Anexos**

  Por último, al final de la memoria se adjuntan una serie de anexos con información adicional sobre el desarrollo del proyecto.

CAPÍTULO 2

Planificación y evaluación de recursos

Índice

2.1. Planificación temporal **7**
2.2. Estimación de recursos **14**
2.3. Estimación de costes **17**

En este capítulo se muestran las tareas que se deben llevar a cabo para el desarrollo del proyecto, se definen las dependencias entre ellas y se estiman sus duraciones. Por lo tanto, se muestra una visión general de la duración del proyecto.

Por otro lado, en este capítulo se realiza un análisis de los recursos necesarios para el desarrollo del proyecto y una estimación de los costes del mismo.

2.1. Planificación temporal

2.1.1. Identificación de tareas

En la figura 2.1 se muestra un listado de las tareas previstas que se deben llevar a cabo en el proyecto. Se muestra en forma de árbol, de modo que las tareas resumen engloban a las tareas hojas (las que no se pueden descomponer en más subtareas).

A continuación se presenta una breve explicación de cada una de las tareas:

- **1: Iniciación**

  Esta tarea engloba todas las tareas de iniciación del proyecto, es decir, la familiarización con el entorno y la documentación de herramientas.

- **1.1: Familiarización con el entorno**

  En esta tarea se realizará un primer estudio y documentación de todos los conceptos necesarios para el proyecto: virtualización, cloud computing, hipervisor Xen, *clusters* de computadores, encendido y apagado remoto de equipos, así como otros conceptos de interés.

- **1.2: Documentación de herramientas**

  Durante esta tarea se realizará una recopilación de las herramientas más relevantes de todos los conceptos mencionados en la tarea anterior. Además, se realizará una primera visión general sobre la estructura y diseño del proyecto a realizar.

- **2: Análisis**

  Esta tarea es una tarea resumen que engloba todas las tareas de análisis del proyecto. En estas tareas se analizan los recursos necesarios para llevar a cabo el proyecto así como los requisitos mínimos que debe cumplir para su finalización.

- **2.1: Análisis de los recursos necesarios**

  En esta tarea se realiza un análisis de los recursos necesarios para la realización del proyecto. Primero se realiza un estudio de los recursos necesarios para posteriormente realizar la petición de dichos recursos.

- **2.2: Análisis de requisitos**

  En esta tarea se realizarán las entrevistas y reuniones oportunas con el tutor para concretar todas las características, requisitos y funcionamiento del proyecto. Esta tarea se repetirá tantas veces como sea necesario hasta concretar claramente todas las funcionalidades y aspectos clave del proyecto.

- **3: Desarrollo**

  Esta tarea engloba todas las tareas correspondientes a la fase de desarrollo del proyecto.

- **3.1: Documentación sobre Cloud Computing, virtualización, Xen y Libvirt**

  Un primer paso para el desarrollo del proyecto es la documentación de todos los conceptos relacionados con el proyecto. Por tanto, esta tarea consistirá en adquirir los conocimientos necesarios sobre Cloud Computing, virtualización, el hipervisor de máquina virtuales Xen y la librería de virtualización Libvirt.

ID.	Nombre de la tarea
0	**Estrategia de ahorro de energía**
1	**Iniciación**
1.1	Familiarización con el entorno
1.2	Documentación de herramientas
2	**Análisis**
2.1	Análisis de los recursos necesarios
2.2	Análisis de requisitos
3	**Desarrollo**
3.1	Documentación sobre Cloud Computing, virtualización, Xen y Libvirt
3.2	**Preparación del sistemas**
3.2.1	Instalación del Sistema Operativo con Xen
3.2.2	Configuración de la red interna
3.3	Gestión básica de máquinas virtuales con Xen
3.4	**Migración en vivo de máquinas virtuales**
3.4.1	Configuración de Xen
3.4.2	Configuración SSH entre equipos
3.4.3	Configuración del sistema de archivos NFS
3.5	Estudio de la librería de virtualización Libvirt
3.6	Algoritmo para la estrategia de ahorro de energía
3.7	**Implementación de la estrategia (Nodos)**
3.7.1	Gestión de ficheros de configuración de los nodos
3.7.2	Programación de librerías intermedias de los nodos
3.7.3	Programación del protocolo de comunicación frontend/nodo
3.7.4	Programación del demonio servidor de datos
3.7.5	Programación del demonio de estadísticas de CPU
3.8	**Implementación de la estrategia (Frontend)**
3.8.1	Gestión de ficheros de configuración del frontend
3.8.2	Programación de librerías intermedias del frontend
3.8.3	Programación del demonio de arbitraje
3.9	Pruebas del sistema
3.10	Estudio teórico del ahorro energético obtenido
4	**Documentación del proyecto**
4.1	Redacción de la memoria
4.2	Preparación de la presentación
4.3	Presentación del proyecto

Figura 2.1: Identificación de tareas.

- **3.2: Preparación del sistema**

  Una vez comprendidos los nuevos conceptos, se deberá preparar el sistema para desarrollar el proyecto. En esta tarea se preparará y configurará la plataforma hardware sobre la que se desarrollará el proyecto. Es decir, se prepararán los distintos elementos que compondrán el *cluster*.

- **3.2.1: Instalación del Sistema Operativo con Xen**

  En esta tarea se determinará el sistema operativo adecuado para el funcionamiento del *cluster* con el hipervisor Xen y se realizará dicha instalación en cada uno de los equipos.

- **3.2.2: Configuración de la red interna**

  En esta tarea se configurará la red interna de los equipos que forman el *cluster*. Cada equipo dispondrá de dos tarjetas de red, una para la red interna entre nodos y otra para la red interna entre máquinas virtuales. El acceso al *frontend*, se realizará a través de un host llamado *neon*, al que se accederá a través de la dirección *neon.act.uji.es*.

- **3.3: Gestión básica de máquinas virtuales con Xen**

  En esta tarea se realizarán distintas pruebas de funcionamiento del hipervisor de máquinas virtuales Xen. Las pruebas consistirán en la creación de nuevas máquinas virtuales, probar las distintas configuraciones que ofrece y realizar una gestión básica de dichas máquinas.

- **3.4: Migración en vivo con Xen**

  Esta tarea engloba las tareas de estudio y configuración de Xen, así como de otros elementos necesarios para realizar la migración en vivo de máquinas virtuales.

- **3.4.1: Configuración de Xen**

  Para poder realizar la migración de máquinas virtuales en vivo con Xen, se deberán configurar algunas de las características de Xen para permitir esta posibilidad, es decir, se deberá estudiar los ficheros de configuración de Xen que se deben modificar.

- **3.4.2: Configuración de SSH entre equipos**

  Esta tarea consiste en realizar la configuración adecuada de la conexión remota con SSH, para que al conectarse remotamente con los nodos no sea necesario autenticar a los equipos de forma manual, es decir, sin necesidad de introducir una contraseña para conectarse entre ellos. Esto se consigue mediante la creación de una llave pública y privada para cada nodo de forma que la llave pública se entrega a cada nodo y la privada sirve para autenticar la llave pública.

- **3.4.3: Configuración del sistema de ficheros NFS**

  Una de las tareas a realizar para poder conseguir la migración en vivo de máquinas virtuales es la creación de un sistema compartido de ficheros entre los equipos implicados en la migración. De esta forma, cuando se migre una máquina, el nuevo nodo que contenga la máquina virtual sea capaz de acceder al disco duro asociado a dicha máquina virtual.

  En esta tarea se creará un sistema compartido de fichero NFS (Network File System) entre todos los nodos que comparten la red interna. De esta forma, se consigue que el directorio que se comparta sea accesible desde cualquier nodo de la red.

- **3.5: Estudio de la librería de virtualización Libvirt**

  La librería Libvirt es la candidata perfecta para la programación del algoritmo de ahorro de energía. Esta librería permite manejar distintos hipervisores entre los cuales se encuentra Xen, por lo tanto, en esta tarea se realizará un estudio de esta librería de virtualización para comprender su funcionamiento y posibilidades.

- **3.6: Algoritmo para la estrategia de ahorro de energía**

  En esta tarea se describirá el algoritmo que se utilizará para realizar la estrategia de ahorro de energía.

- **3.7: Implementación de la estrategia (Nodos)**

  Esta tarea engloba las tareas de programación de los demonios, librerías y protocolos necesarios para los nodos del *cluster*.

- **3.7.1: Gestión de ficheros de configuración de los nodos**

  Esta tarea consiste en crear la estructura adecuada para los ficheros de configuración de los nodos y los ficheros de registro necesarios para controlar el estado de los nodos en el *frontend*. Este fichero de registro guardará un registro periódico del estado de la CPU del nodo donde se esté ejecutando.

  Por lo tanto, esta tarea consiste en la creación de la estructura de dichos ficheros, la información a manejar en los mismos, los directorios donde se almacenarán, etc. Por otra parte, también se creará una función para comprobar que los ficheros de configuración siguen la estructura adecuada.

- **3.7.2: Programación de librerías intermedias de los nodos**

  Para obtener algunos datos sobre los nodos, para su posterior utilización en los demonios, es necesario crear una serie de librerías que obtengan dichos datos. Algunos de los datos necesarios para los demonios son la

carga de CPU en un nodo y el número de dominios (máquinas virtuales) que contenga un nodo.

Por lo tanto, esta tarea consiste en la creación de aquellas funciones necesarias para gestionar el estado de un nodo.

- **3.7.3: Programación del protocolo de comunicación frontend/nodo**

  Debe existir una comunicación entre el *frontend* y los nodos para que el *frontend* pueda recabar la información de estado de los nodos. Por ello, se debe definir el protocolo de comunicación (a nivel de aplicación) que asegure el correcto intercambio de información.

- **3.7.4: Programación del demonio servidor de datos**

  Esta tarea consiste en desarrollar un demonio en los nodos que se encargue de mantener un servicio de comunicación con el *frontend*, de manera que le informe del estado actual del nodo. De esta forma, cuando el *frontend* necesite información sobre el estado actual de nodo, este demonio se encargará de proporcionarle dicha información utilizando para ello las librerías desarrolladas en la tarea 3.8.2.

- **3.7.5: Programación del demonio de estadísticas de CPU**

  Este demonio será el encargado de guardar estadísticas sobre la CPU de cada nodo. Dichas estadísticas se guardarán de forma periódica en un fichero de registro, utilizando para ello las librerías desarrolladas en la tarea 3.7.2.

- **3.8: Implementación de la estrategia (Frontend)**

  Esta tarea engloba las tareas de programación de los demonios, librerías y ficheros de configuración para el *frontend* del *cluster*.

- **3.8.1: Gestión de ficheros de configuración del frontend**

  El demonio de arbitraje del *frontend* necesitará unos parámetros de entrada introducidos por el usuario, como por ejemplo, el número de nodos conectados, la dirección IP de cada nodo, las máquinas virtuales creadas, etc. Por tanto, para introducir estos parámetros al demonio de arbitraje del *frontend* es necesario crear un fichero de configuración donde, siguiendo una estructura determinada, el usuario podrá introducir dichos parámetros.

  Esta tarea consiste en establecer la estructura de los ficheros de configuración para el demonio del *frontend* y crear una función que compruebe que dicho fichero sigue la estructura adecuada.

- **3.8.2: Programación de las librerías intermedias del frontend**

En esta tarea se crearán las librerías intermedias necesarias en el *frontend*, para su posterior utilización en la programación del demonio de arbitraje del *frontend*.

- **3.8.3: Programación del demonio de arbitraje**

  Esta tarea es una de las más importantes de todas, ya que consiste en implementar el algoritmo descrito en la tarea 3.6. Éste será el encargado de realizar el arbitraje de los nodos y las máquinas virtuales, así como el responsable de encender y apagar remotamente los nodos.

  Esta tarea consiste en la creación de un demonio que, a través de los ficheros de configuración y las librerías intermedias desarrolladas en el *frontend*, ejecute un algoritmo de arbitraje para conseguir un ahorro de energía en los nodos, aprovechando sus recursos de una forma más eficiente, utilizando la migración pertinente de las máquinas virtuales creadas.

- **3.9: Pruebas del sistema**

  En esta tarea se deben realizar distintas pruebas en las que se observe el funcionamiento de todo el sistema, corrigiendo los errores encontrados y estableciendo los parámetros que mejor se ajusten al funcionamiento esperado.

- **3.10: Estudio teórico del ahorro energético obtenido**

  Esta tarea consiste en estudiar de forma teórica el posible ahorro energético que se obtendría utilizando la estrategia implementada en este proyecto.

- **4: Documentación del proyecto**

  Esta tarea engloba las tareas de documentación del desarrollo del proyecto. Éstas, se corresponden con las tareas de redacción de la memoria, preparación de la presentación y presentación final del proyecto ante un tribunal.

- **4.1: Redacción de la memoria**

  En esta tarea se redactará la memoria del proyecto, en la que se detallarán los objetivos, los detalles de implementación, los problemas encontrados, las soluciones, las decisiones tomadas, la planificación, los anexos, etc.

- **4.2: Preparación de la presentación**

  En esta tarea se establecerá un índice a seguir en la presentación del proyecto y se prepararán todas las transparencias de las que se hará uso en la misma.

- **4.3: Presentación del proyecto**

  En esta última tarea se realizará una presentación del proyecto ante un tribunal para exponer todo el trabajo realizado y dar a conocer los principales resultados del proyecto desarrollado.

2.1.2. Estimación de la duración de las tareas y relaciones de precedencia

En la figura 2.2 se muestra la duración de cada una de las tareas así como sus relaciones de precedencia.

Hay algunas tareas que tienen un tipo de precedencia obligatoria, pero otras se pueden realizar en paralelo. Pero, dado que este proyecto solo tiene asignado un desarrollador, todas las tareas se realizarán de forma secuencial.

Se puede observar que la duración total estimada del proyecto es de 60 días de trabajo correspondientes a los 300 créditos destinados a la asignatura. Se estima una media de trabajo de 5 horas diarias (de 9:00 a 12:00 y de 18:00 a 20:00).

2.1.3. Diagrama de planificación temporal

En la figura 2.3 se muestra el resultado del diagrama de Gantt de planificación a llevar a cabo en el proyecto.

La fecha estimada del comienzo del proyecto está prevista para el día **16 de Abril de 2012**. Y la fecha prevista para terminar el proyecto está prevista para el día **6 de Julio de 2012**.

2.2. Estimación de recursos

En esta sección se presentan los recursos que se estima que van a ser necesarios para el desarrollo del proyecto. Esta estimación de recursos se lleva a cabo partiendo de los requisitos y objetivos acordados.

Recursos hardware:

- Tres equipos completos con las características mínimas para utilizar la virtualización. Uno de ellos realizará la función de *frontend* y el resto, nodos de *clúster*.
- Los tres equipos deben disponer de dos tarjetas de red para realizar dos redes internas Las tarjetas de red deben disponer de la conexión *Wake On LAN* para realizar el encendido remoto de los nodos.

ID.	Nombre de la tarea	Duración	Predecesoras
0	**Estrategia de ahorro de energía**	**60 días**	
1	**Iniciación**	**5 días**	
1.1	Familiarización con el entorno	2días	
1.2	Documentación de herramientas	3 días	
2	**Análisis**	**5 días**	
2.1	Análisis de los recursos necesarios	2 días	
2.2	Análisis de requisitos	3 días	
3	**Desarrollo**	**35 días**	
3.1	Documentación sobre Cloud Computing, virtualización, Xen y Libvirt	2 días	
3.2	**Preparación del sistemas**	**3 días**	
3.2.1	Instalación del Sistema Operativo con Xen	2 días	
3.2.2	Configuración de la red interna	1 días	3.2.1
3.3	Gestión básica de máquinas virtuales con Xen	2 días	3.2.1
3.4	**Migración en vivo de máquinas virtuales**	**5 días**	
3.4.1	Configuración de Xen	3 días	3.2.1
3.4.2	Configuración SSH entre equipos	1 días	3.2.1
3.4.3	Configuración del sistema de archivos NFS	1 días	3.2.1
3.5	Estudio de la librería de virtualización Libvirt	3 días	3.2.1
3.6	Algoritmo para la estrategia de ahorro de energía	3 días	3.5
3.7	**Implementación de la estrategia (Nodos)**	**7 días**	
3.7.1	Gestión de ficheros de configuración de los nodos	1 días	3.6
3.7.2	Programación de librerías intermedias de los nodos	2 días	3.7.1
3.7.3	Programación del protocolo de comunicación frontend/nodo	1 días	3.7.2
3.7.4	Programación del demonio servidor de datos	1 días	3.7.3
3.7.5	Programación del demonio de estadísticas de CPU	2 días	3.7.3
3.8	**Implementación de la estrategia (Frontend)**	**7 días**	
3.8.1	Gestión de ficheros de configuración del frontend	2 días	3.6
3.8.2	Programación de librerías intermedias del frontend	2 días	3.8.1
3.8.3	Programación del demonio de arbitraje	3 días	3.8.2
3.9	Pruebas del sistema	1 días	3.8.3
3.10	Estudio teórico del ahorro energético obtenido	2 días	3.9
4	**Documentación del proyecto**	**15 días**	
4.1	Redacción de la memoria	12 días	3.10
4.2	Preparación de la presentación	2 días	4.1
4.3	Presentación del proyecto	1 días	4.2

Figura 2.2: Duración y precedencias de las tareas.

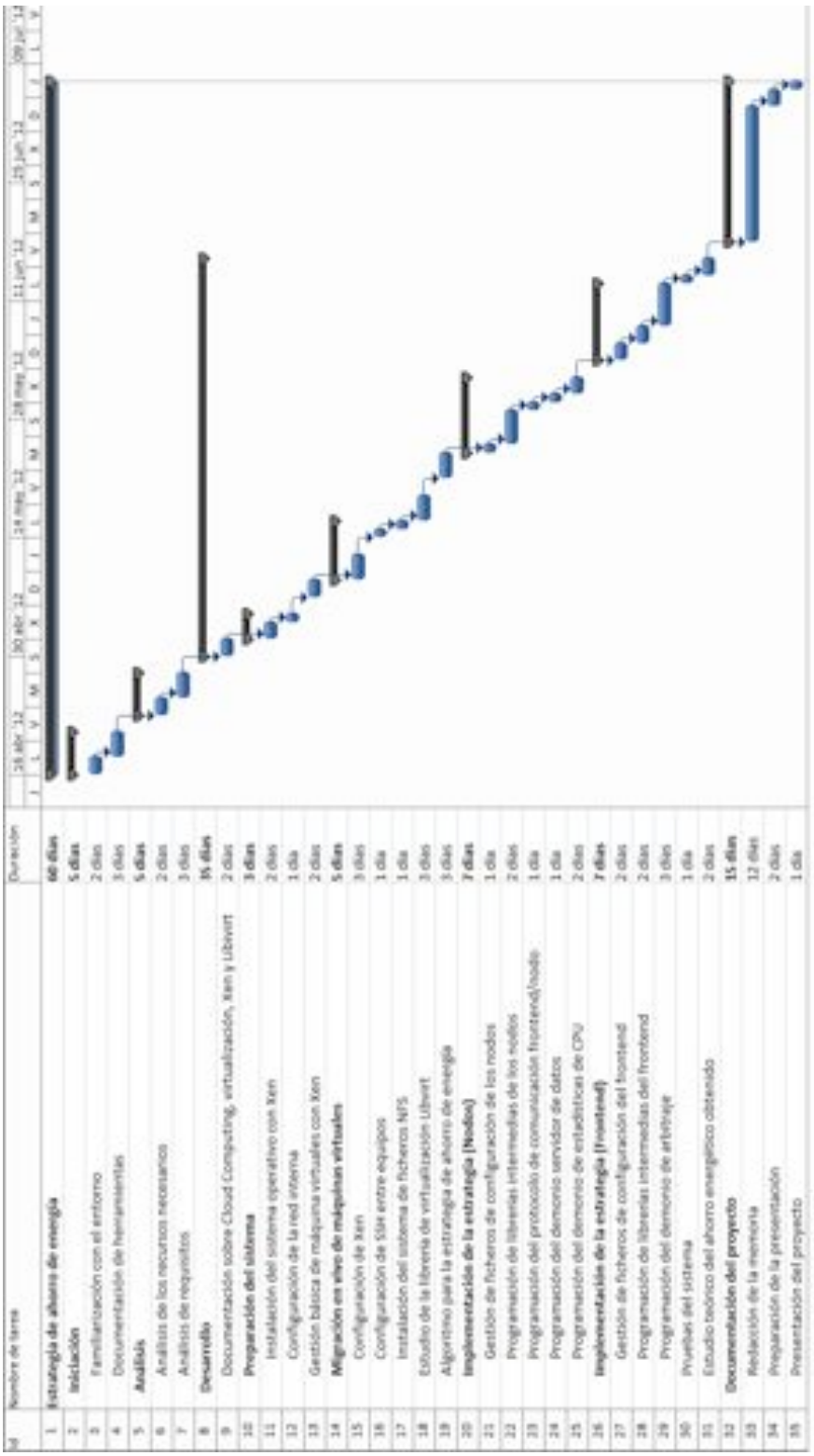


Figura 2.3: Diagrama de Gantt.

- Dos *switches* para montar las redes internas del *clúster*.
- Conexión a Internet que se configurará, en este caso, a través de la red de la universidad para poder acceder al *clúster* de forma remota.
- Un equipo externo al *clúster* para realizar toda la programación y configuración del *clúster* de forma remota.
- Un monitor, teclado y ratón para realizar la instalación y configuración del *clúster*.

Recursos software:

- Microsoft Project 2010 para realizar la planificación.
- Sistema Operativo Ubuntu 12.04 LTS.
- Librerías y software de *Xen*.
- Librería Libvirt para la programación de *Xen*.
- Librerías de Python.
- Sistema de composición de textos libre, LaTeX.

Recursos humanos:

- Un desarrollador supervisado por el tutor del proyecto.

2.3. Estimación de costes

Todos los recursos hardware requeridos son proporcionados por grupo de investigación HPC&A de la universidad, por lo tanto no hay que realizar ninguna inversión en este caso. Por otro lado, la licencia del Microsoft Project 2010 ha sido proporcionada por la universidad como licencia de estudiante. Los otros recursos software son de carácter libre por lo tanto tampoco suponen ningún coste. Teniendo en cuenta estos criterios, solo se contabilizarán los costes del trabajo realizado por el desarrollador en todas las fases de desarrollo del proyecto.

Suponiendo que el coste por hora de un desarrollador informático es de 15€/hora y teniendo en cuenta que la estimación temporal total en horas es de 300, el coste total estimado es el siguiente:

$$300\ \textit{horas}\ \text{x}\ 15€/\textit{hora}\ \text{x}\ 1\ \textit{desarrollador} = \mathbf{4.500\ €}$$

CAPÍTULO 3

Desarrollo y resultados del proyecto

Índice

3.1. Virtualización **19**
3.2. Cloud Computing **27**
3.3. Documentación de Xen **31**
3.4. Preparación del *clúster* **34**
3.5. Gestión de máquinas virtuales **37**
3.6. Configuración para la migración con *Xen* **40**
3.7. Programación de las librerías intermedias **43**
3.8. Demonios para los nodos **48**
3.9. Demonio para el *frontend* **51**
3.10. Ficheros de configuración **54**
3.11. Arquitectura del sistema **55**

En este capítulo se presenta el desarrollo del proyecto realizado. En primer lugar se realiza una breve descripción de algunos conceptos aprendidos sobre la virtualización y Cloud Computing. A continuación, se muestra todo el trabajo realizado para alcanzar con los objetivos fijados.

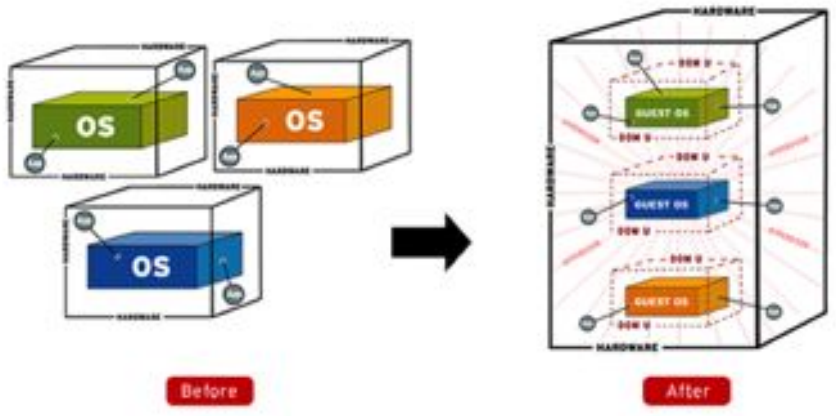
3.1. Virtualización

En esta primera sección se presenta una breve descripción del concepto de virtualización y sus tipos.

En este proyecto se habla de Cloud Computing como tema principal del proyecto, pero no se puede hablar de Cloud Computing sin describir primero

la base tecnológica en la que se apoya, es decir, la virtualización.

Virtualización se refiere a la implementación software de un recurso físico real en un recurso virtual. En computación moderna, se denomina virtualización a toda abstracción de un recurso computacional.



La virtualización ofrece una serie de beneficios de gran interés, aplicables a la mayoría de empresas e instituciones. Entre los beneficios cabe destacar el aprovechamiento de recursos hardware que permite esta tecnología; la reducción del número de servidores activos manteniendo la misma calidad de servicio, es decir, ahorro considerable de energía; reducción de costos derivados del aprovechamiento de recursos; flexibilidad de uso, ya que permite arrancar y apagar tantas máquinas virtuales como sean necesarias; recuperación rápida ante desastres, ya que, en caso de que algún servidor se estropeara, se podría iniciar una nueva máquina virtual en otro servidor con el mismo servicio, recuperando este servicio en pocos minutos.

3.1.1. Tipos de vitualización

Existen dos tipos de virtualización:

- **Completa**
  Se virtualiza un computador en su integridad, obteniéndose una máquina virtual. La virtualización completa es la implementación mediante software de una máquina virtual, capaz de ejecutar programas del mismo modo que lo haría una maquina física.

- **Parcial**
  Se virtualizan sólo determinados componentes, obteniéndose un recurso virtual (memoria, almacenamiento, red, etc).

La capa de software que controla las máquinas virtuales se llama monitor de máquinas virtuales (VMM) o hipervisor. El hipervisor se encarga de manejar, gestionar y arbitrar todos los recursos de hardware tanto de la máquina física como de las máquinas virtuales.

3.1.2. Monitor de máquinas virtuales o Hipervisor

Existen dos tipos de hipervisores según su ubicación en las capas del sistema:

- **Tipo I. Nativos**
  Estos hipervisores se ejecutan directamente sobre el hardware, pudiendo tener acceso directo a todos los recursos físicos del computador. En la figura 3.1 se muestra su esquema.

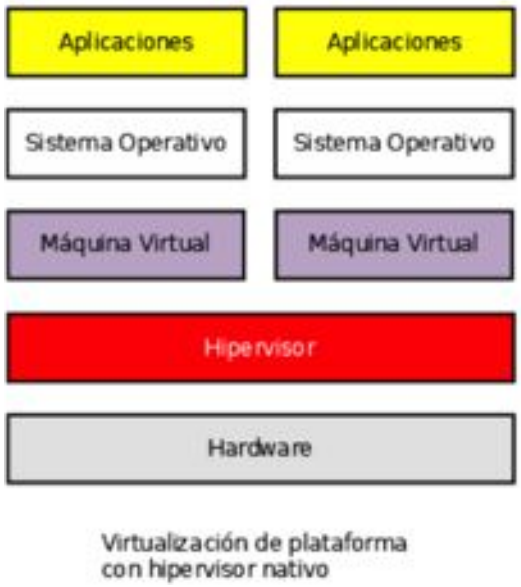


Figura 3.1: Esquema hipervisor Tipo I. Nativo

  Algunos ejemplos de hipervisores nativos son: *VMware ESX, Microsoft Hyper V, Xen, Parallels Server, KVM, etc.*

- **Tipo II. Aplicaciones hipervisoras**
  Estos hipervisores se instalan sobre un sistema operativo anfitrión con lo que deben solicitar los recursos hardware a través del sistema operativo. Este tipo de hipervisor suele ser menos eficiente ya que debe competir con los recursos hardware que utiliza el propio sistema operativo. En la figura 3.2 se muestra su esquema.

  Algunos ejemplos de aplicaciones hipervisoras son: *VMware Server, VMware Fusion, QEMU, Microsoft Virtual PC, VirtualBox, etc.*

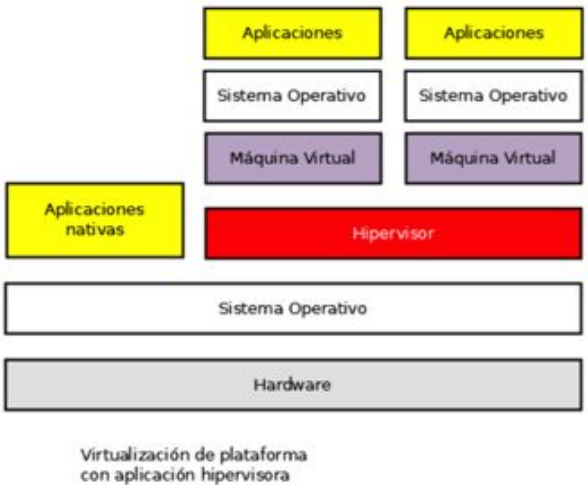


Figura 3.2: Esquema hipervisor Tipo II. Aplicación hipervisora

3.1.3. Virtualización completa

La virtualización completa es la implementación, mediante software, de una máquina virtual, capaz de ejecutar programas del mismo modo que lo haría un máquina física. Aunque la virtualización suele replicar con extrema precisión el comportamiento de una máquina física, la acepción del término actualmente incluye a máquinas virtuales que no tienen ninguna equivalencia directa con hardware real.

Como desarrollo, la virtualización nace en los años 60, cuando IBM implementa una tecnología para particionar sus grandes *mainframes* en pequeñas máquinas virtuales aisladas. Máquinas que, al arrancar concurrentemente, permitieron a los *mainframes* anfitriones alcanzar la multitarea a efectos prácticos: cada una de ellas ejecutaba distintas aplicaciones y procesos. En los años 80 y los 90, los *mainframes* fueron siendo barridos del mercado en favor de los nuevos paradigmas (computación distribuida, arquitecturas x86, arquitecturas cliente/servidor, etc). Actualmente, nos encontramos ante un parque de servidores potentes y caros de mantener, cuya ocupación de recursos computacionales no pasa del 15 % de los disponibles. Volvemos a tener máquinas físicas infrautilizadas y rígidas que bien se podrían convertir en un conjunto de máquinas virtuales capaces de aprovechar mejor el inmovilizado material.

Existen dos tipos de virtualización completa:

- **Virtualización de plataforma**
  La máquina virtual ejecuta directamente un sistema operativo no modificado.

- **Procesamiento virtual**
  La máquina virtual queda instalada sobre el sistema operativo anfitrión, donde ejecutará determinadas aplicaciones, que se ejecutarán en un entorno de procesamiento virtual.

Virtualización de plataforma

En 1974, Popek y Golberg ya establecieron en su artículo [13] un conjunto de condiciones suficientes para que una arquitectura de computadores soportara eficientemente la virtualización.

- **Equivalencia.** Todo programa ejecutándose bajo el hipervisor tiene que tener un comportamiento idéntico al de su homólogo ejecutándose directamente en el hardware.
- **Control de recursos.** El hipervisor debe tener control completo sobre los recursos virtualizados.
- **Eficiencia.** Una fracción representativa de las instrucciones máquina deben ser ejecutadas sin la intervención del hipervisor.

Popek y Goldberg clasificaron, además, las instrucciones máquina **conflictivas** para la virtualización en tres grupos.

- Instrucciones privilegiadas
- Instrucciones sensibles por control
- Instrucciones sensibles por comportamiento

Existirán, por tanto, arquitecturas que por su diseño (incluyen instrucciones conflictivas) **no soportarán** la virtualización de plataforma según Popek & Goldberg, x86 es una de ellas. Algunas técnicas que se han desarrollado para solucionar este problema son las siguientes:

- Traducción binaria.
- Añadir circuitería especial al hardware para ejecutar directamente las instrucciones conflictivas (AMD Secure Virtual Machine, Intel Virtualization Technology).
- Utilizar una técnica software llamada **paravirtualización**, donde se modifica el sistema operativo huésped para que algunas instrucciones conflictivas se ejecuten directamente en el hardware.

Actualmente existen diferentes hipervisores, como por ejemplo VirtualBox, VMware, Xen, Parallels Desktop, Hyper-V, etc. En este proyecto se ha elegido Xen porque cuenta con todas las características necesarias para el desarrollo de este proyecto, como por ejemplo la migración en vivo, además de ser de carácter libre.

Procesamiento virtual

Una máquina de procesamiento virtual es una aplicación que se ejecuta sobre un SO anfitrión para ofrecer un entorno de ejecución virtual a terceras aplicaciones. Cuando la máquina de procesamiento virtual emule una arquitectura no nativa o tenga implementaciones en diferentes arquitecturas, las aplicaciones que se ejecutarán sobre la máquina de procesamiento virtual serán independientes de la plataforma. Algunas máquinas de procesamiento virtual, emulan un sistema no nativo, que sólo existe como una especificación y no como hardware real. Ejemplos:

- Máquina virtual Java
- Common Language Runtime

Otras máquinas de procesamiento virtual hacen emulación de sistemas reales:

- Wine

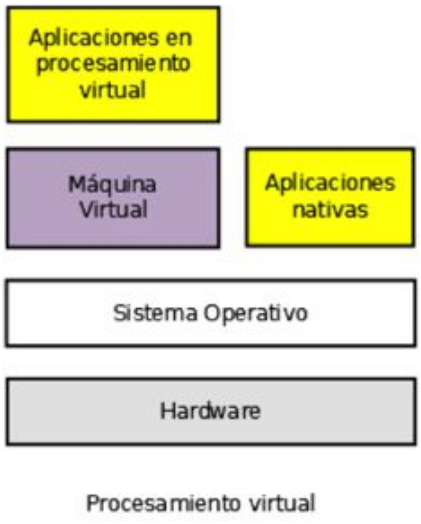


Figura 3.3: Esquema de procesamiento virtual

Bajo la denominación de procesamiento virtual se agrupa toda una colección de tecnologías para el encapsulamiento y disociación de las aplicaciones respecto a su entorno de ejecución. Se persigue desconectar las aplicaciones de los sistemas operativos al proveer un entorno artificial que garantice la portabilidad y compatibilidad de las aplicaciones, que se limitarán a interactuar con un subconjunto controlado de recursos del sistema.

3.1.4. Virtualización Parcial

Dentro de la virtualización parcial, los recursos más habituales para virtualizar en los sistemas operativos modernos son:

- Memoria virtual
- Almacenamiento virtual
- Redes virtuales
- Escritorios virtuales

Memoria Virtual (Memoria de intercambio o Swap)

Permiten asignar a las aplicaciones amplios rangos de memoria de trabajo contiguos que en la realidad son bloques fragmentados y, si es necesario, residentes en almacenamiento secundario.

Almacenamiento virtual

El almacenamiento virtual abstrae la localización física de los datos, presentando un espacio lógico de almacenaje que luego se mapeará a emplazamientos físicos reales. Esto permite obtener independencia sobre la ubicación de los datos, flexibilizando la asignación de los recursos de almacenaje; lo cual conllevará múltiples beneficios. Ejemplos: RAID o LVM.

Redes virtuales

NAT (Network Address Translation) es un mecanismo que permite intercambiar paquetes de datos entre redes a base de traducir las direcciones de red. Suelen implementarlo los enrutadores en el caso de las redes físicas. En el caso (redes virtuales), será el hipervisor el encargado de implementar NAT.

Escritorios virtuales

La virtualización de escritorio es un término relativamente nuevo, introducido en la década de los 90, que describe el proceso de separación entre el

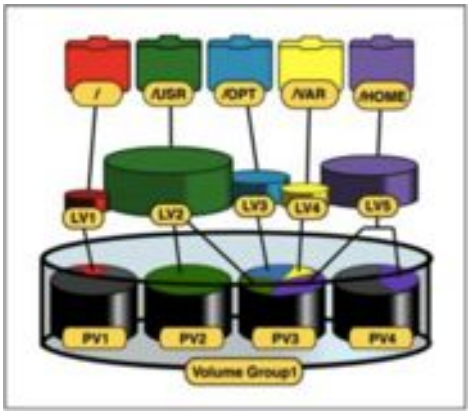


Figura 3.4: Logical Volume Management

escritorio, que engloba los datos y programas que utilizan los usuarios para trabajar, de la máquina física. El escritorio 'virtualizado' es almacenado remotamente en un servidor central en lugar de en el disco duro del ordenador personal. Esto significa que cuando los usuarios trabajan en su escritorio desde su portátil u ordenador personal, todos sus programas, aplicaciones, procesos y datos se almacenan y ejecutan centralmente, permitiendo a los usuarios acceder remotamente a sus escritorios desde cualquier dispositivo capaz de conectarse remotamente al escritorio, tales como un portátil, PC, smartphone o cliente ligero.

Figura 3.5: Escritorio virtual. EyeOs

3.2. Cloud Computing

La computación en la nube, concepto conocido también bajo los términos servicios en la nube, informática en la nube, nube de cómputo o nube de conceptos, del inglés cloud computing, es un paradigma que permite ofrecer servicios de computación a través de Internet.

La tendencia actual del mercado sobre la oferta de cloud computing muestra que es un servicio de futuro. Esta tecnología permite ofrecer una serie de servicios a través de Internet sin tener conocimiento alguno sobre la ubicación y la gestión de dichos servicios. Es un nuevo modelo de prestación de servicios de negocio y tecnología, que permite al usuario acceder a un catálogo de servicios estandarizados y responder a las necesidades de su negocio, de forma flexible y pagando únicamente por el consumo efectuado. Una de las principales ventajas que supone la utilización de cloud computing es el ahorro de los costes a medio y largo plazo. Comparado con la informática tradicional, se aprovecha mejor el potencial del hardware en relación a su valor.

El cloud computing es una plataforma altamente escalable que permite un acceso rápido al recurso hardware o software y donde el usuario no necesita ser experto para su manejo y acceso. Las nubes suelen apoyarse en tecnologías como la virtualización, técnicas de programación como el multitenancy (permite que una misma ejecución de una aplicación dé servicio a varios clientes) y/o habilidades para la escalabilidad, balanceo de carga y rendimiento óptimo, para conseguir ofrecer el recurso de una manera rápida y sencilla. Además estas técnicas generan economías de escala derivadas del aprovechamiento eficiente de los recursos hardware y humanos que terminan repercutiendo en el precio que paga el cliente.

3.2.1. Modelos de servicio

El Cloud Computing lo podemos dividir en tres niveles en función de los servicios que ofrece.

El software como servicio (software as a service, SaaS) se encuentra en la capa más alta y caracteriza una aplicación completa ofrecida como un servicio, bajo demanda, multitenancy, que significa que una sola instancia del software que se ejecuta en la infraestructura del proveedor y sirve a múltiples organizaciones de clientes. Un ejemplo de SaaS sería Google Apps que ofrece servicios básicos de negocio como el e-mail.

La plataforma como servicio (platform as a service, PaaS) se ubica en la capa del medio, es la encapsulación de una abstracción de un ambiente de desarrollo y el empaquetamiento de una carga de servicios. Las ofertas de

PaaS pueden dar servicio a todas las fases del ciclo de desarrollo y pruebas del software, o pueden estar especializadas en cualquier área en particular, tal como la administración del contenido. Los ejemplos comerciales incluyen Google App Engine, que sirve aplicaciones de la infraestructura Google, y también Windows Azure, de Microsoft. Esto es una plataforma en la nube que permite el desarrollo y ejecución de aplicaciones codificadas en varios lenguajes y tecnologías como .NET, Java y PHP. Servicios PaaS tales como éstos permiten gran flexibilidad, pero pueden ser restringidos por las capacidades que están disponibles a través del proveedor.

La infraestructura como servicio (infrastructure as a service, IaaS) se encuentra en la capa inferior y es un medio de entregar almacenamiento básico y capacidades de cómputo como servicios estandarizados en la red. Servidores, sistemas de almacenamiento, conexiones, enrutadores, y otros sistemas se concentran (por ejemplo a través de la tecnología de virtualización) para manejar tipos específicos de cargas de trabajo. El ejemplo comercial mejor conocido es Amazon Web Services, cuyos servicios EC2 y S3 ofrecen cómputo y servicios de almacenamiento esenciales (respectivamente).

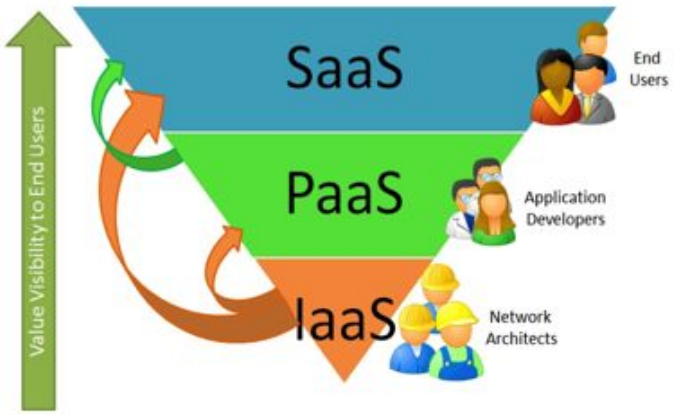


Figura 3.6: Modelos de Cloud Computing

3.2.2. Tipos de nube

En este apartado se presentan los diferentes tipos de nube que existen.

Las **nubes públicas** se refieren al modelo estándar de computación en nube, en el cual un proveedor de servicios, coloca sus recursos tales como aplicaciones y almacenamiento disponibles al público en general a través de Internet. Los servicios de la nube pueden ser libres u ofrecidos a través de un modelo de pago por uso. Un ejemplo de este tipo de nubes son Amazon Web Services y RackSpace.

Las **nubes privadas** son redes o centros de cómputo propietarios que usan tecnologías de computación en nube, tales como la virtualización. Se caracterizan por ser administradas por la organización a la que sirven y encontrarse aseguradas por medio de un Firewall. Un ejemplo de este tipo de nubes son OpenNebula y OpenStack.

Un tercer modelo son las denominadas **nubes híbridas**, son una mezcla de los dos modelos anteriores: las nubes públicas y privadas. Como ejemplo existe Eucalyptus + AWS.

3.2.3. Características

Un modelo de cloud computing, según The NIST Definition of Cloud Computing, debe cumplir las siguientes cinco características esenciales:

1. **Autoservicio bajo demanda.** El usuario puede acceder a capacidades de computación "en la nube" de forma automática conforme las necesita sin necesidad de una interacción humana con su proveedor o sus proveedores de servicios Cloud.

2. **Múltiples formas de acceder a la red.** Los recursos son accesibles a través de la red y por medio de mecanismos estándar que son utilizados por una amplia variedad de dispositivos de usuario, desde teléfonos móviles a ordenadores portátiles o PDAs.

3. **Compartición de recursos.** Los recursos (almacenamiento, memoria, ancho de banda, capacidad de procesamiento, máquinas virtuales, etc.) de los proveedores son compartidos por múltiples usuarios, a los que se van asignando capacidades de forma dinámica según sus peticiones. Los usuarios pueden ignorar el origen y la ubicación de los recursos a los que acceden, aunque sí es posible que sean conscientes de su situación a determinado nivel, como el de CPD o el de país.

4. **Elasticidad.** Los recursos se asignan y liberan rápidamente, muchas veces de forma automática, lo que da al usuario la impresión de que los recursos a su alcance son ilimitados y están siempre disponibles.

5. **Servicio medido.** El proveedor es capaz de medir, a determinado nivel, el servicio efectivamente entregado a cada usuario, de forma que tanto

proveedor como usuario tienen acceso transparente al consumo real de los recursos, lo que posibilita el pago por el uso efectivo de los servicios.

En la siguiente imagen se muestra el modelo tradicional de trabajo de las empresas de TI, y abajo el nuevo modelo de trabajo utilizando Cloud Computing.

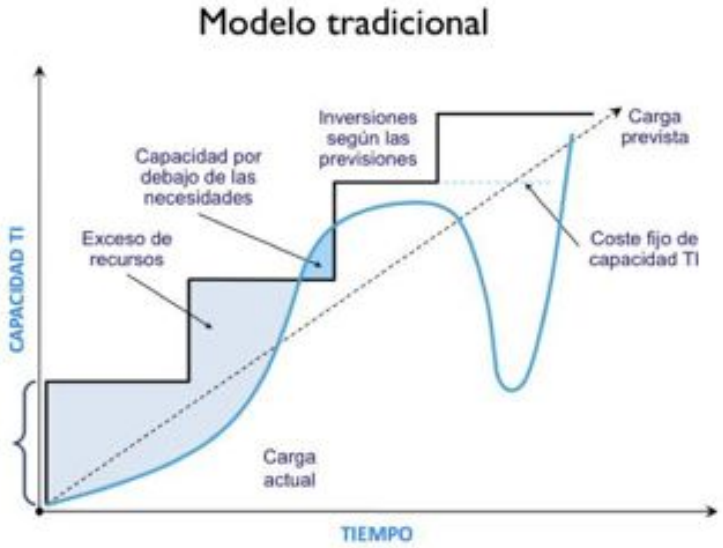


Figura 3.7: Modelo tradicional

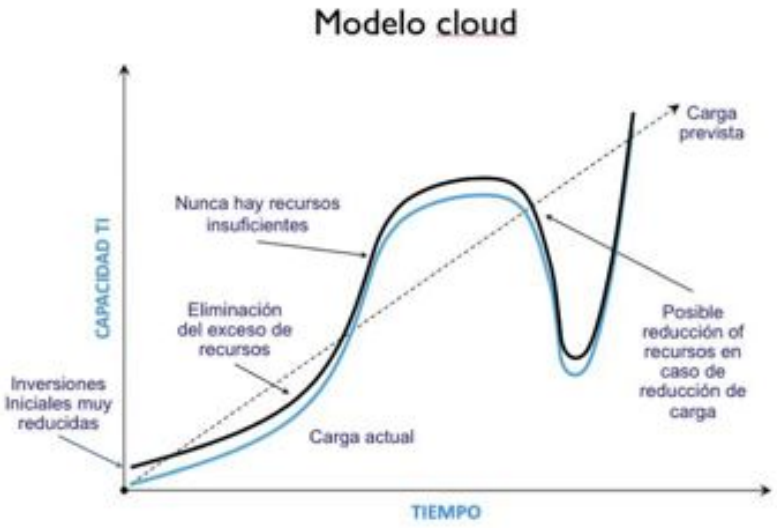


Figura 3.8: Modelo de Cloud Computing

3.3. Documentación de Xen

En esta segunda sección se presenta una breve descripción del entorno de virtualización *Xen*.

Xen es un hipervisor de máquinas virtuales de código abierto donde su finalidad es poder ejecutar instancias de sistemas operativos, con todas sus características, de forma completamente funcional en un equipo sencillo. *Xen* proporciona aislamiento seguro, control de recursos, garantías de calidad de servicio y migración de máquinas virtuales en caliente.

Xen utiliza la técnica de *paravirtualización* para alcanzar alto rendimiento (es decir, bajas penalizaciones del rendimiento). Con la *paravirtualización* se puede alcanzar alto rendimiento incluso en arquitecturas (x86) que no se suele conseguir con técnicas tradicionales de virtualización.

Está formado por los siguientes componentes:

- Hipervisor
- Dominios
- Kernel modificado de Linux
- Herramientas de administración por vía comando o de manera gráfica

El hipervisor es la pieza fundamental de *Xen*: es lo primero que ejecuta el sistema y se encarga de controlar el hardware (CPU, memoria, etc.) y distribuir su uso entre las diversas máquinas virtuales, se ejecuta por debajo incluso del sistema operativo anfitrión (**dominio 0**) proporcionando estabilidad, aislamiento entre máquinas y políticas de calidad, por estos motivos necesita ejecutarse en un lugar privilegiado.

Xen denomina a las máquinas virtuales *dominios* **domX** donde el **dom0** corresponde con el primer dominio que se lanza cuando arranca el kernel modificado de *Xen*, el anfitrión. El dominio **dom0** tiene los privilegios para crear, arrancar o eliminar otras máquinas virtuales o dominios, además de ayudar en las tareas de administración al hipervisor. El resto de dominios **domU** son distintas máquinas virtuales creadas a partir del dominio **dom0** y gestionadas por el hipervisor.

Una de las herramientas proporcionadas por *Xen* es el comando `xm`, que permite crear, arrancar, eliminar y configurar máquinas virtuales.

Con *Xen* también se puede utilizar la herramienta *Xen Tools* [11], que permite gestionar las máquinas virtuales de una forma más sencilla y agradable al usuario. De igual forma, se pueden crear nuevas máquinas creando un fichero de configuración y ejecutando un simple comando de *Xen Tools*.

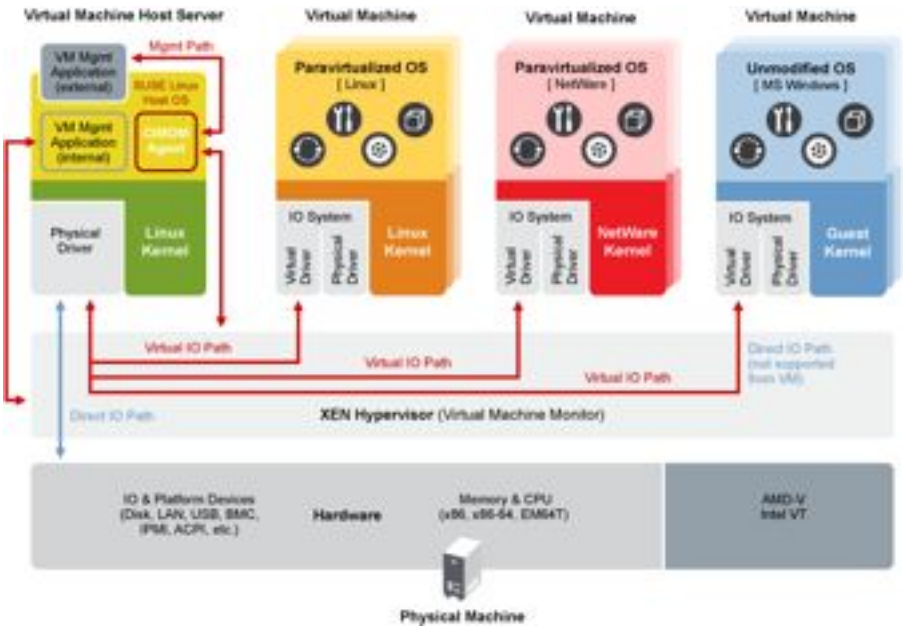


Figura 3.9: Esquema de *Xen*.

Otra de las herramientas para la creación de máquinas virtuales es *Virtual Machine Manager* [12]. Esta herramienta, escrita en Python, permite gestionar las máquinas virtuales utilizando un entorno gráfico agradable al usuario. También posee otras herramientas como la clonación de máquinas virtuales de forma fácil o la conexión al escritorio de una máquina virtual de forma remota.

A parte de estas herramientas, existen diversas herramientas que gestionan las máquinas virtuales de *Xen* para distintos propósitos, todas ellas creadas a partir de la API que proporciona *Xen*.

En la figura 3.9[1] se muestra el esquema que sigue *Xen* para su funcionamiento.

3.3.1. Migración en vivo de máquinas virtuales

Una de las características por las que se ha elegido *Xen* para este proyecto es la posibilidad de migrar máquinas virtuales en vivo, es decir, sin ser paradas, a otros nodos remotos. Esta característica hace que se puedan migrar las máquinas virtuales que contenga un nodo para posteriormente apagar el mismo, sin que el usuario final sea consciente de dicho suceso, pudiendo continuar

[1]Desde: www.xen.org

con el servicio que le estaba ofreciendo.

Durante el proceso de migración, la memoria de la máquina virtual origen se copia de forma iterativa al destino sin detener su ejecución. Solamente es necesaria una pausa muy breve, alrededor de 60 a 300 ms, para realizar la sincronización final antes de que la máquina virtual comience a ejecutarse en su destino final. Las conexiones de red de dicha máquina virtual tampoco se interrumpen, sus rutas son redirigidas en el nuevo entorno. De igual forma, iterativamente se copian los registros del estado de CPU y transacciones E/S. Por otro lado, el disco duro debe ser compartido por los nodos que forman parte de la migración, puesto que el disco duro permanece estático, es decir, no se copia.

3.3.2. API de *Xen*

Xen proporciona una API, *Xen Management API*, que es una interfaz para la configuración y control remotos de dominios activos en un host con *Xen*.

La API que utiliza Xen está realizada mediante RPC (Remote Procedure Calls) basada en XML-RPC. Tiene la ventaja que no especifica ningún lenguaje predefinido, es decir, se adapta a cualquier lenguaje que realice y controle las llamadas RPC.

En la figura 3.10[2] se muestra un esquema de gestión de la programación con Xen y algunas librerías que se han ido adaptando a este funcionamiento. Algunas de ellas, como *libvirt*, *xm* o *virt-manager* son las que se han utilizado para el desarrollo de este proyecto.

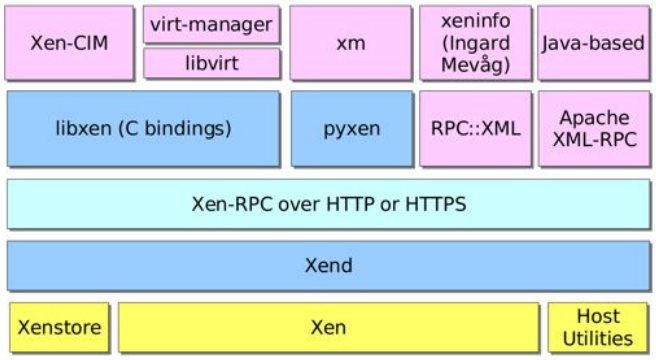


Figura 3.10: API de Xen.

[2]Desde: Xen Summit, Tj Watson Research Center, 2007

3.3.3. Librería *Libvirt*

Tal y como se ha indicado previamente, una de las librerías que se ha utilizado en este proyecto es *Libvirt* [10], que permite gestionar máquinas virtuales con Xen a través de su API.

Libvirt es una librería que utiliza una API de virtualización, de carácter libre, que permite a los hipervisores administrar de forma segura los sistemas operativos huéspedes que se ejecutan en un host. Ofrece una API común para la funcionalidad común que implementan los hipervisores soportados. *Libvirt* se diseñó originalmente como API de gestión para *Xen*, pero desde entonces se ha ampliado para soportar gran cantidad de hipervisores. Su API puede ser utilizada en distintos lenguajes de programación como Python, C, C++, Java y C#, entre otros.

En la figura 3.11[3] se muestra la arquitectura *Libvirt* basada en drivers.

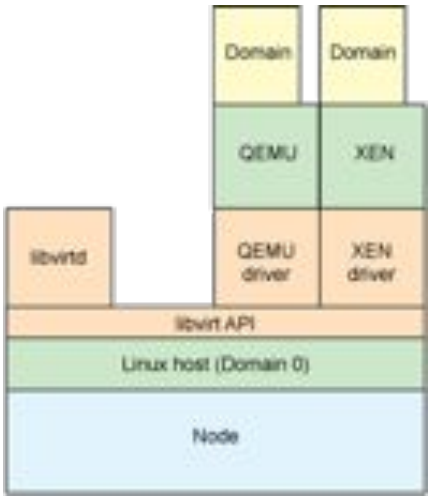


Figura 3.11: Arquitectura de Libvirt.

3.4. Preparación del *clúster*

Una vez descritos los conceptos de virtualización, cloud computing y Xen, se pasa a describir el trabajo realizado comenzando por la preparación del *clúster* que se ha utilizado para el proyecto.

[3]Desde: http://www.ibm.com/developerworks/ssa/linux/library/l-libvirt/

3.4.1. Instalación de equipos

La primera tarea a realizar ha sido la configuración de un *clúster* en un *rack* formado por tres equipos, un *frontend* y dos nodos. En la figura 3.12 se puede observar la instalación realizada. Además, se ha instalado un *switch* para crear la red interna entre los nodos del *clúster*, tal y como se observa en la figura 3.12.

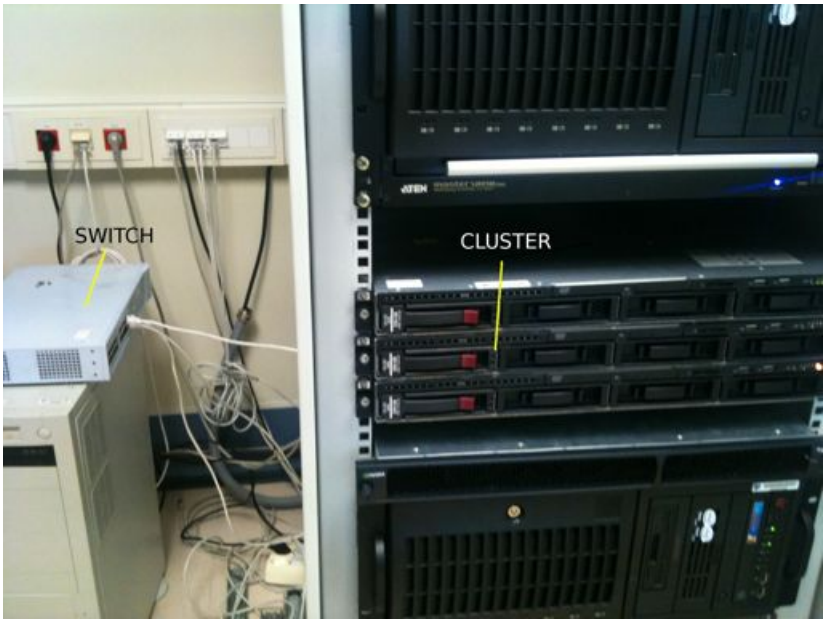


Figura 3.12: Montaje del *clúster*.

Los dos nodos del *clúster* pueden ser encendidos de forma remota por el *frontend* a través del comando `ether-wake`. Para ello, disponen de la correcta configuración de la BIOS y del cable Wake On LAN que conecta la placa base del nodo con su tarjeta de red. En la figura 3.13 se puede observar la conexión realizada en uno de los nodos. En ella se resalta el cable Wake On LAN, la tarjeta de red y el conector JWOL de la placa base.

3.4.2. Instalación del sistema operativo en los equipos del clúster

A continuación se ha realizado la instalación, en cada uno de los equipos del *clúster*, de la distribución Ubuntu Server 12.04 LTS de linux. Durante la instalación se ha marcado la opción de instalación SSH para poder acceder a los equipos de forma remota. Una vez instalado el sistema operativo se ha instalado el paquete del hipervisor Xen que se encuentra en los repositorios

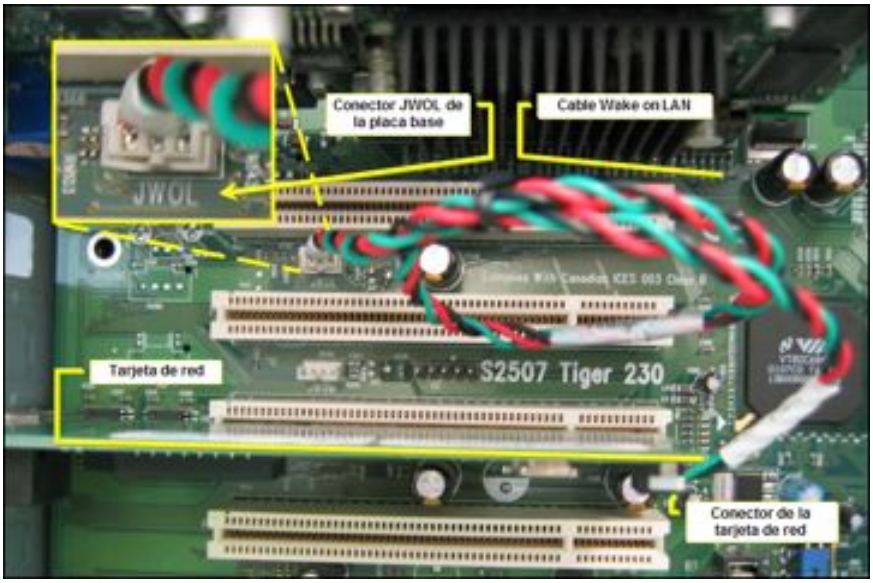


Figura 3.13: Conexión del cable Wake On LAN.

de Ubuntu, con lo que una vez instalado se ha agregado una nueva entrada en el Grub correspondiente al dominio 0 (máquina virtual principal en Xen), que contiene el kernel modificado con Xen. Todo esto se ha realizado en los tres equipos.

3.4.3. Configuración de la red

La siguiente tarea consiste en la creación de la red interna en el *clúster*, mediante el switch descrito anteriormente. De esta forma, los nodos se pueden comunicar entre sí para poder realizar posteriormente la migración de máquinas virtuales. El nombre asignado al *clúster* es *Neon*.

A cada equipo se le ha asignado una dirección IP estática, de forma que el *frontend* posee la dirección `192.168.1.1/24` y el resto de nodos poseen la direcciones correlativas a ésta, en el caso del *Nodo1* la dirección `192.168.1.2/24` y para el *Nodo2* la dirección `192.168.1.3/24`.

Por último, el *frontend* posee una tarjeta de red adicional para su conexión a Internet, por lo que se ha asignado una dirección IP estática proporcionada por los técnicos del departamento de Ingeniería y Ciencia de los Computadores y se le ha asignado un nombre al equipo para acceder a él desde Internet de una forma más cómoda, siendo dicho nombre *neon.act.uji.es*.

3.4.4. Configuración SSH entre equipos

Para evitar introducir una contraseña cada vez que se accede a un nodo desde el *frontend* a través de SSH manteniendo el nivel de seguridad, es necesario utilizar un método de acceso sin contraseña. Este método consiste en la generación de un par de claves RSA [8], una pública y otra privada. Todo mensaje que se cifra con una clave pública solo puede ser descifrado por su clave privada y a la inversa.

En todos los equipos del *clúster* se han generado este par de claves. Además, la clave pública de cada equipo se ha distribuido al resto de equipos de la red interna. Con ello, cada equipo guarda un registro de claves públicas de los otros equipos en un archivo para el acceso autorizado con SSH.

Cuando un equipo desee autenticarse vía SSH, el emisor busca la clave pública del receptor, cifra la autenticación con dicha clave, y una vez que la autenticación cifrada llega al receptor, éste se ocupa de descifrarla usando su clave privada. De esta forma se consigue autenticar los equipos de forma segura sin necesidad de utilizar contraseña, ya que las claves pública y privada ofrecen ambos beneficios.

Una vez generadas y transmitidas todas la claves necesarias, es necesario reiniciar el demonio de SSH en todos los equipos para que la configuración surja efecto.

3.5. Gestión de máquinas virtuales

Una vez configurado el *clúster*, la siguiente tarea ha sido la gestión de máquinas virtuales en dicho *clúster*.

En esta sección se muestran las máquina virtuales creadas en el desarrollo del proyecto, así como la configuración dada a cada una de ellas.

3.5.1. Creación de máquinas virtuales

Cada máquina virtual tiene asociado un fichero de configuración a partir del cual se arranca la máquina. Dicho fichero incluye características como la cantidad de memoria RAM, la cantidad de CPUs virtuales, o la ubicación de su disco duro entre otros. Este fichero se puede crear de forma manual o a partir de herramientas para la gestión de máquinas virtuales con *Xen*. Cada máquina virtual necesita un disco duro que se debe crear previamente usando un fichero o una partición.

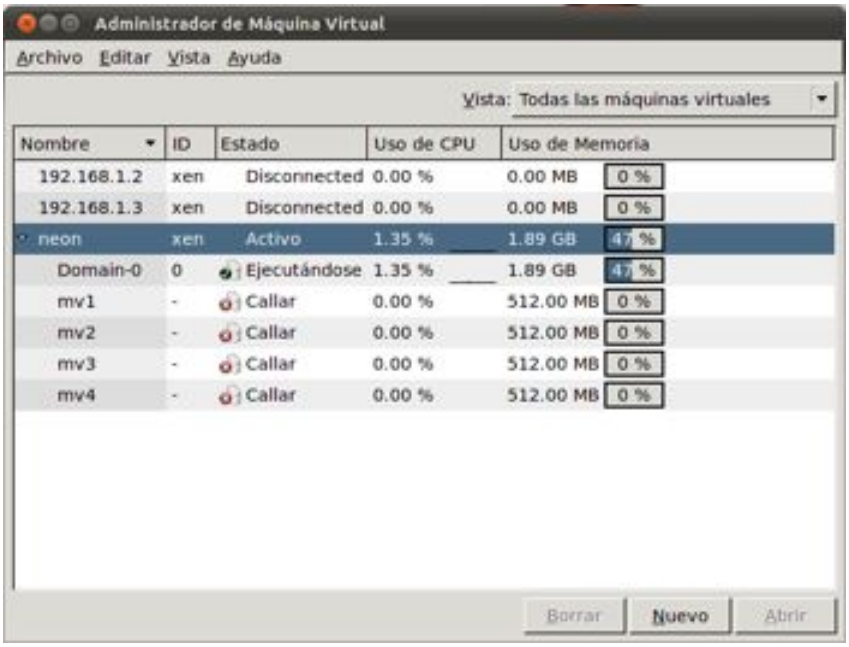


Figura 3.14: Interfaz de Virtual Machine Manager.

En este proyecto se han creado cuatro máquinas virtuales completamente virtualizadas en el *frontend* utilizando para ello la herramienta de gestión de máquinas virtuales *Virtual Machine Manager*. En cada una de ellas se ha instalado el sistema operativo linux Ubuntu 12.04 LTS Server. Se han llamado *mv1, mv2, mv3 y mv4* respectivamente. Cada una de ellas es un clon de la anterior, cambiando su nombre de host y dirección IP.

En la figura 3.14 se muestra la interfaz gráfica de la herramienta *Virtual Machine Manager* con las cuatro máquinas virtuales creadas.

Todos los discos duros de dichas máquinas comparten un espacio de almacenamiento común en el directorio `/vm` del *frontend*, este directorio corresponde con una partición que se describe más adelante.

En la tabla 3.1 se muestra la configuración dada a las máquinas virtuales creadas:

Nombre	Host	IP	VCPUs	Memoria	Ubicación
mv1	mv1	192.168.1.100/24	2	512 MB	/vm/mv1.img
mv2	mv2	192.168.1.101/24	2	512 MB	/vm/mv2.img
mv3	mv3	192.168.1.102/24	2	512 MB	/vm/mv3.img
mv4	mv4	192.168.1.103/24	2	512 MB	/vm/mv4.img

Cuadro 3.1: Configuración de máquinas virtuales.

3.5.2. Configuración de la red de las máquinas virtuales

Xen utiliza distintos scripts para gestionar la red entre las máquinas virtuales. A continuación se describen tres de ellos, pero se pueden crear más dependiendo de las necesidades de la red:

- **Puente (*bridge*)**
  La configuración por defecto de Xen crea puentes de red dentro de **dom0** para permitir que todos los dominios aparezcan en la red como hosts independientes. Este método lo que hace es tomar una dirección IP dentro del rango de la red donde se conecta, ya sea estática o dinámica.
- **Encaminador (*router*)**
  Esta configuración se aplica cuando:
  - Las máquinas están en una LAN diferente.
  - El tráfico de red se dirige hacia el exterior.
- **NAT**

  Esta configuración se realiza cuando el **domU** se encuentra en una LAN privada, de esta forma, **domU** hace NAT con **dom0** para poder llegar a otra LAN, pareciendo como si el tráfico procediera de **dom0**.

En este caso se ha utilizado la configuración más común, puente (*bridge*), en la que las máquinas virtuales pasan a formar parte de la red interna del *clúster*. Para ello, cuando se lanza el demonio de Xen, *xend*, se ejecuta el script `network-bridge`, que realiza las siguientes acciones:

- Crea un nuevo brigde llamado `xenbr0`.
- La interfaz real `eth0` se desactiva.
- Se copia la dirección IP y MAC de `eth0` a la interfaz virtual `veth0`.
- Se renombra la interfaz real como `peth0`.
- Se renombra la interfaz virtual como `eth0`.

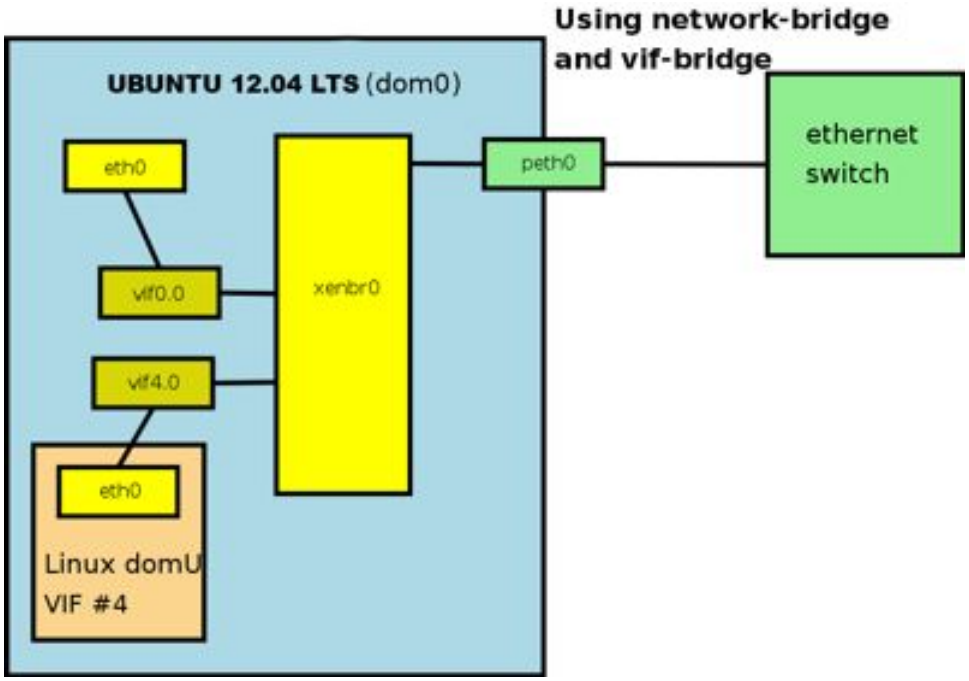


Figura 3.15: Configuración de la red con *Xen* mediante un puente.

- `peth0` y `vif0.0` se conectan al puente `xenbr0`.
- Se activan `xenbr0, peth0, eth0` y `vif0.0`.

Cuando se inicia un domU, *xend* (que se está ejecutando en dom0) ejecuta el script *vif-bridge*, que realiza las siguientes operaciones:

- Conecta `vif<id#>.0` a `xenbr0`.
- Activa `vif<id#>.0`.

Donde `id#` es el número de identificación del domU.

En la figura 3.15 se puede ver de forma esquemática cómo quedan conectados los dominios tras configurar el puente en *Xen* con un domU.

3.6. Configuración para la migración con *Xen*

Una vez creadas las máquinas virtuales y configurada la red entre las mismas, se pasa a describir cómo configurar Xen para realizar la migración en vivo de máquinas virtuales entre los nodos del *clúster*.

3.6.1. Configuración de *Xen*

Xen posee un archivo de configuración en `/etc/xen/xend-config.sxp` a través del cual se pueden modificar algunos parámetros de configuración.

En este caso se pretende configurar *Xen* para permitir la migración de máquinas virtuales entre los nodos de la red interna. Para ello, se deben descomentar las siguientes líneas de dicho fichero:

- `(xend-relocation-server yes)`

  Este parámetro activa el servidor de migración para permitir la migración de máquinas virtuales.

- `(xend-relocation-port 8002)`

  Este parámetro indica el puerto por el que se realizará la migración, en este caso es el 8002.

- `(xend-relocation-address ")`

  Este parámetro indica las direcciones que puede escuchar el demonio *xend*, es decir, los hosts que pueden realizar la petición de migrar alguna de sus máquinas virtuales a este host. Si el campo entre comillas está vacío indica que cualquier host puede realizar la petición.

- `(xend-relocation-hosts-allow ")`

  Este parámetro se utiliza para indicar a qué hosts se les puede solicitar la migración de alguna de las máquinas virtuales que contenga. Si el valor de las comillas está vacío indica que todas las conexiones están permitidas.

Una vez realizada la configuración es necesario reiniciar el demonio *xend* para que surjan efecto los nuevos parámetros de configuración.

Estos parámetros se han configurado en todos los equipos del *clúster*, incluido el *frontend*.

3.6.2. Sistema de ficheros NFS

Tal y como se ha comentado previamente, cada máquina virtual creada en el *frontend* tiene un disco duro asociado, que se almacena como un fichero más del sistema. Cuando se realiza la migración de una máquina virtual se copia toda la memoria y todos los parámetros de la CPU y E/S, exceptuando el disco duro, es decir, el disco duro permanece en el mismo sitio donde se creó en un principio para la máquina virtual.

Por tanto, para poder realizar la migración entre equipos, es necesario la creación de un sistema de compartición de archivos, en los que se compartan los archivos correspondientes a los discos duros de las máquinas virtuales creadas. Es decir, debe existir un espacio común entre todos los nodos del *clúster* en donde se almacenen todos los discos duros de las máquinas virtuales.

El sistema de compartición de ficheros utilizado para el proyecto es el sistema NFS (Network File System) [9].

Para este proyecto, el *frontend* es el encargado de almacenar todos los ficheros correspondientes a los discos duros de cada máquina virtual en un espacio compartido a todos los nodos. Para ello, se ha creado una partición, `/vm`, donde se almacenarán dichos ficheros. Esta partición se ha compartido a través de NFS modificando el fichero `/etc/exports`, y los ficheros de configuración de permisos `/etc/hosts.allow` y `/etc/hosts.deny`.

En cada nodo, se ha modificado el fichero `/etc/fstab` para que monte el correspondiente sistema de ficheros compartido por el *frontend* en el directorio `/vm` para que, de esta forma, forme parte de su sistema de ficheros cada vez que se inicie el sistema.

3.6.3. Configuración de puertos

Para mantener la seguridad de los equipos es necesario tener activado el cortafuegos. Pero, para que funcionen algunas de las configuraciones que se han realizado hasta ahora, es necesario abrir una serie de puertos en los equipos del *clúster*.

A continuación se muestran los servicios configurados y el puerto que se ha abierto en cada equipo. Para abrir un puerto se ha introducido la línea correspondiente mediante el comando *iptables* de Ubuntu.

- **Migración de máquinas virtuales con Xen**

  Puerto: 8002 TCP

- **Configuración SSH**

  Puerto: 22 TCP

- **Sistema de compartición de ficheros, NFS**

  Puerto: 111 TCP/UDP, 2049 TCP/UDP

  En este caso se han modificado una serie de puertos del archivo de configuración de dicho servicio para permitir montar la unidad y otros servicios, modificando el archivo `/etc/sysconfig/nfs`, con lo que también se han abierto los puertos: 32772 TCP/UDP y 32774 TCP/UDP.

- **Enersis**

  Puerto: 5320 TCP

  Este servicio corresponde con los demonios creados para este proyecto descritos más adelante.

3.7. Programación de las librerías intermedias

Una vez preparado el *clúster* para realizar la migración en vivo de máquinas virtuales, se pasa a describir las librerías y demonios que permiten desarrollar el sistema de ahorro de energía Enersis.

Considerando como punto de partida la librería *Libvirt* descrita anteriormente, se han desarrollado las siguientes librerías. A continuación se detallan las características de las mismas.

3.7.1. Librería *CPUinfo*

Para obtener la carga de cada nodo, es decir, el porcentaje de CPU utilizado en cada nodo, ha sido necesario la creación de una nueva librería escrita en Python que realice dicha tarea. Esta librería, denominada *CPUinfo*, se ha situado en todos los nodos del *clúster* en `/usr/lib/enersis`. Es importante comentar que cuando se inicia un nodo, el sistema operativo anfitrión corresponde con el **dom0**, es decir, es una máquina virtual como tal. Esta librería permite obtener el porcentaje de CPU que se está utilizando, en un momento dado, en el nodo completo y no solamente en el **dom0**, para ello se ha utilizado la librería *Libvirt*.

La librería *CPUinfo* está compuesta por la clase `CPUinfo`, de forma que cuando se crea una nueva instancia de dicha clase, realiza las siguientes acciones:

- Establece una conexión con el hipervisor del nodo donde se encuentra.
- Guarda información sobre el número de CPUs que contiene dicho nodo.

Esta clase tiene una función `compute`, que cuando se ejecuta devuelve el porcentaje de CPU en intervalos de tiempo de 0,1 segundos. Esta función realiza las siguientes acciones:

- Recorre todos los dominios (máquinas virtuales) activos que contenga el nodo.
- Guarda el tiempo de CPU usado en cada dominio en nanosegundos.
- Realiza un espera de 0,1 segundos.
- Vuelve a guardar un nuevo tiempo de CPU usado en cada dominio.
- Guarda en una variable la suma de las diferencias de los tiempos de cada dominio.
- Obtiene el porcentaje de CPU, realizando una división entre el valor obtenido por cien y el intervalo de tiempo multiplicado por 10^9 y el números de CPUs del nodo. Con ello se extrae el tiempo de carga que ha tenido el nodo en dicho intervalo de tiempo.

Esta librería devuelve dicho porcentaje de CPU cuando se llama a la función `compute`.

Cabe destacar la importancia de esta librería en este proyecto, ya que es un elemento clave a la hora de realizar el balanceo de carga entre los nodos del sistema.

3.7.2. Librería *Servidor*

Esta librería se ha creado para realizar la tarea de servidor de datos cuando el *frontend* necesite información sobre un nodo, es decir, su carga o el número de dominios que contiene. Se ha situado en todos los nodos del *clúster* en `/usr/lib/enersis`. Esta librería también hace uso de la librería *Libvirt* para realizar las conexiones con el hipervisor y obtener información sobre él.

Está compuesta por la clase `Cliente` que hereda de la clase `Thread`. Cada instancia que se crea de esta clase corresponde con un nuevo *Thread* que se ejecuta de forma independiente en el nodo donde se encuentre. Por tanto se pueden tener tantos *Threads* de información como soporte el nodo.

Cuando se crea una nueva instancia de la clase `Cliente` se deben pasar los siguientes parámetros:

- El socket cliente, es decir, la conexión realizada una vez aceptada la conexión de un nuevo cliente.
- Los datos del socket, que contienen información sobre la conexión aceptada.

Todo *Thread* tiene una función de arranque, en este caso, cuando se ejecute la función `start` de la nueva instancia, se ejecutará la función `run` del código creado, que realiza las siguientes acciones:

- Recibe la petición enviada por el cliente.
- Si se recibe la palabra `cpu` seguida de un entero mayor que 1 se realizan las siguientes acciones:
  - Se extrae el tiempo deseado por el cliente.
  - Se lee el fichero de registro o LOG (`/var/log/enersis-cpu.log`) que contiene toda la información sobre los porcentajes de CPU de cada segundo.
  - Se leen los últimos segundos del fichero, pedidos por el cliente.
  - Se realiza la media aritmética de los datos obteniendo, con ello, el porcentaje medio.
  - Se devuelve el porcentaje medio por el socket.
- Si se recibe la palabra `domains` se realiza una conexión con el hipervisor del nodo donde se encuentra y se obtiene la información del número de dominios activos. Se devuelve dicha información por el socket.
- En caso de no corresponder con ninguno de estos dos casos, se descarta la petición.

3.7.3. Librería *Nodo*

La librería *Nodo* se ha creado para dar soporte al demonio principal del *frontend*, *enersis-frontend*. Se ha situado en el directorio `/usr/lib/enersis` del *frontend*.

Esta librería contiene todas las funciones necesarias para la gestión de los nodos a través del *frontend*, entre ellas están tareas como encender y apagar un nodo, comprobar la carga y obtener el número de dominios entre otros.

La librería *Nodo* está formada por la clase `Nodo` y hace uso de las librerías *Libvirt* y *Socket*, entre otras.

Cuando se crea una nueva instancia de la clase `Nodo`, se debe iniciar con los siguientes parámetros para realizar la correcta correspondencia con el nodo deseado:

- **IP**: corresponde con la dirección IP del nodo.
- **MAC**: corresponde con la dirección MAC del nodo.
- **Tiempo de CPU**: este parámetro indica el intervalo de tiempo del que se desea obtener información de CPU del nodo.

A continuación se describen las funciones que tiene dicha librería para la gestión de cada nodo:

- **Encender**

  Para poder encender un nodo cuando está apagado es necesario utilizar el sistema *Wake On LAN* descrito en el inicio de la memoria. Mediante el comando `ether-wake` junto con la dirección MAC del nodo, se envía un paquete mágico por la red interna del *clúster*, el cual es detectado por el nodo que contiene dicha MAC y enciende el nodo.

  Para detectar que el nodo se ha encendido correctamente y está disponible para su uso, se ha iniciado un *socket* a la espera de que el nodo le responda con un `1`, que envía el demonio *enersis-servidor* del nodo una vez iniciado el sistema.

  Una vez recibida la información de arranque correcto del nodo, espera durante un intervalo de tiempo de 10 segundos para que se inicien correctamente todos los servicios del nodo. Posteriormente, se realiza la conexión con su hipervisor para poder realizar las migraciones correspondientes.

- **Apagar**

  Para apagar un nodo, primero se realiza la desconexión con su hipervisor para terminar correctamente con el servicio conectado. Posteriormente, se envía un comando `shutdown -h now` a través de *SSH* en el que se indica al nodo que debe apagar el equipo inmediatamente.

  Una vez enviado el comando se espera un intervalo de tiempo de 35 segundos para asegurarse que realmente se ha apagado el nodo, ya que este tiempo es el que necesita el nodo para apagar todos sus servicios y realizar la desconexión.

- **Conectar**

  Mediante esta función se realiza la conexión con el hipervisor utilizando para ello la librería *Libvirt* junto con la IP del nodo. Además, se inicia un *socket* que realiza la conexión al nodo, para poder posteriormente pedir y obtener información sobre él.

- **Desconectar**

  Esta función envía la petición `exit` a través del *socket* creado en la conexión para indicar al nodo que ya no desea realizar más peticiones. Una vez enviado el comando, se cierra el *socket*.

- **Carga**

  Para obtener información sobre el estado de la CPU del nodo, en el intervalo de tiempo indicado al iniciar la instancia, se envía el comando `cpu` seguido del intervalo de tiempo a través del *socket* iniciado en la conexión.

  Una vez enviado el comando, pasa a la espera de recibir la información pedida al nodo. El nodo recibe la petición, y una vez procesada, envía la información solicitada a través del *socket*.

- **Dominios**

  Esta función permite obtener el número de dominios activos en el nodo, es decir, el número de máquinas virtuales activadas que almacena en ese instante. Para ello envía el comando `domains` a través del *socket* creado en la conexión.

  Una vez enviado el comando, pasa a la espera de recibir la información pedida al nodo. Cuando el nodo recibe la petición, envía la información solicitada a través del *socket*.

3.7.4. Librería *Virtual Machine*

Esta librería da soporte al demonio principal del *frontend*, *enersis-frontend*. Se ha situado en el directorio `/usr/lib/enersis` del *frontend*.

Esta librería contiene las funciones necesarias para la gestión de las máquinas virtuales, entre ellas están las funciones de migración y comprobación de activación de una máquina en el *frontend*, además de una función para obtener la carga de CPU que supone esta máquina virtual para el nodo.

La librería *Virtual Machine* está compuesta por la clase `VirtualMachine` y hace uso de las librerías *Libvirt* y *Nodo*.

Cuando se crea una nueva instancia de dicha clase se debe iniciar con los siguientes parámetros descritos a continuación:

- **Máquina**: corresponde con el nombre de la máquina virtual.
- **Estado**: indica el estado de la máquina, 1 encendida, 0 apagada.
- **Nodo**: corresponde con la IP del nodo que la alberga, por defecto es el *frontend*.

Una vez iniciados, la función de inicio realiza una conexión al hipervisor del nodo que la contiene y conecta con la máquina virtual.

A continuación se describen las funciones que tiene dicha librería para gestionar las máquinas virtuales.

- **Migrar**

  Esta función tiene como parámetro la instancia del nodo donde se desea migrar la máquina virtual, de esta forma, a través de una función de la librería *Libvirt*, se realiza la migración en caliente de dicha máquina al nodo destino.

  Una vez migrada la máquina, se realiza nuevamente la conexión con el hipervisor del nodo destino y la máquina migrada.

- **enFronEnd**

  Esta función conecta con el *frontend* y comprueba si se ha iniciado la máquina virtual en el *frontend*.

- **cpuRelativa**

  Esta función utiliza la librería *Libvirt* para obtener la carga de CPU relativa que ocupa esta máquina virtual respecto al nodo que la contiene. Obtiene la información de forma similar a como se obtiene la carga de CPU de un nodo, es decir, analiza la diferencia de CPU en un intervalo de 0.1 segundos y mediante una operación matemática obtiene el porcentaje que está utilizando sobre el nodo que la contiene. Esta función se utiliza para el algoritmo de toma de decisiones a la hora de decidir qué máquina virtual mover.

3.8. Demonios para los nodos

Para dar servicio al *frontend* en la toma de sus decisiones a la hora de realizar la migración de las máquinas virtuales, se han creado dos demonios en los nodos, de forma que cada vez que el *frontend* necesite comprobar el estado de un nodo, los demonios realizan las tareas de recopilar información sobre el nodo e informar al *frontend* de su estado. Mediante estos demonios, se puede determinar si un nodo está muy cargado y por lo tanto necesita migrar alguna de las máquinas virtuales que contenga, o si por el contrario tiene muy poca carga y puede liberar las máquinas virtuales que contenga para poder ser apagado y ahorrar energía, ya que si no se estaría usando de forma ineficiente gastando energía innecesaria.

Estos demonios se sitúan en el directorio `/usr/sbin` de cada nodo.

A continuación se explica el funcionamiento y las tareas de cada uno de ellos.

3.8.1. Demonio *enersis-cpu*

Este demonio se ha creado para recopilar información, de forma periódica, sobre el porcentaje de CPU usado por el nodo. Este servicio es arrancado, al iniciar el nodo, por el arrancador `enersis` descrito más adelante. Una vez iniciado, guarda información de la CPU en un fichero de registro (LOG), situado en `/var/log/enersis-cpu-log`, en intervalos de tiempo de 1 segundo.

El demonio está estructurado de la siguiente forma:

- Carga las librerías *Libvirt*, *CPUinfo* y *Thread* entre otras.
- Crea la clase `CargaCPU` que hereda de la clase `Thread`, de forma que cuando se realiza una nueva instancia recibe como parámetro el fichero de registro donde debe escribir.
- Cuando se empieza a ejecutar el *Thread*, inicia una nueva instancia de la clase `CPUinfo`.
- Repite las siguientes tareas de forma indefinida:
  - Para escribir en intervalos de un segundo, recopila la información de la CPU mediante la función `compute` de `CPUinfo` durante 10 veces (`CPUinfo` proporciona el estado de la CPU del último 0,1 segundo).
  - Recopilada la información, realiza la media aritmética y formatea la salida a cuatro decimales para facilitar la posterior búsqueda en el fichero de registro.
  - Una vez obtenido el porcentaje de CPU de un segundo, escribe al final del fichero de registro.
- Por otra parte, se ha creado una función `daemonize` para convertir esta tarea en un demonio del sistema, de forma que guarda el PID de la tarea en el fichero `/var/run/enersis-cpu.pid`.
- Por último, se ha creado un `Main` que inicia todo el proceso.

3.8.2. Demonio *enersis-servidor*

Este demonio tiene la función de recibir las peticiones del *frontend* y proporcionarle la información que le pida sobre el nodo. Para ello, utiliza la librería *Servidor* descrita anteriormente. Además, este demonio es el encargado de enviarle al *frontend* la información de encendido correcto del nodo, es decir,

envía un `1` por un *socket* al *frontend* indicándole que se ha encendido de forma correcta. Este demonio se situa en el directorio `/usr/sbin` de cada nodo.

Este demonio realiza las siguiente acciones:

- Lee el fichero de configuración que contiene la dirección IP del *frontend*. Este archivo esta situado en `/etc/enersis/enersis-servidor.conf` de cada nodo.
- Para comprobar que el fichero de configuración es correcto, realiza un test mediante una expresión regular de forma que si todo es correcto continua con la ejecución, de lo contrario aborta la ejecución.
- Una vez obtenida la IP del *frontend*, inicia un *socket* que conecta con el `frontend` y le envía un `1` indicando que el nodo se ha encendido correctamente.
- Luego, cierra la conexión iniciada con el *socket* e inicia otra a la espera de peticiones del *frontend*.
- Utilizando la librería *Servidor*, descrita anteriormente, realiza un bucle infinito a la espera de recibir nuevas peticiones e iniciar una nueva instancia de la clase `Cliente`.
- Cuando se recibe una nueva petición, inicia un nuevo *Thread* para atender la petición y a través de la librería *Servidor* se envía la información correspondiente.
- Inicia la función `daemonize` para convertir la función que realiza en un demonio del sistema, y guarda su PID en el fichero `/var/run/-enersis-servidor.pid`.
- Por último, la función `Main` inicia todo el proceso.

3.8.3. Arrancador *enersis*

Para iniciar los demonios de los nodos, descritos anteriormente, se ha creado un código de arranque llamado *enersis*, de forma que cuando se inicia el sistema ejecuta este código e inicia los demonios correspondientes a los nodos. Está escrito en *bash* siguiendo la estructura de los arrancadores de Linux.

Se le ha dado la prioridad mínima en el proceso de arranque automático al iniciar el sistema, ya que los demonios creados necesitan utilizar otras librerías como *Libvirt* que son iniciadas por otros demonios. Está situado en `/etc/rc.d/init.d/` para que se pueda iniciar de forma automática al iniciar el sistema, además de poder ser utilizado como un servicio más.

Este código realiza las siguientes funciones:

- `start`

  Esta función realiza la tarea de iniciar los demonios `enersis-cpu` y `enersis-servidor`.

- `stop`

  Con esta función se termina con la ejecución de los demonios iniciados y borra los ficheros PID.

- `status`

  Esta función comprueba si los demonios se están ejecutando de forma correcta a través de la función `statusES`. Si alguno de los demonios no se está ejecutando y no se ha creado el fichero PID, se muestra el mensaje pertinente, si no se ha iniciado correctamente pero sí que ha creado el PID también informa al usuario del suceso.

- `restart`

  Esta función realiza las tareas de apagar los demonios y volver a iniciarlos.

- `statusES`

  Esta es una función que es utilizada por la función `status` para realizar la comprobación de ejecución correcta de los demonios.

Este código se inicia de forma automática al iniciar sistema, pero puede ser iniciado de forma manual con el siguiente comando:

```
#service enersis {start|stop|restart|status}
```

3.9. Demonio para el *frontend*

3.9.1. Demonio *enersis-frontend*

Este demonio, `enersis-frontend`, es el principal demonio de este proyecto ya que es el que realiza el arbitraje de arranque y apagado de nodos y migración de máquinas virtuales entre los mismos. Está situado en `/usr/sbin` y es iniciado por el código arrancador `enersis`. Una vez iniciado, guarda el PID en `/var/run/enersis-frontend.pid`.

Cuando se inicia este demonio, su primera tarea es cargar toda la información sobre los nodos y las máquinas virtuales creadas. Esta información se lee a partir de un fichero de configuración situado en `/etc/enersis/-enersis-frontend.conf`. Con esta información se crea un vector de nodos y

de máquinas virtuales utilizando las librerías *Nodo* y *Virtual Machine* descritas anteriormente. Posteriormente se ejecuta el algoritmo principal que realiza la función de ahorro de energía.

A continuación se muestra con más detalle el funcionamiento de este demonio:

- Lee el fichero de configuración situado en el directorio `/etc/enersis/-enersis-frontend.conf`.
- Comprueba que el fichero está estructurado de forma correcta y contiene información válida, para ello se ha creado una función que, a través de expresiones regulares comprueba que es correcto. En caso de no contener alguna información válida aborta la ejecución.
- Carga toda la información del fichero de configuración a un vector de máquinas virtuales y a otro de nodos. Además, carga información sobre los umbrales de porcentaje de CPU a tener en cuenta para la migración de las máquinas virtuales, así como el tiempo de ciclo de cada comprobación que se realiza en los nodos. Esta información se obtiene del fichero de configuración *enersis-frontend.conf*.
- Una vez cargados todos los datos, inicia un bucle infinito que ejecuta el algoritmo de arbitraje cada tiempo de ciclo.
- Inicia la función `daemonize` para convertir la función principal en un demonio del sistema, guardando su correspondiente PID en el fichero `/var/run/enersis-frontend.pid`.
- Se ha creado una función *Main* que se encarga de iniciar todo el proceso.
- Además, se han creado tres funciones adicionales para controlar el estado de los nodos y las máquina virtuales:
  - La función `buscarNodoLibre`, se encarga que detectar qué nodo es el más optimo para realizar una migración. Esta función recibe varios parámetros, entre ellos se encuentra una lista de todos los nodos físicos que existen en el sistema, el nodo actual donde se encuentra la máquina virtual solicitante, la carga actual de CPU de la máquina virtual en concreto, y por último un parámetro que indica si el nodo solicitante corresponde con el *frontend* del sistema, en cuyo caso se realizan otras tareas. De este modo, esta función, determina cuál es el nodo más óptimo para migrar la máquina virtual que lo ha solicitado. Dependiendo de si la carga actual del nodo está por encima o por debajo del umbral, esta función busca un nodo que pueda albergar la máquina virtual elegida para la migración.

En caso de no haber ningún nodo encendido, esta función también se encarga de encender un nuevo nodo.

- La función `buscarMVenNodo` se encarga de detectar las máquinas virtuales que hay en un cierto nodo físico.
- La función `elegirMaquinaVirtual` recibe como parámetros las máquinas virtuales que hay en un cierto nodo físico y en base a la carga relativa de CPU que ocupan el nodo, devuelve la segunda máquina virtual con más carga. De esta forma se evita migrar la máquina virtual con más carga en un nodo físico, ya que lo ideal es no mover máquinas virtuales con mucha carga para evitar congestión de la red además de equilibrar mejor las cargas.

Algoritmo de arbitraje

A continuación se describe el algoritmo de arbitraje utilizado, en forma de *pseudo-código*, para obtener el ahorro de energía según la carga de los nodos:

```
Mientras cierto:
    Para cada maquina virtual:
        Si se ha iniciado en el frontend:
            Buscar nodo libre óptimo
            Migrar a nodo libre

    Para cada nodo:
        Si el nodo no está apagado:
            Si carga mayor que umbral superior:
                Si contiene más de una máquina virtual:
                    Obtener lista de máquinas virtuales que contiene
                    Elegir la segunda máquina virtual con más carga
                    Buscar un nodo libre óptimo
                    Migrar máquina virtual a nodo libre
            Si carga menor que umbral inferior:
                Si contiene al menos una máquina virtual:
                    Obtener lista de máquinas virtuales que contiene
                    Para cada máquina virtual en nodo:
                        Buscar nodo libre óptimo
                        Migrar máquina virtual
                Si no contiene ninguna máquina virtual:
                    Apagar nodo
    Esperar tiempo de ciclo
```

Resumen del algoritmo de arbitraje:

Respecto a las máquinas virtuales, cuando se inician en el *frontend*, se busca un nodo libre óptimo y se migra a dicho nodo. Un nodo libre óptimo se define como aquel nodo perteneciente al sistema que puede albergar la máquina virtual seleccionada sin desperdiciar energía. Un nodo libre óptimo también puede corresponderse con un nodo apagado en caso de sobrecarga de los nodos encendidos.

Respecto a cada nodo que no esté apagado, se comprueban los umbrales superiores e inferiores cada ciclo de trabajo. Si en un determinado ciclo la carga está por encima del umbral superior, se detecta el número de máquinas virtuales que contiene dicho nodo. En caso de contener más de una, se migra a un nodo óptimo la segunda máquina virtual que más carga contenga, de esta forma se evitan sobrecargas de la red y se permite balancear mejor las cargas. En caso de que en un determinado ciclo la carga esté por debajo del umbral inferior, se intenta migrar todas las máquinas virtuales que contenga dicho nodo a otro nodo libre. Pero, en caso de no haber ningún nodo libre en ese instante, se espera al siguiente ciclo y se vuelve a comprobar el nodo. Si se detecta que no existe ninguna máquina virtual en dicho nodo, se apaga.

3.9.2. Arrancador *enersis*

Este código es el que inicia la ejecución del demonio descrito anteriormente en el *frontend*, es decir, inicia automáticamente el servicio cuando se inicia el sistema. Su funcionamiento es el mismo que el descrito en el arrancador *enersis* de los nodos, en la sección 3.8.3, pero en este caso inicia el demonio `enersis-frontend`.

3.10. Ficheros de configuración

Para el correcto funcionamiento de las librerías y demonios se deben configurar mediante los siguiente ficheros.

3.10.1. Fichero *enersis-servidor.conf*

Este fichero está situado en `/etc/enersis` en todos los nodos del *clúster*. Este archivo de configuración contiene la siguiente información:

- frontend=*ip*

  Esta línea indica la dirección IP del *frontend*, de esta forma, el demonio *enersis-servidor* sabe a qué dirección debe enviar la información de que el nodo está encendido.

3.10.2. Fichero *enersis-frontend.conf*

Este fichero está situado en `/etc/enersis` en el *frontend* y contiene información sobre los nodos del *clúster*, las máquinas virtuales creadas, el tiempo de ciclo del algoritmo de arbitraje y los umbrales de trabajo. En concreto, dicha información es la siguiente:

- umbral_superior=*[entero de 1 a 99]*

  Corresponde con el umbral superior de porcentaje de CPU que indica que un nodo está muy cargado.

- umbral_inferior=*[entero de 1 a 99]*

  Corresponde con el umbral inferior de porcentaje de CPU que indica que un nodo tiene muy poca carga.

- tiempo_ciclo=*[entero]*

  Indica el tiempo de ciclo en segundos para cada iteración del algoritmo de arbitraje.

- *[IP] [MAC] [Tiempo de CPU]*

  Cada línea de este tipo representa un nodo del *clúster*. Para cada uno de ellos se indica su dirección IP, su dirección MAC y el tiempo de CPU en segundos que se desea tener en cuenta para realizar la comprobación de su carga.

- *[Maquina Virtual]*

  Cada línea de este tipo representa una máquina virtual creada en el *frontend*. Para ello se debe indicar el nombre de cada máquina virtual creada en una línea distinta. Este nombre proviene del fichero de configuración perteneciente a cada máquina virtual.

3.11. Arquitectura del sistema

La figura 3.16 muestra un esquema de la arquitectura del sistema Enersis. En este esquema se muestra cómo se relacionan todas las librerías, demonios y otros ficheros creados para el proyecto.

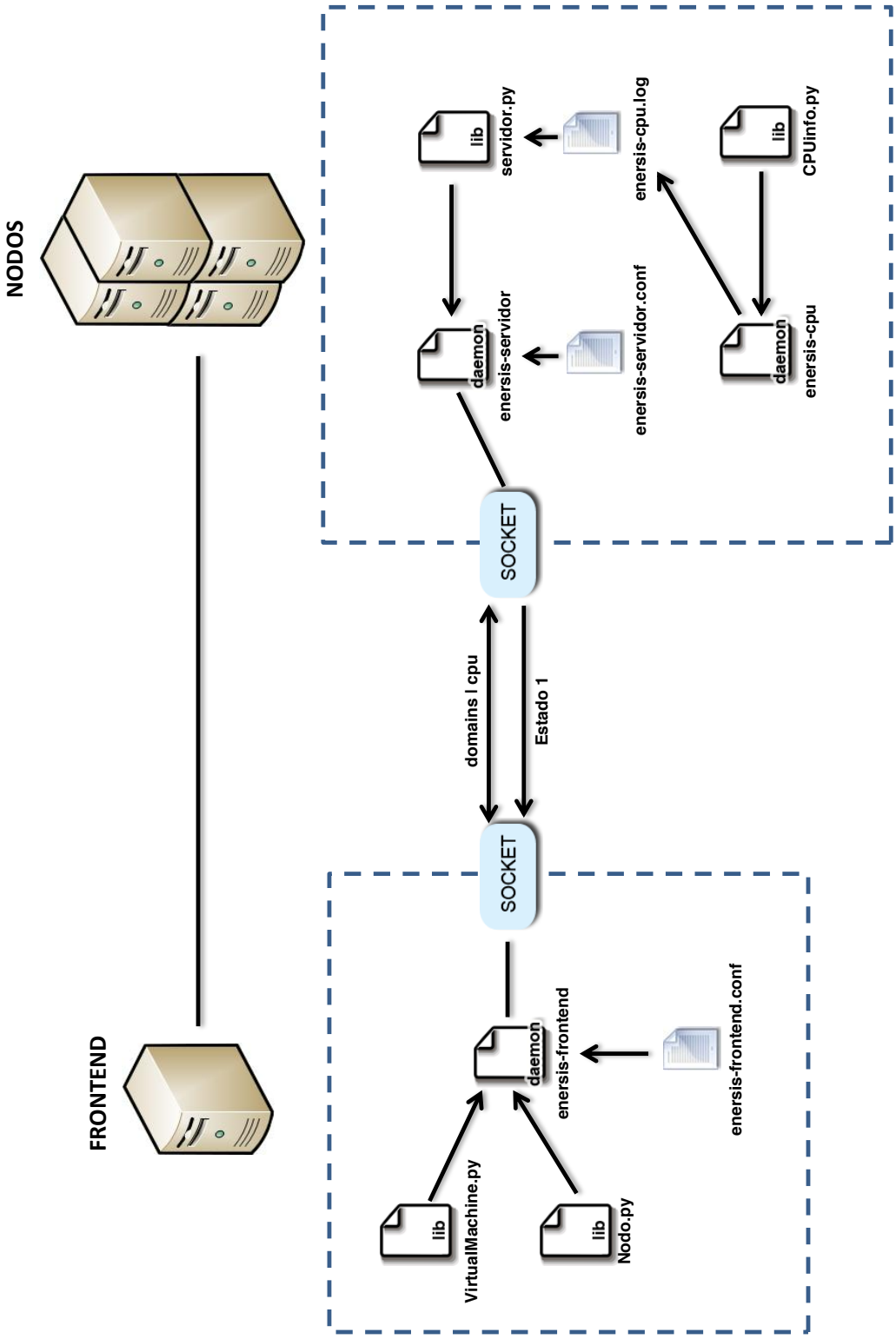


Figura 3.16: Arquitectura Enersis.

CAPÍTULO 4

Desviaciones del proyecto

Índice

4.1. Diagrama de planificación temporal realizado **57**
4.2. Costes finales del proyecto **59**

En este capítulo se muestran las desviaciones que ha sufrido el proyecto a lo largo de su ciclo de vida. Primero se muestra la planificación real que ha tenido finalmente el proyecto, y a continuación se muestran los costes reales del desarrollo del proyecto.

4.1. Diagrama de planificación temporal realizado

Durante la realización del proyecto han surgido una serie de circunstancias y problemas que han hecho que el proyecto tuviera una ligera desviación respecto a la planificación estimada en el capítulo 2. En la figura 4.1 se muestra el diagrama de Gantt de seguimiento en el que se muestra la duración real del proyecto.

Como se observa, la duración real del proyecto ha sido de 69 días, es decir 9 días más de la previsión inicial. Por lo tanto, el proyecto ha terminado el día **19 de Julio de 2012** en lugar del día **6 de Julio de 2012** como estaba previsto.

Como en la mayoría de proyectos, existen desviaciones a lo largo de su desarrollo y este ha sido un caso más. Algunas desviaciones destacadas son la elaboración del algoritmo principal de decisión, ya que era muy complicado es-

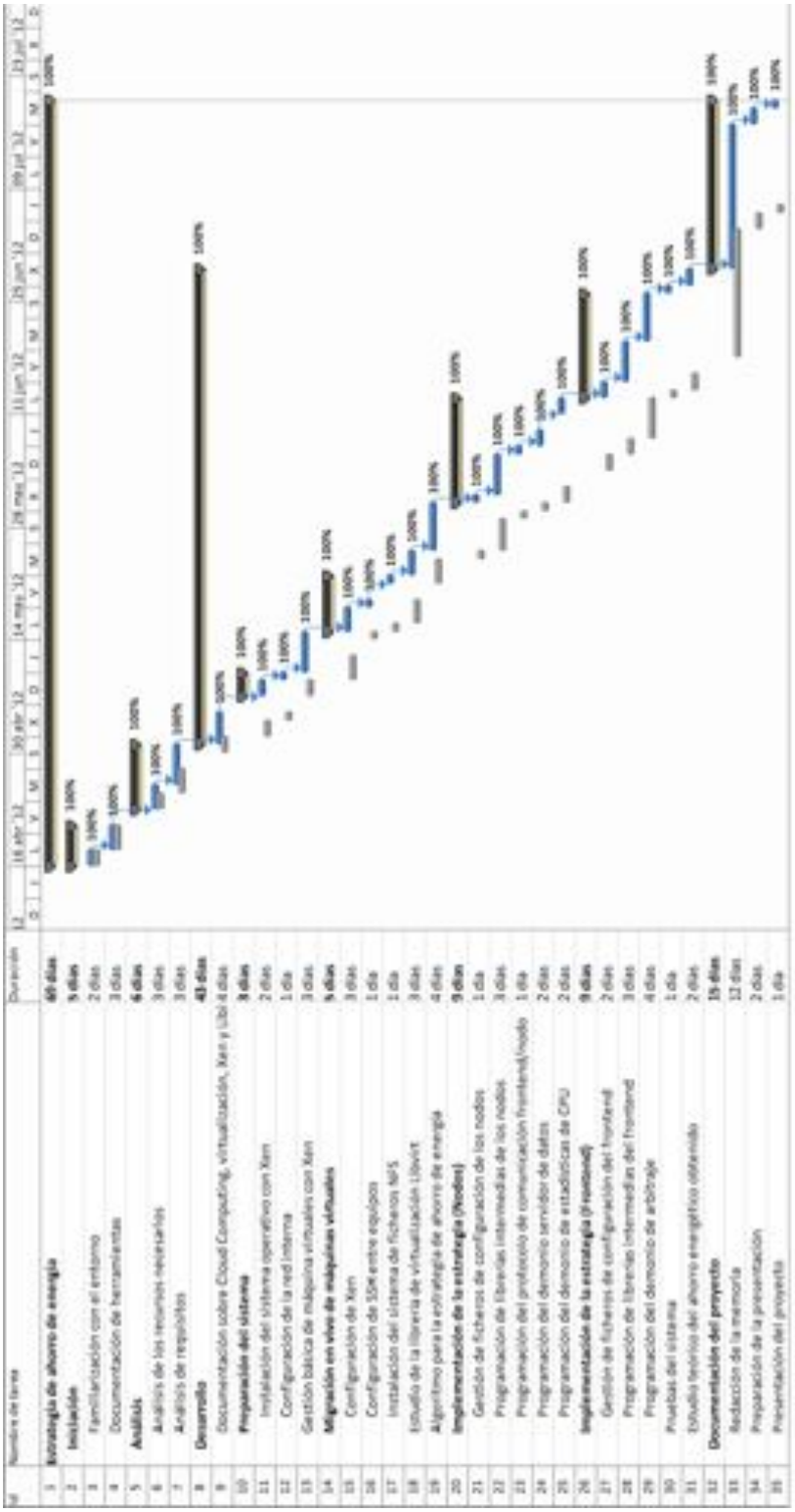


Figura 4.1: Diagrama de Gantt de seguimiento

tablecer un algoritmo capaz de aprovechar bien los recursos disponibles, con lo que se invirtió más tiempo de lo previsto en su desarrollo. Otras desviaciones han sido dadas por el desarrollo de algunas librerías del sistema, ya que para obtener la carga real de CPU de un nodo no se puede obtener simplemente consultando al sistema operativo, ya que este mismo sistema operativo es una máquina virtual más del sistema, con lo que había que consultar al hipervisor Xen para obtener los datos. Este problema condujo a un pequeña desviación respecto a la planificación inicial.

Por último, otra desviación importante ha sido el estudio teórico del ahorro energético obtenido. La asignatura de proyectos dispone de 300 horas para dedicar al proyecto, pero, como se describe anteriormente, el proyecto ha tenido una duración de 345 horas aproximadamente sin disponer de tiempo material para poder realizar el estudio teórico debido a su complejidad.

4.2. Costes finales del proyecto

Como se ha descrito en la sección anterior, el proyecto ha tenido una desviación de 9 días respecto a la planificación inicial estimada, por lo tanto también han aumentado el número de horas invertidas en él. Finalmente, realizando una jornada de trabajo de 5 horas al día (9:00 a 12:00 y de 18:00 a 20:00) se han invertido:

$$69\ \textit{días}\ \text{x}\ 5\ \textit{horas/día} = \textbf{345 horas}$$

Suponiendo que el coste por hora de un desarrollador informático es de 15€/hora y teniendo en cuenta que la duración total en horas ha sido de 345, el coste total del desarrollo del proyecto asciende a:

$$345\ \textit{horas}\ \text{x}\ 15€/\textit{hora}\ \text{x}\ 1\ \textit{desarrollador} = \textbf{5175 €}$$

CAPÍTULO 5

Conclusiones y trabajo futuro

Índice

5.1. Conclusiones **61**
5.2. Trabajo futuro **62**

En este capítulo se muestran las conclusiones del proyecto, así como sus futuras ampliaciones.

5.1. Conclusiones

Como conclusiones del proyecto se puede destacar el cumplimiento de los dos objetivos principales del proyecto.

El primer objetivo era el estudio sobre el funcionamiento de *Xen*, en el cual, después de leer mucha documentación y realizar distintas pruebas, se ha llegado a conseguir realizar un funcionamiento básico de gestión de máquinas virtuales con el hipervisor de *Xen*, además de preparar todo el sistema para conseguir realizar la migración de máquinas virtuales en vivo entre equipos remotos. Por lo tanto, este objetivo se ha conseguido de forma satisfactoria.

El segundo objetivo era la implementación de una estrategia de ahorro de energía para los *clusters* usando máquinas virtuales. Una vez estudiado el funcionamiento de *Xen*, se han buscado las herramientas y librerías adecuadas para su programación, además de realizar distintos programas de pruebas. Con ello, se han ido creando distintas librerías para conseguir que un progra-

ma principal en un *frontend* sea capaz de mantener el servicio ininterrumpido de n máquinas virtuales. Para conseguir el ahorro de energía, este programa creado se encarga de apagar y encender nodos en función de su carga, además de migrar las máquinas virtuales a cada nodo según convenga. Es decir, en función de la carga, se realiza una eficiente estrategia de ahorro de energía, en la que solo están encendidos los nodos que lo requieren. Este objetivo también se ha conseguido de forma satisfactoria.

Como valoración personal, ha sido un proyecto extenso pero a la vez muy didáctico y positivo, en el que se han aprendido muchos conceptos hasta ahora desconocidos como, por ejemplo, el mundo de la virtualización y sus distintas variantes, el nuevo concepto de Cloud Computing, aspecto muy interesante y muy útil en la actualidad, además de aprender muchos conceptos sobre sistemas operativos y Linux en general (programación de demonios, gestión de redes y puertos, SSH, NFS, etc). Por lo tanto, como valoración personal ha sido un proyecto muy positivo.

5.2. Trabajo futuro

Como trabajo futuro, este proyecto podría ser una base para crear un sistema real de ahorro de energía, preparado para ser incorporado, con leves modificaciones, a una herramienta de gestión de Cloud Computing como por ejemplo *OpenStack*, *Nimbus* o *Eucalyptus*.

La estrategia de ahorro de energía descrita se basa solamente en la carga de los nodos del *cluster* y de las máquinas virtuales usando unos umbrales de carga superior e inferior para determinar su carga. Esta estrategia se podría mejorar teniendo en cuenta muchos más aspectos a la hora de realizar la migración de las máquinas virtuales entre los nodos, como por ejemplo la carga de memoria, las transacciones de la red, etc.

Otra posible mejora sería la gestión de la estrategia de ahorro de energía, actualmente el *frontend* es el encargado de monitorizar la carga de las CPUs de los distintos nodos del *cluster* y realizar el arbitraje y las decisiones de migración. Se podría pensar en un sistema totalmente distribuido, en el que a partir de un procotolo de comunicación fueran los nodos los que se gestionaran entre ellos enviándose mensajes de estado a sus vecinos en cada caso, de esta forma se liberaría carga en el *frontend*, gestionándose los nodos de forma autónoma.

Por otra parte, otro posible trabajo futuro sería realizar un estudio teórico del posible ahorro energético que se podría obtener mediante esta estrategia.

Bibliografía

[1] *Xen*, Virtual Machine Monitor, University of Cambridge Computer Laboratory, 2003 http://www.xen.org

[2] University os California. *Rocks®Clusters*. http://www.rocksclusters.org

[3] C. R. Das, Eun Jung Kim, M. J. Irwin, M. Kandemir, Ki Hwan Yum, G. M. Link, and N. Vijaykrishnan. A holistic approach to designing energy-efficient cluster interconnects. *IEEE Transactions on Computers,* 53(6), 2005.

[4] R. P. Dick, Huazhong Yang, Li Shang, Yongpan Liuand, and H. Wang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. *Internacional 8th Symposium on Quality Electronic Design (ISQED '07),* pages 204-209, March 2007.

[5] Phinheiro, Eduardo; Bianchini, Ricardo; Carrera, Enrique V.; Heath, Taliver: Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In: *In Workshop on Compilers and Operating Systems for Low Power*, 2001

[6] Hermenier, Fabien; Lorca, Xavier; Menaud, Jean-Marc; Muller, Gilles; Lawall, Julia: Entropy: a consolidation manager for clusters. In: *VEE '09*: Proceedings of the 2009 ACM SIGPLAN/SIGOPS internacional conference on Virtual executions evironments. New York, NY, USA : ACM, 2009

[7] Wake on LAN, Wikipedia, http://en.wikipedia.org/wiki/Wake-on-LAN

[8] Rivest, Shamir y Adleman, RSA. 1997 http://en.wikipedia.org/wiki/RSA

[9] NFS (Network File System), Sun Microsystems, 1984, http://en.wikipedia.org/wiki/Network_File_System_%28protocol%29

[10] The virtualization API, Libvirt http://libvirt.org

[11] Xen Tools http://www.xen-tools.org

[12] Virtual Machine Manager, Red Hat `http://virt-manager.et.redhat.com`

[13] Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. Commun. ACM 17, 7 (July 1974), 412-421. DOI=10.1145/361011.361073 http://doi.acm.org/10.1145/361011.361073

Apéndice A

Código fuente realizado

Este apéndice muestra el código fuente desarrollado durante el proyecto. Cada fichero se ha explicado de forma detallada en el capítulo 3 de la memoria, por lo que en las distintas secciones solo se mostrará el código fuente desarrollado en cada caso.

A.1. Ficheros fuente para los nodos

A.1.1. Estructura de ficheros

Todos los ficheros fuente desarrollados en este proyecto se han colocado siguiendo la estructura de ficheros de Linux. En este caso, los ficheros creados para los nodos siguen la estructura mostrada en la figura A.1.

La estructura es la siguiente:

- **Arrancador Enersis**

  `/etc/rc.d/init.d/enersis`

- **Demonios Enersis**

  `/usr/sbin/enersis-cpu`
  `/usr/sbin/enersis-servidor`

- **Librerías adicionales**

  `/usr/lib/enersis/CPUinfo.py`
  `/usr/lib/enersis/servidor.py`

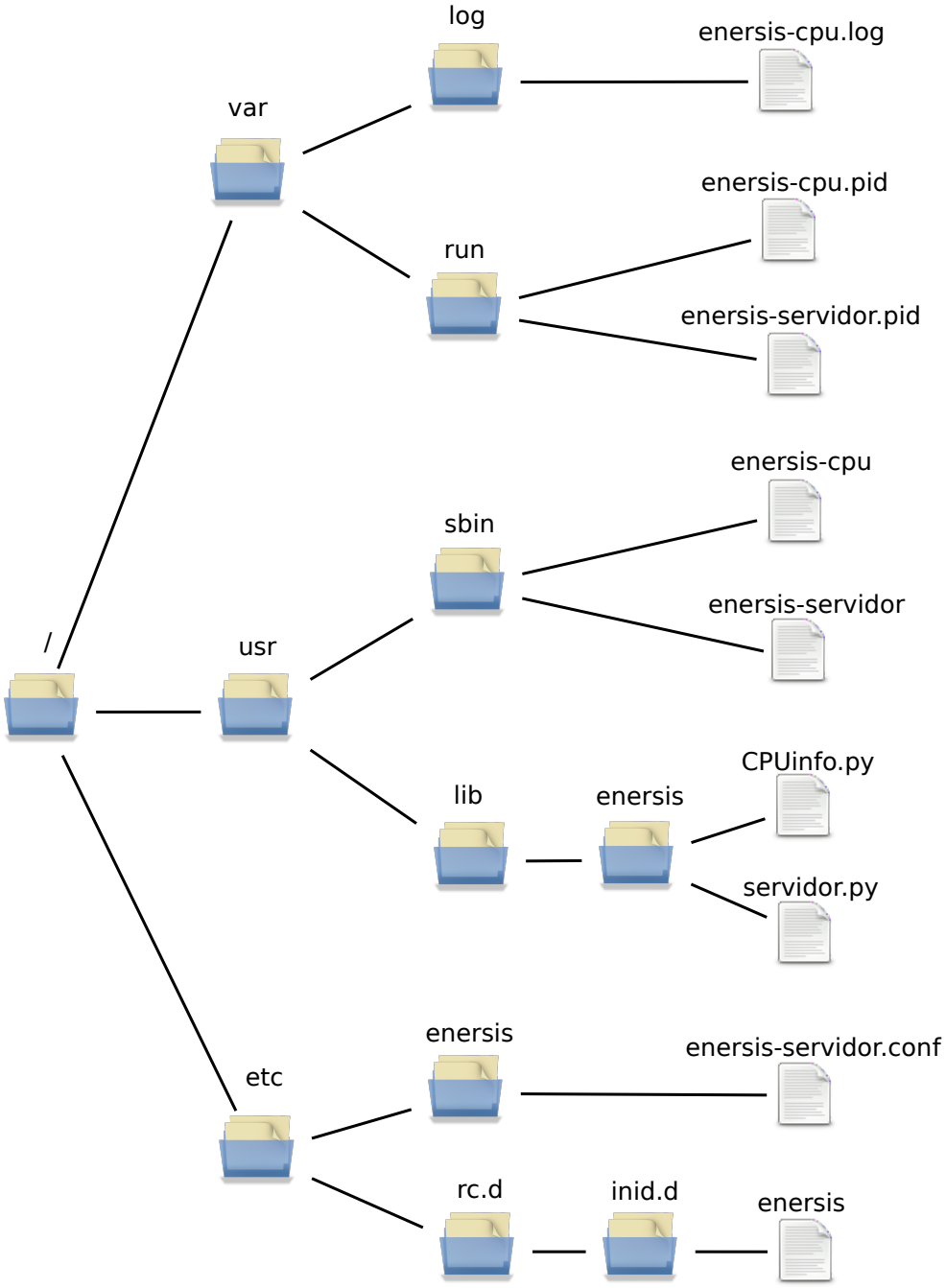


Figura A.1: Estructura de ficheros Enersis para los nodos.

- **Ficheros de configuración**

  `/etc/enersis/enersis-servidor.conf`

- **Ficheros de registro o LOG**

  `/var/log/enersis-cpu.log`

- **Ficheros PID**

  `/var/run/enersis-cpu.pid`
  `/var/run/enersis-servidor.pid`

A.1.2. Demonio *enersis-cpu*

Código Fuente A.1: Demonio *enersis-cpu*

```
#!/usr/bin/env python
import sys, time, os
sys.path.append("/usr/lib/enersis")
from CPUinfo import *
from threading import Thread
import re

class CargaCPU(Thread):

    def __init__(self, fichero):
        Thread.__init__(self)

        #Se abre el fichero para escritura al final del fichero
        self.f = open(fichero, "a")


    def run(self):

        while True:
            #Se llama a CPUinfo para obtener datos de la CPU del nodo
            cpu = CPUinfo()
            v = []

            #Se almacenan los resultados de 1s de trabajo
            #de CPU, es decir, 10 resultados de 0.1s
            for i in range(10):
                v.append(cpu.compute())

            #Se realiza la media aritmetica y
            #se formatea el resultado
            linea = "%3.4f\n" % (sum(v)/len(v))
            aux = len(linea.split(".")[0])
            if aux == 1:
                linea = "00"+linea
            elif aux == 2:
                linea = "0"+linea

            #Se escribe en el fichero LOG
            self.f.write(linea)
            self.f.flush()


def daemonize(stdout='/dev/null', stderr=None, stdin='/dev/null', \
        pidfile=None, startmsg='started with pid %s'):
    """
    do the UNIX double-fork magic, see Stevens' "Advanced
    Programming in the UNIX Environment" for details
    (ISBN 0201563177)
    http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16
    """
```

```
    try:
        pid = os.fork()
        if pid > 0:
            # exit first parent
            sys.exit(0)
    except OSError, e:
        sys.stderr.write("fork #1 failed: \
            %d (%s)\n" % (e.errno, e.strerror))
        sys.exit(1)

    # decouple from parent environment
    os.chdir("/")
    os.setsid()
    os.umask(0)

    # do second fork
    try:
        pid = os.fork()
        if pid > 0:
            # exit from second parent
            sys.exit(0)
    except OSError, e:
        sys.stderr.write("fork #2 failed: \
            %d (%s)\n" % (e.errno, e.strerror))
        sys.exit(1)

    if not stderr: stderr = stdout
    # redirect standard file descriptors
    si = file(stdin, 'r')
    so = file(stdout, 'a+')
    se = file(stderr, 'a+', 0)
    pid = str(os.getpid())
    sys.stderr.flush()

    # write pidfile
    if pidfile: file(pidfile,'w+').write("%s\n" % pid)

    os.dup2(si.fileno(), sys.stdin.fileno())
    os.dup2(so.fileno(), sys.stdout.fileno())
    os.dup2(se.fileno(), sys.stderr.fileno())


def main():

    carga = CargaCPU("/var/log/enersis-cpu.log")
    carga.start()


if __name__ == "__main__":

    daemonize(pidfile='/var/run/enersis-cpu.pid', startmsg='')
    main()
```

A.1.3. Demonio *enersis-servidor*

Código Fuente A.2: Demonio *enersis-servidor*

```
#!/usr/bin/env python

import sys, time, os, re
sys.path.append("/usr/lib/enersis")
from servidor import *


def daemonize(stdout='/dev/null', stderr=None,
        stdin='/dev/null', pidfile=None,
                startmsg='started with pid %s'):
        """
        do the UNIX double-fork magic, see Stevens' "Advanced
        Programming in the UNIX Environment" for details
        (ISBN 0201563177)
        http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16
        """
        try:
                pid = os.fork()
                if pid > 0:
                        # exit first parent
                        sys.exit(0)
        except OSError, e:
                sys.stderr.write("fork #1 failed: \
                    %d (%s)\n" % (e.errno, e.strerror))
                sys.exit(1)

        # decouple from parent environment
        os.chdir("/")
        os.setsid()
        os.umask(0)

        # do second fork
        try:
                pid = os.fork()
                if pid > 0:
                        # exit from second parent
                        sys.exit(0)
        except OSError, e:
                sys.stderr.write("fork #2 failed: \
                    %d (%s)\n" % (e.errno, e.strerror))
                sys.exit(1)

        if not stderr: stderr = stdout
        # redirect standard file descriptors
        si = file(stdin, 'r')
        so = file(stdout, 'a+')
        se = file(stderr, 'a+', 0)
        pid = str(os.getpid())
        sys.stderr.flush()
```

```
        # write pidfile
        if pidfile: file(pidfile,'w+').write("%s\n" % pid)

        os.dup2(si.fileno(), sys.stdin.fileno())
        os.dup2(so.fileno(), sys.stdout.fileno())
        os.dup2(se.fileno(), sys.stderr.fileno())

def main():

        #Fichero de configuracion enersis-servidor
        f = open("/etc/enersis/enersis-servidor.conf")
        linea =  f.readline()


        #Expresion regular de una IP
        ip = "([1-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]) \
              (\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]))"
        ip += "(\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])) \
             (\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]))$"

        if not(re.match("^frontend="+ip, linea)):
                print "Error en el archivo de configuracion"
                sys.exit(1)

        frontend = linea.split("=")[1]

        #Se envia la notificacion de encendido al frontend
        s = socket.socket()
        s.connect((frontend, socket.getservbyname("enersis")))
        s.send("1")
        s.close()


        #Se el socket de reception de peticiones
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)

        s.bind(('', socket.getservbyname("enersis")))
        s.listen(0)

        #Se realiza un bucle a la espera de nuevas conexiones
        while True:
                socket_cliente, datos_cliente = s.accept()
                hilo = Cliente(socket_cliente, datos_cliente)
                hilo.start()

if __name__ == "__main__":
        daemonize(pidfile='/var/run/enersis-servidor.pid', \
            startmsg='')
        main()
```

A.1.4. Librería *CPUinfo.py*

Código Fuente A.3: Librería *CPUinfo.py*

```
import time
import libvirt

class CPUinfo:

    def __init__(self):
        #Se abre una conexion con el hipervisor
        self.conn = libvirt.open(None)

        #Se obtiene informacion sobre el numero de CPUs del nodo
        self.numCPUs = self.conn.getInfo()[2]

    def compute(self):

        cpu = 0.0
        mvs = []
        info1 = []
        info2 = []

        #Se buscan todos los dominios activos
        for id in self.conn.listDomainsID():
            mvs.append(self.conn.lookupByID(id))

        #Se obtiene informacion de los nanosegundos de CPU
        #usados de cada dominio
        for mv in mvs:
            info1.append(mv.info()[4])

        #Se realiza una espera de 0.1s
        time.sleep(0.1)


        #Se vuelven a tomar datos sobre la CPU de los dominios
        for mv in mvs:
            info2.append(mv.info()[4])

        #Se suman las diferencias de la informacion recogida en
        # cada dominio
        for i in range(len(info1)):
            if info2[i] == 0.0:
                info1[i] = 0.0

            cpu = cpu + (info2[i]-info1[i])


        #Se obtiene el porcentaje de CPU del nodo dividiendo la suma
        #anterior por el intervalo de tiempo de datos recogido y
        #el numero de CPUs del nodo
        porcentaje = (cpu * 100.0) / (0.1*1000.0*1000.0*1000.0*self.numCPUs)
```

```
        #Se ajustan los resultados
        if porcentaje > 100.0:
            porcentaje = 100.0

        if porcentaje < 0.0:
            porcentaje = 0.0

        return porcentaje
```

A.1.5. Librería *servidor.py*

Código Fuente A.4: Librería *servidor.py*

```
import socket
import time
from CPUinfo import *
import libvirt
from threading import Thread
import copy
import re
import sys, os


class Cliente(Thread):

    def __init__(self, socket_cliente, datos_cliente):
        Thread.__init__(self)

        #Se guardan los datos sobre la conexion
        self.sc = socket_cliente
        self.addr = datos_cliente

    #Bucle para atender al cliente
    def run(self):

        seguir = True

        while seguir:

            #Espera a recibir los datos
            recibido = self.sc.recv(1024)

            #Carga de la CPU
            #Comprueba el dato recibido
            if re.match("cpu [1-9][0-9]*",recibido):
                self.tiempo = int(recibido.split(" ")[1])
                v = []

                #Abre el fichero de LOG donde se almacenan los
                #porcentajes de CPU
                fd = open("/var/log/enersis-cpu.log")

                #Mueve el puntero a los ultimos segundos pedidos
                fd.seek(-(self.tiempo*9),2)
                for linea in fd:
                    v.append(float(linea.split("\n")[0]))
                fd.close()

                #Envia la media de los resultados pedidos
                self.sc.send(str(sum(v)/len(v)))

            #Numero de dominios
            elif recibido == "domains":
```

```
            #Conecta con el hipervisor del nodo
            conn = libvirt.open(None)

            #Consulta el numero de dominios activos
            num_dom = conn.numOfDomains()

            #Envia la informacion por el socket
            self.sc.send(str(num_dom))

        #Cierra la conexion
        elif recibido == "exit":
            #Cierra la conexion con el socket
            self.sc.close()
            seguir = False
```

A.1.6. Fichero de configuración *enersis-servidor.conf*

Ejemplo de configuración:

Código Fuente A.5: Fichero de configuración *enersis-servidor.conf*

```
frontend=192.168.1.1
```

A.1.7. Arrancador *enersis*

Código Fuente A.6: Arrancador *enersis*

```
#!/bin/sh
#
# Enersis System - This script will start and stop the Enersis daemons
#
# chkconfig: 345 99 1     - start or stop process definition
# within the boot process
# description: Enersis System: Modulo de Ahorro de Energia en
# Clusters de Computadores
# processname: enersis
# pidfile: /var/run/enersis-servidor.pid, /var/run/enersis-cpu.pid

# Source function library.      This creates the operating environment
# for the process to be started

DAEMONCPU=/usr/sbin/enersis-cpu
DAEMONSERVIDOR=/usr/sbin/enersis-servidor
PIDCPU=/var/run/enersis-cpu.pid
PIDSERVIDOR=/var/run/enersis-servidor.pid

statusES(){
        local base=${1##*/}
        local pid

        pid=`ps x | grep ${base} | grep python | cut -d " " -f1 | head -n1`
        if [ -n "$pid" ]; then
                echo $"${base} (pid $pid) is running..."
                return 0
        fi

        if [ -f /var/run/${base}.pid ]; then
                read pid < /var/run/${base}.pid
                if [ -n "$pid" ]; then
                        echo $"${base} dead but pid file exists"
                        return 1
                fi
        fi

        echo $"${base} is stopped"

}


case "$1" in
  start)
        echo -n "Starting Enersis daemons: \n"
        if [ -x $DAEMONSERVIDOR ] ; then
        echo -n "Starting enersis-servidor: \n"
        start-stop-daemon --start --pidfile $PIDSERVIDOR --exec \
        $DAEMONSERVIDOR &
        fi
```

```
        if [ -x $DAEMONCPU ] ; then
        echo -n "Starting enersis-cpu: \n"
        start-stop-daemon --start --pidfile $PIDCPU --exec $DAEMONCPU &
        fi

        ;;
  stop)
        echo -n "Shutting down Enersis daemon enersis-servidor: \n"
        start-stop-daemon --stop --oknodo --pidfile $PIDSERVIDOR  \
        --name python
        echo
        rm -f /var/run/enersis-servidor.pid

        echo -n "Shutting down Enersis daemon enersis-cpu: \n"
        start-stop-daemon --stop --pidfile $PIDCPU --name python
        echo
        rm -f /var/run/enersis-cpu.pid

        ;;
  status)
        statusES enersis-servidor
        statusES enersis-cpu
        ;;
  restart)
        echo "Restarting Enersis"
        $0 stop
        $0 start
        echo "done."
        ;;
  *)
        echo "Usage: $0 {start|stop|restart|status}"
        exit 1
esac

exit 0
```

A.2. Ficheros fuente para el *frontend*

A.2.1. Estructura de ficheros

Todos los ficheros fuente desarrollados en este proyecto se han colocado siguiendo la estructura de ficheros de Linux. En este caso, los ficheros desarrollados para el *frontend* siguen la estructura mostrada en la figura A.2.

La estructura es la siguiente:

- **Arrancador Enersis**

  `/etc/rc.d/init.d/enersis`

- **Demonios Enersis**

  `/usr/sbin/enersis-frontend`

- **Librerías adicionales**

  `/usr/lib/enersis/Nodo.py`
  `/usr/lib/enersis/VirtualMachine.py`

- **Ficheros de configuración**

  `/etc/enersis/enersis-frontend.conf`

- **Ficheros de registro o LOG**

  `/var/log/enersis-frontend.log`

- **Ficheros PID**

  `/var/run/enersis-frontend.pid`

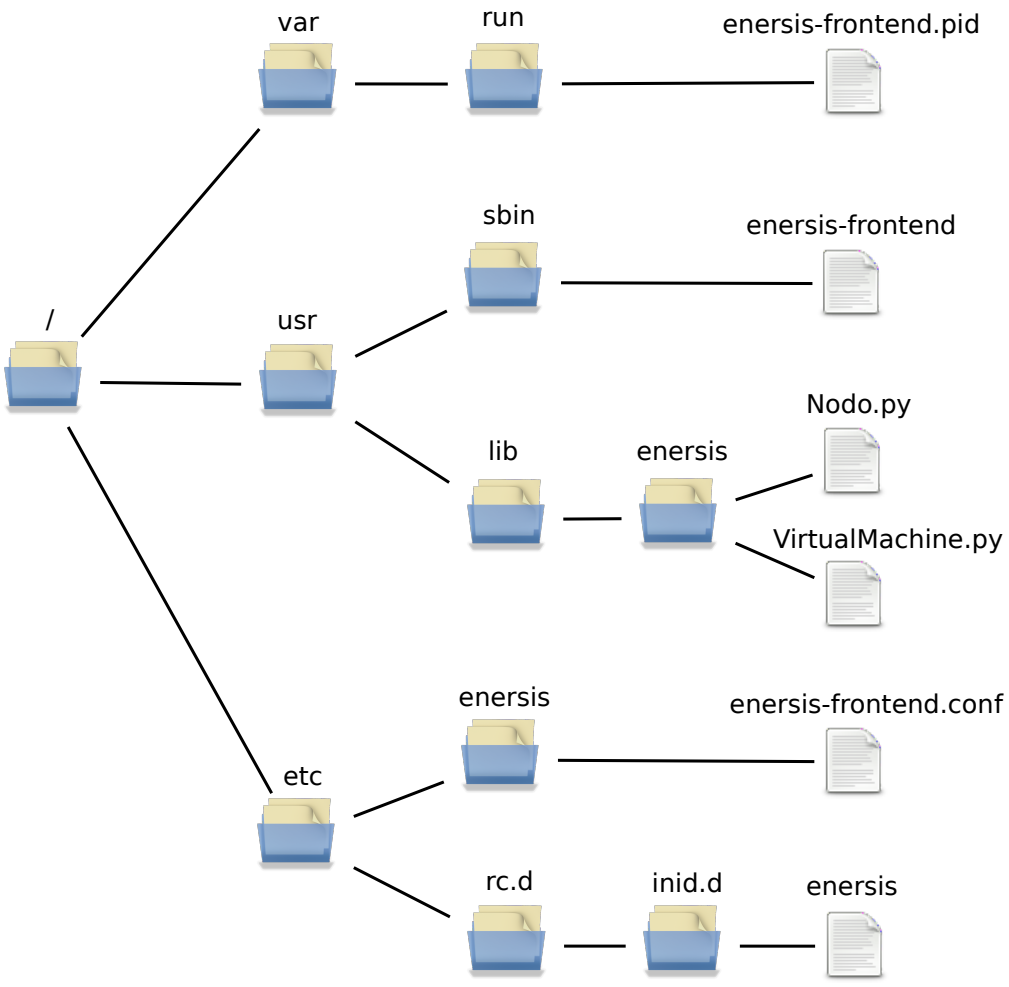


Figura A.2: Estructura de ficheros Enersis para el *frontend*.

A.2.2. Demonio *enersis-frontend*

Código Fuente A.7: Demonio *enersis-frontend*

```
import libvirt
import socket
import time
import sys
sys.path.append("/usr/lib/enersis")
from Nodo import *
from VirtualMachine import *
import re
import random

ficheroConf = "/etc/enersis/enersis-frontend.conf"

umbralSup = 0.0
umbralInf = 0.0
tiempoCiclo = 0

def daemonize(stdout='/dev/null', stderr=None, stdin='/dev/null', \
pidfile=None, startmsg='started with pid %s'):
        """
        do the UNIX double-fork magic, see Stevens' "Advanced
        Programming in the UNIX Environment" for details (ISBN 0201563177)
        http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16
        """
        try:
                pid = os.fork()
                if pid > 0:
                        # exit first parent
                        sys.exit(0)
        except OSError, e:
                sys.stderr.write("fork #1 failed: %d (%s)\n" % (e.errno, \
                 e.strerror))
                sys.exit(1)

        # decouple from parent environment
        os.chdir("/")
        os.setsid()
        os.umask(0)

        # do second fork
        try:
                pid = os.fork()
                if pid > 0:
                        # exit from second parent
                        sys.exit(0)
        except OSError, e:
                sys.stderr.write("fork #2 failed: %d (%s)\n" % (e.errno, \
                e.strerror))
                sys.exit(1)
```

```
        if not stderr: stderr = stdout
        # redirect standard file descriptors
        si = file(stdin, 'r')
        so = file(stdout, 'a+')
        se = file(stderr, 'a+', 0)
        pid = str(os.getpid())
        sys.stderr.flush()

        # write pidfile
        if pidfile: file(pidfile, 'w+').write("%s\n" % pid)

        os.dup2(si.fileno(), sys.stdin.fileno())
        os.dup2(so.fileno(), sys.stdout.fileno())
        os.dup2(se.fileno(), sys.stderr.fileno())



def buscarMVenNodo(maquinasVirtuales, nodo):

        mvEnNodo = []
        for mv in maquinasVirtuales:
                if mv.nodo == nodo.ip:
                        mvEnNodo.append(mv)
        return mvEnNodo


def compruebaConf():

        ip = "^([1-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]) \
        (\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]))"
        ip += "(\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]))\
        (\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]))$"

        mac = "^[0-9A-F][0-9A-F]:[0-9A-F][0-9A-F]:[0-9A-F][0-9A-F]: \
        [0-9A-F][0-9A-F]:[0-9A-F][0-9A-F]:[0-9A-F][0-9A-F]$"

        f = open(ficheroConf)

        linea = f.readline()
        if linea != "[PARAMETROS]\n":
                return 1
        if not(re.match("^umbral_superior=[1-9][0-9]?\n$", f.readline())):
                return 2
        if not(re.match("^umbral_inferior=[1-9][0-9]?\n$", f.readline())):
                return 3
        if not(re.match("^tiempo_ciclo=[1-9][0-9]*\n$", f.readline())):
                return 4
        if f.readline() != "[NODOS]\n":
                return 5

        linea = f.readline()
        linea = linea.split(" ")
        if len(linea) != 3:
                return 6
```

```
        if not(re.match(ip, linea[0])):
                return 7
        if not(re.match(mac, linea[1])):
                return 8
        if not(re.match("^[1-9][0-9]*\n$", linea[2])):
                return 9

        linea = f.readline()
        while(linea != "[MV]\n"):
                linea = linea.split(" ")
                if len(linea) != 3:
                        return 10
                if not(re.match(ip, linea[0])):
                        return 11
                if not(re.match(mac, linea[1])):
                        return 12
                if not(re.match("^[1-9][0-9]*\n$", linea[2])):
                        return 13
                linea = f.readline()

        linea = f.readline()
        if not(re.match("^[0-9A-Za-z]+\n$", linea)):
                return 14

        linea = f.readline()
        while linea != "[END]\n":
                if not(re.match("^[0-9A-Za-z]+\n$", linea)):
                        return 15
                linea = f.readline()

        if f.readline() != "":
                return 16

        f.close()
        return 0


def cargaConf():

        global umbralSup, umbralInf, tiempoCiclo

        nodos = []
        maquinasVirtuales = []

         #Se cargan los datos del fichero de configuracion
        f = open(ficheroConf)
        f.readline()
        umbralSup = float(f.readline().split("=")[1])
        umbralInf = float(f.readline().split("=")[1])
        tiempoCiclo = int(f.readline().split("=")[1])

        f.readline()

        n = f.readline().split(" ")
```

```
        nodos.append(Nodo(n[0], n[1], int(n[2])))

        linea = f.readline()
        while linea != "[MV]\n":
                n = linea.split(" ")
                nodos.append(Nodo(n[0], n[1], int(n[2])))
                linea = f.readline()

        nombreMV = f.readline().split("\n")[0]
        maquinasVirtuales.append(VirtualMachine(nombreMV))
        linea = f.readline()

        while linea != "[END]\n":
                nombreMV = linea.split("\n")[0]
                maquinasVirtuales.append(VirtualMachine(nombreMV))
                linea = f.readline()

        f.close()

        return nodos, maquinasVirtuales



# Se hace una lista con las CPUs relativas que ocupan en el nodo las
# maquina virtuales y se elige la segunda que mas carga
def elegirMaquinaVirtual(mvs):

        l = []
        for mv in mvs:
                l.append((mv, mv.cpuRelativa()))

        ordenCarga = sorted(l, key=lambda carga: carga[1])
        return ordenCarga[len(l) - 2][0]

def buscarNodoLibre(nodos, nodoActual, cargaMV, esFrontend=0):

    global umbralSup, umbralInf

    apagados = []
    disponibles = []

    if esFrontend:
       for n in nodos:
           if n.estado:
               if n.carga() < umbralSup:
                   disponibles.append((n, n.carga()))
           else:
               apagados.append(n)

       if len(disponibles) != 0:
           ordenNodosDisponibles = sorted(disponibles, \
            key=lambda disp: disp[1])
           aux = ordenNodosDisponibles[0][0]
           return aux
```

```
        if len(apagados) != 0:
            aux = apagados[random.randrange(0, len(apagados), 1)]
            aux.encender()
            return aux


    if nodoActual.carga() > umbralSup:
        for n in nodos:
            if n.ip != nodoActual.ip:
                if n.estado:
                    if (n.carga() + cargaMV) < umbralSup:
                        disponibles.append((n, n.carga()))
                else:
                    apagados.append(n)

        if len(disponibles) != 0:
            ordenNodosDisponibles = sorted(disponibles, \
              key=lambda disp: disp[1])
            aux = ordenNodosDisponibles[0][0]
            return aux

        if len(apagados) != 0:
            aux = apagados[random.randrange(0, len(apagados), 1)]
            aux.encender()
            return aux


    if nodoActual.carga() < umbralInf:
        for n in nodos:
            if n.ip != nodoActual.ip:
                if n.estado:
                    if (n.carga() + cargaMV) < umbralSup:
                        disponibles.append((n, n.carga()))

        if len(disponibles) != 0:
            ordenNodosDisponibles = sorted(disponibles, \
              key=lambda disp: disp[1])
            aux = ordenNodosDisponibles[0][0]
            return aux

    return 0


def main():

        global umbralSup, umbralInf, tiempoCiclo

        if compruebaConf() != 0:
                print "Error en el archivo de configuracion"
                sys.exit(1)


        nodos, maquinasVirtuales = cargaConf()
```

```
        frontEnd = libvirt.open(None)

        #Programa principal
        while True:

            #Comprobamos si se ha arrancado una nueva maquina
        for mv in maquinasVirtuales:
            if mv.enFrontEnd(frontEnd):
                nodoLibre = buscarNodoLibre(nodos, '', mv.cpuRelativa(), 1)
                time.sleep(3)
                mv.migrar(nodoLibre)


        #Comprobamos carga en los nodos
        for nodo in nodos:
            if nodo.estado != 0: #Comprobamos si esta encendido

                #Carga Superior del nodo
                if nodo.carga() > umbralSup:
                    if nodo.dominios() > 2:
                        mvsEnNodo = buscarMVenNodo(maquinasVirtuales, nodo)

                        #Elegir maquina virtual a migrar
                        mVirtualElegida = mvsEnNodo[0]
                        mVirtualElegida = elegirMaquinaVirtual(mvsEnNodo)
                        nodoLibre = buscarNodoLibre(nodos, nodo, \
                           mVirtualElegida.cpuRelativa())
                        if nodoLibre != 0:
                            mVirtualElegida.migrar(nodoLibre)

                # Carga Inferior del nodo
                if nodo.carga() < umbralInf:
                    if nodo.dominios() > 1:
                        mvsEnNodo = buscarMVenNodo(maquinasVirtuales, nodo)

                        for mv in mvsEnNodo:
                            nodoLibre = buscarNodoLibre(nodos, nodo, \
                                mv.cpuRelativa())
                            if nodoLibre != 0:
                                mv.migrar(nodoLibre)

                    # Si se han migrado todas las maquinas virtuales
                    # se apaga el nodo
                    else:
                        nodo.apagar()

        time.sleep(tiempoCiclo)

if __name__ == "__main__":
        daemonize(pidfile='/var/run/enersis-frontend.pid', startmsg='')
        main()
```

A.2.3. Librería *Nodo.py*

Código Fuente A.8: Librería *Nodo.py*

```
import libvirt
import os
import socket
import time

class Nodo:

    def __init__(self, ip, mac, tiempocpu):

        self.ip = ip #IP del nodo
        self.mac = mac #Direccion MAC de la tarjeta de red del nodo
        self.tiempocpu = tiempocpu #Tiempo de CPU para obtener la carga
        self.con = -1  #Estado de la conexion
        self.estado = 0 #Estado del nodo, 0 apagado, 1 encendido


    def encender(self): #Enciende el nodo

        #Envia un paquete magico al nodo para encenderlo
        os.system("ether-wake -i eth0 %s" % self.mac)


        #Inicia un socket a la espera de recibir confirmacion del nodo
        #para determinar que se ha encendido
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
        s.bind(('', socket.getservbyname("enersis")))
        s.listen(0)

        sock, datos = s.accept()
        aux = int(sock.recv(1024))

        #Realiza una espere para que se
        #terminen de arrancar los servicios del nodo
        time.sleep(10)

        #Conecta con su hipervisor
        self.conectar()

        #Actualiza el estado
        self.estado = 1

        #Cierra las conexiones del socket
        sock.close()
        s.close()
        return self.estado


    def apagar(self): #Apaga el nodo
```

```
        #Desconecta el socket creado
        self.desconectar()

        #Envia el comando de apagado
        os.system("ssh %s shutdown -h now" % self.ip)

        #Espera a que se apague realmente el nodo
        time.sleep(35)

        #Actualiza el estado
        self.estado = 0

    def conectar(self):  #Conecta con el hipervisor del nodo

        try:
            #Conecta con el hipervisor
            self.con = libvirt.open("xen+ssh://"+self.ip)

            #Abre un socket para enviarle peticiones
            self.s = socket.socket()
            self.s.connect((self.ip, socket.getservbyname("enersis")))
            return 0
        except:
            return -1

    def desconectar(self): #Desconecta

        #Envia la peticion de desconectar
        self.s.send("exit")

        #Cierra el socket
        self.s.close()

    def carga(self):

        #Realiza la peticion obtener el porcentaje de cpu en
        # del tiempo indicado
        self.s.send("cpu "+str(self.tiempocpu))
        return float(self.s.recv(1024))

    def dominios(self):

        #Realiza la peticion de obtener el numero de dominios
        self.s.send("domains")
        return self.s.recv(1024)
```

A.2.4. Librería *VirtualMachine.py*

Código Fuente A.9: Librería *VirtualMachine.py*

```
import libvirt
from Nodo import *

class VirtualMachine:

    def __init__(self, maquina, estado=0, nodo="192.168.1.1"):


        self.nombre = maquina #Nombre del dominio
        self.estado = estado #0 apagada, 1 encendida
        self.nodo =  nodo #IP del nodo que contiene el dominio

        #Se conecta con el hipervisor del nodo
        self.conexionNodo = libvirt.open("xen+ssh://"+self.nodo)

        #Conecta con el dominio
        self.con = self.conexionNodo.lookupByName(self.nombre)

    def migrar(self, nodoDest):

        # nodoDest indica el nodo destino
        # Se realiza la migracion al nodo destino en modo LIVE,
        # es decir, sin pausar el dominio

        self.con.migrate(nodoDest.con, libvirt.VIR_MIGRATE_LIVE, None, \
          nodoDest.ip, 0)

        #Se actualizan los nuevos datos
        self.nodo = nodoDest.ip
        self.con = nodoDest.con.lookupByName(self.nombre)


    def enFrontEnd(self, connFrontEnd):

        #Se recibe como parametro la conexion al frontend
        #Se realiza la conexion del dominio con el frontend
        mv = connFrontEnd.lookupByName(self.nombre)

        #Se comprueba si esta activo el domino
        estado =  mv.isActive()
        if estado:
            #Se actualiza el estado
            self.estado = 1
        return estado

    def cpuRelativa(self):

        info1 = self.con.info()[4]
        #Se realiza una espera de 0.1s
```

```
        time.sleep(0.1)
        info2 = self.con.info()[4]
        cpu = info2-info1
        numCPUs = self.conexionNodo.getInfo()[2]

        # Se obtiene el porcentaje de CPU del nodo dividiendo la
        # suma anterior por el intervalo de tiempo de datos recogido
        # sy el numero de CPUs del nodo
        porcentaje = (cpu * 100.0) / (0.1*1000.0*1000.0*1000.0*numCPUs)


        #Se ajustan los resultados
        if porcentaje > 100.0:
            porcentaje = 100.0

        if porcentaje < 0.0:
            porcentaje = 0.0

        return porcentaje
```

A.2.5. Fichero de configuración *enersis-frontend.conf*

Ejemplo de configuración:

Código Fuente A.10: Fichero de configuración *enersis-frontend.conf*

```
[PARAMETROS]
umbral_superior=80
umbral_inferior=10
tiempo_ciclo=10
[NODOS]
192.168.1.2 D4:85:64:38:BE:CC 10
192.168.1.3 D4:85:64:38:BC:00 10
[MV]
mv1
mv2
mv3
mv4
[END]
```

A.2.6. Arrancador *enersis*

Código Fuente A.11: Arrancador *enersis*

```
#!/bin/sh
#
# Enersis System - This script will start and stop the Enersis daemons
#
# chkconfig: 345 99 1     - start or stop process
#                                  definition within the boot process
# description: Enersis System: Modulo de Ahorro de Energia en
#                  Clusters de Computadores
# processname: enersis
# pidfile: /var/run/enersis-frontend.pid

# Source function library.      This creates the operating
# environment for the process to be started
. /etc/rc.d/init.d/functions

statusES(){
        local base=${1##*/}
        local pid

        pid=`ps x | grep ${base} | grep python | cut -d " " -f1 | head -n1`
        if [ -n "$pid" ]; then
                echo $"${base} (pid $pid) is running..."
                return 0
        fi

        if [ -f /var/run/${base}.pid ]; then
                read pid < /var/run/${base}.pid
                if [ -n "$pid" ]; then
                        echo $"${base} dead but pid file exists"
                        return 1
                fi
        fi

        echo $"${base} is stopped"

}

case "$1" in
  start)
        echo -n "Starting Enersis daemons: "
        if [ -x /usr/sbin/enersis-frontend ] ; then
        echo -n "Starting enersis-frontend: "
        daemon  enersis-frontend
        echo
        touch /var/lock/subsys/enersis-servidor
        fi


        ;;
  stop)
```

```
        echo -n "Shutting down Enersis daemon enersis-frontend: "
        killproc enersis-frontend
        echo
        rm -f /var/lock/subsys/enersis-frontend
        rm -f /var/run/enersis-frontend.pid


        ;;
  status)
        statusES enersis-frontend
        ;;
  restart)
                echo "Restarting Enersis"
        $0 stop
        $0 start
        echo "done."
        ;;
  *)
        echo "Usage: $0 {start|stop|restart|status}"
        exit 1
esac

exit 0
```