DESARROLLO DE INTERFACES DE ALTO NIVEL PARA LA INTERACCIÓN CON GESTORES DE MÁQUINAS VIRTUALES EN INFRAESTRUCTURAS DE TIPO CLOUD

por

Amanda Calatrava Arroyo

Escuela Técnica Superior de
Ingeniería Informática

2010

Director: Dr. D. Germán Moltó Martínez

Fecha: Septiembre 2010

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

INFORMÁTICA

RESUMEN

Desarrollo de Interfaces de Alto Nivel para la Interacción con Gestores de Máquinas Virtuales en Infraestructuras de tipo Cloud

por Amanda Calatrava Arroyo

En este Proyecto Final de Carrera se ha desarrollado una interfaz Java para la interacción con el Gestor de Máquinas Virtuales (GMVs) OpenNebula. Este API ofrece compatibilidad uno a uno con todas las operaciones ofrecidas por el GMVs. Para ello, internamente se realizan llamadas basadas en XML-RPC para la interacción con el servidor. Permite procesar las respuestas basadas en XML de OpenNebula y ofrecer métodos sencillos para acceder a esta información con el objetivo de facilitar la labor del programador. Además, incluye operaciones para el despliegue y gestión de clusters de máquinas virtuales, facilitando aún más la labor del programador que desarrolle aplicaciones que interactúen con el GMVs. Cabe destacar que este API ofrece mecanismos para detectar los problemas que surjan durante las invocaciones remotas.

Escuela Técnica Superior de Ingeniería **Informática** etsinf

AGRADECIMIENTOS

Deseo expresar mi más sincero agradecimiento al profesor Germán Moltó, quien ha sido mi tutor durante estos meses, por la oportunidad ofrecida para la realización de este proyecto en el ámbito de investigación, por su colaboración en la preparación del mismo y por adentrarme en un campo de la informática que desconocía y que he podido comprobar lo extraordinario que puede llegar a ser. Además, manifiesto mi agradecimiento a los profesores que he tenido oportunidad de conocer durante la carrera, los cuales me han aportado numerosos conocimientos, y gracias a ellos he logrado alcanzar mis objetivos satisfactoriamente.

Gracias también a mis familiares y amigos, y más concretamente a mis padres, por el apoyo recibido a lo largo de mi carrera y en especial en estos últimos meses, pues sin él hubiese sido más complicado alcanzar el nivel, tanto personal como profesional, en el que me encuentro actualmente.

Índice general

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 3
1.3. Estructura de la memoria 4
1.4. Conceptos iniciales 5
1.4.1. Cloud Computing 5
1.4.2. Gestores de máquinas virtuales 8
1.4.3. Hipervisores 9

2. Tecnologías Empleadas **12**
2.1. Java 12
2.2. Eclipse Galileo (IDE) 12
2.3. Subversion 13
2.4. XML-RPC 14
2.5. XPath 15
2.6. Log4j 17
2.7. OpenNebula 19

3. Desarrollo del API **22**
3.1. Aspectos generales del API 22
3.1.1. Esquema de clases 22
3.1.2. La excepción CloudException 26
3.1.3. La clase CloudOneResult 24
3.1.4. El método *extractResult* 30

3.2. Operaciones destinadas a la gestión de máquinas virtuales 31
3.2.1. Operaciones disponibles en la clase *OpenNebulaClient* 32
3.2.2. El objeto de alto nivel *VMInfoResult* 37
3.2.3. La clase *ONEVMTemplate* 39

3.3. Operaciones destinadas a la gestión de Hosts 41
3.3.1. Operaciones disponibles en la clase *OpenNebulaClient* 41
3.3.2. El objeto de alto nivel *HostInfoResult* 43

3.4. Operaciones destinadas a la gestión de redes virtuales 45
3.4.1. Operaciones disponibles en la clase *OpenNebulaClient* 45
3.4.2. El objeto de alto nivel *VNInfoResult* 48

3.5. Operaciones destinadas a la gestión usuarios 48
3.5.1. Operaciones disponibles en la clase *OpenNebulaClient* 49
3.5.2. El objeto de alto nivel *UserInfoResult* 50

3.6. Operaciones destinadas a la gestión de clusters de MVs 51

3.6.1. Operaciones disponibles en la clase *OpenNebulaClient* 51
3.6.2. La clase *OneVMCluster* 54

4. Conclusiones **55**

Anexos **58**

A. Descripción de un Caso de Estudio **58**

B. Instalación de Subversion en Eclipse Galileo **61**

C. Ejemplos de resultados XML devueltos por OpenNebula **62**

Bibliografía y referencias **65**

LISTA DE TABLAS

Número *Página*

3.1. Métodos destinados a la gestión de MVs 32
3.2. Métodos disponibles en la clase *VMInfoResult* 38
3.3. Atributos de un objeto *ONEVMTemplate*..40
3.4. Métodos destinados a la gestión de Hosts 43
3.5. Métodos disponibles en la clase *HostInfoResult* 44
3.6. Métodos destinados a la gestión de VNs 46
3.7. Métodos disponibles en la clase *VNInfoResult*...48
3.8. Métodos destinados a la gestión de usuarios 49
3.9. Métodos disponibles en la clase *UserInfoResult*...51
3.10. Métodos destinados a la gestión de clusters de MVs...................................52

LISTA DE FIGURAS

Número *Página*

1.1. Distintas Máquinas Virtuales ejecutándose sobre un único recurso físico ...1
1.2. Esquema de funcionamiento de una nube 5
1.3. Arquitectura del Cloud Computing 6
1.4. Esquema de localización de un GMVs 9
1.5. Arquitectura de trabajo de un Hipervisor 10
1.6. Diferentes tipos de Hipervisores 11
2.1. Estructura de árbol de Subversion 14
2.2. Esquema de trabajo del protocolo XML-RPC 15
2.3. Ejemplo de fichero XML y su representación en forma de árbol 16
2.4. Ejemplo de uso de Log4J 18
2.5. Ciclo de vida de una MV 21
3.1. Estructura del paquete *opennebula* 24
3.2. Estructura del paquete *result* 25
3.3. Estructura del paquete *test* 26
3.4. Esquema de las excepciones en Java 27
3.5. Ejemplo de expresiones válidas para XPath 29
3.6. Ejemplo de *template* válido para la creación de una MV 40
A. Salida obtenida tras la ejecución del caso de estudio 60

1. INTRODUCCIÓN

En esta primera sección de la memoria se describen los motivos que han llevado a realizar este proyecto y los objetivos a alcanzar. También incluye una breve explicación de los primeros conceptos considerados de interés para poder situarse de forma correcta en el ámbito del proyecto.

1.1. MOTIVACIÓN

El auge de las técnicas de virtualización durante los últimos años ha permitido el surgimiento de nuevos escenarios de computación que pueden ayudar, de forma considerable, a la computación científica. Uno de los principales motivos por el cual estas técnicas juegan un papel tan relevante en el campo científico es la posibilidad de desplegar de forma concurrente distintas Máquinas Virtuales (MVs) capaces de proporcionar un entorno de ejecución para las aplicaciones independiente de la plataforma de hardware y del sistema operativo. De esta forma se facilita la tarea de satisfacer los requisitos concretos que plantean este tipo de aplicaciones, tales como sistemas operativos determinados, versiones de software concretas o dependencias externas con librerías numéricas.

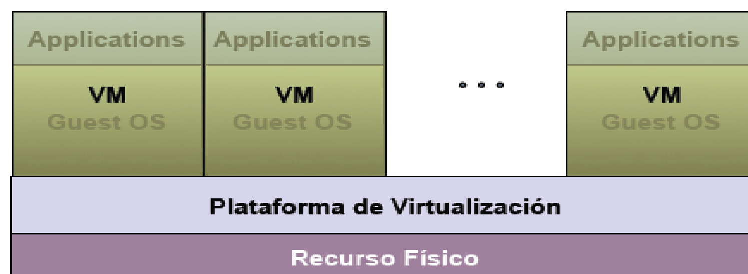

Figura 1.1 – Distintas Máquinas Virtuales ejecutándose sobre un único recurso físico

La computación basada en Cloud utiliza estas técnicas de virtualización para poder crear infraestructuras capaces de acoplarse perfectamente a los requisitos de ejecución de sus usuarios. Bajo el término de Cloud Computing [1] (computación en la nube) se encuentra otro que caracteriza a este tipo de computación, la elasticidad, ya que esta infraestructura funciona bajo demanda, es decir, puede crecer o decrecer en el momento en el que se considere oportuno mediante el despliegue o la finalización de MVs, todo ello de forma automática. Esta capacidad que proporcionan los entornos Cloud es la que le ha hecho ser hoy en día, uno de los términos más utilizados en el campo de la computación.

Para poder gestionar de forma eficiente las MVs sobre una infraestructura física, se precisan unos componentes denominados Gestores de Máquinas Virtuales (GMVs) capaces de desplegar y monitorizar las MVs a petición del usuario. Funcionan como software intermedio entre el usuario y la infraestructura física disponible, dando lugar a la aparición una de nueva capa de abstracción que separa la gestión del servicio que se le presta al usuario de la gestión de los recursos físicos, todo ello de forma transparente al usuario. Entre los más destacados se encuentra OpenNebula [2]. Este gestor, que se estudiará más profundamente en los próximos capítulos, le brinda al usuario la oportunidad de desplegar una MV especificándole todos los detalles de creación y se encarga de ponerla en marcha sobre la infraestructura física existente.

Generalmente, la forma más común de comunicarse con los GMVs es a través de la línea de comandos. Este tipo de interfaz, aunque cuenta con ventajas tales como la potencia y el control que le brinda al usuario, se convierte en una forma incómoda de interactuar con el gestor a través de un lenguaje de programación.

En el caso concreto de OpenNebula, se puede observar que ofrece interfaces para lenguajes como Ruby [3] o Phyton [4], pero la interfaz que ofrecía hasta principios de marzo de 2010 para el lenguaje de programación Java era insuficiente. Por este motivo se planteó realizar el proyecto final de carrera en este ámbito.

A principios de abril del presente año aparece una nueva interfaz de programación para este lenguaje [5], pero nuevamente presenta carencias y aspectos mejorables. Por ello, el principal trabajo de este proyecto se centra en crear un API (*Application Programming Interface,* interfaz de programación de aplicaciones) que mejore al existente y que proporcione nuevas funcionalidades como por ejemplo, la gestión de clusters de MVs y el procesado de la información resultante de las operaciones de OpenNebula. El desarrollo de este API facilitará la creación de aplicaciones Java que requieran el despliegue de MVs vía OpenNebula.

1.2. OBJETIVOS

En este proyecto se pretende construir una interfaz de programación Java para la interacción con el Gestor de Máquinas Virtuales OpenNebula. Aprovechando que este GMVs proporciona soporte para realizar llamadas basadas en XML-RPC [6] al servidor, ya que la arquitectura que presenta es de tipo cliente-servidor, se desarrollará en primera instancia un API capaz de ofrecer soporte a todas y cada una de las operaciones que ofrece OpenNebula.

Posteriormente, se implementará una interfaz de alto nivel que, utilizando el API creado anteriormente, ofrezca métodos más sencillos para el despliegue de máquinas virtuales. Estos métodos serán capaces de devolver objetos de alto nivel para facilitar el acceso a la información resultante de las operaciones de OpenNebula. A través de estos objetos se podrá acceder de forma sencilla a todos los atributos incluidos en los documentos XML (*eXtensible Markup Language*, Lenguaje de Marcas eXtensible) que se obtienen como respuesta a las operaciones que realiza OpenNebula. Esto facilitará el despliegue de aplicaciones virtualizadas sobre infraestructuras de tipo Cloud desde programas que interactúen con el GMVs.

Por último, tras analizar la funcionalidad ofrecida por el GMVs, se ha optado por añadir al API soporte para realizar operaciones con clusters de MVs, ya que son de

interés en el ámbito de la computación científica las operaciones tanto con grupos de MVs como con una de ellas de forma aislada.

Cabe decir que el API ofrecerá mecanismos para detectar los problemas que puedan surgir durante las invocaciones remotas al servidor. Asimismo, las dificultades que puedan aparecer al procesar los resultados de las operaciones que ofrece OpenNebula también serán controladas para que los componentes de alto nivel dispongan de tolerancia a fallos.

1.3. ESTRUCTURA DE LA MEMORIA

Esta memoria se encuentra estructurada mediante capítulos que analizan todos los aspectos del trabajo realizado. En el capítulo 1 se sitúa la introducción al proyecto y los conceptos más relevantes para poder situarse en el contexto adecuado. Seguidamente, en el capítulo 2, se analizan una a una las tecnologías y herramientas utilizadas para la realización del proyecto.

A continuación se encuentra el capítulo 3, que describe el desarrollo del trabajo. En el primer subcapítulo se tratan los aspectos comunes de la interfaz de alto nivel desarrollada, desde la excepción definida para controlar cualquier problema que surja interiormente, fruto de las invocaciones a las operaciones que el API ofrece, hasta los métodos comunes para cualquier tipo de componente. Tras él, el resto de subcapítulos redactan las peculiaridades de cada uno de los objetos de alto nivel creados en el API, como son las máquinas virtuales, los hosts, las redes virtuales y los usuarios. En el subcapítulo 3.6 se encuentra la explicación de los métodos que permiten gestionar clusters de MVs.

Finalmente, en el capítulo 4 se exponen las conclusiones tras la realización del proyecto. En el apéndice A se encuentra un caso de uso para ver una aplicación real que utiliza el API desarrollado y en el B una guía para configurar el entorno de trabajo empleado. El apéndice C contiene ejemplos de resultados XML de OpenNebula.

1.4. CONCEPTOS INICIALES

En este apartado de la memoria se analizan, sin profundizar en exceso, los conceptos que se han considerado más relevantes para poder situarse de forma correcta en el ámbito del proyecto.

1.4.1. CLOUD COMPUTING

El término Cloud Computing (que en castellano significa "Computación en la nube", donde "nube" es una metáfora del concepto de Internet) hace referencia a un nuevo paradigma informático que surge para ofrecer servicios de cómputo a los usuarios, ya sea de forma gratuita (Clouds públicos) o pagando (Clouds privados). Al ser un servicio, como lo es la electricidad, los usuarios pueden acceder a él sin la necesidad de tener conocimientos sobre la gestión de los recursos que usan y sin tener que preocuparse por el proveedor del servicio para disfrutar de él.

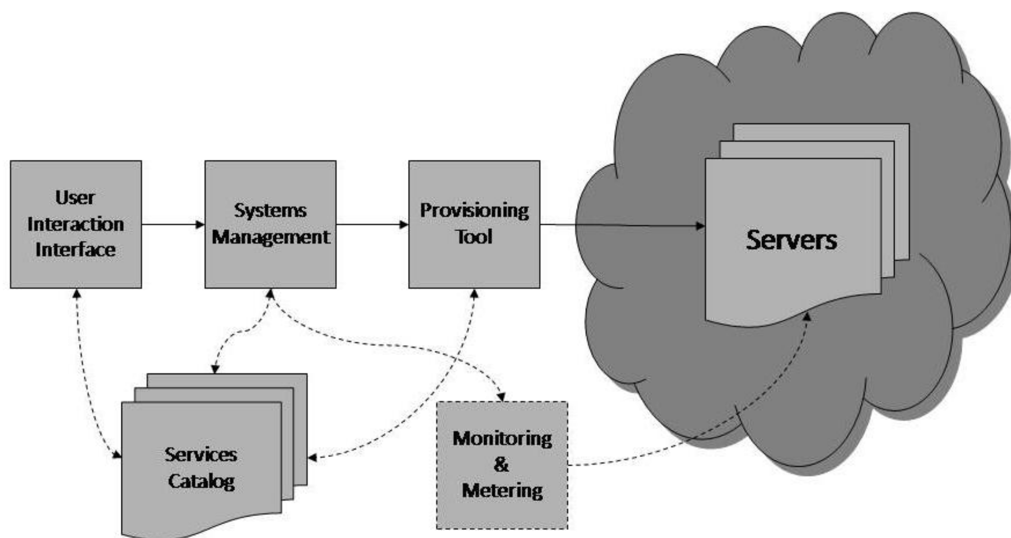

Figura 1.2 – Esquema de funcionamiento de una nube

Una característica relevante de este tipo de computación es su elasticidad. Los entornos Cloud son escalables, es decir, capaces de ajustarse a la demanda de los usuarios. Ante un pico de elevados requisitos de cómputo, un usuario puede solicitar mayor capacidad de cálculo, que provocará el despliegue automático de nuevas MVs sobre la infraestructura física existente. Esta es una gran ventaja, puesto que uno de los problemas que existían hasta la aparición del Cloud Computing era la creación de una infraestructura propietaria preparada para soportar demandas no previsibles de cómputo, ya que esto suponía una elevada inversión y además no se podía asegurar que fuese capaz de responder correctamente a estos incrementos repentinos en la demanda de los recursos de cómputo.

Además, las infraestructuras de tipo Cloud se apoyan en Internet para ofrecer su servicio, aprovechándose de las ventajas que ofrece esta red de redes. Los proveedores pueden ofrecer un gran número de servicios de forma rápida y eficiente, y los usuarios acceder a ellos de forma cómoda y segura.

La arquitectura que presenta este paradigma de computación está dividida en tres capas fundamentales, como podemos apreciar en la Figura 1.3.

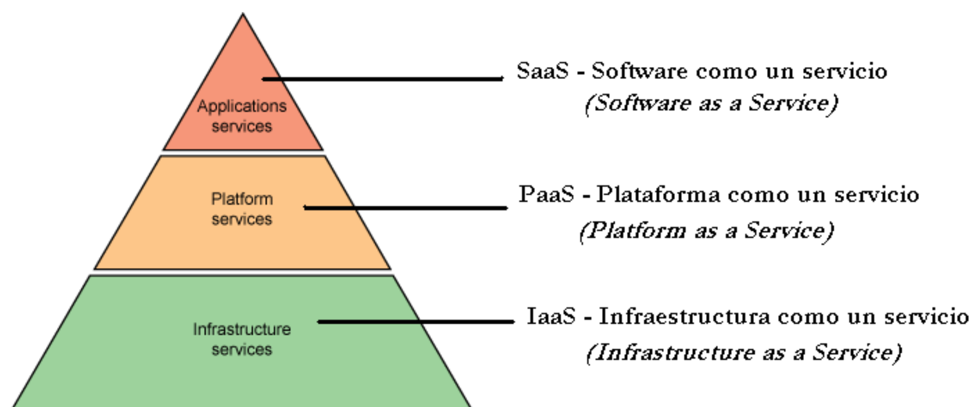

Figura 1.3 – Arquitectura del Cloud Computing

La capa superior, que se corresponde con el nombre de SaaS (*Software as a Sservice*, Software como un Servicio) es la que aloja a las aplicaciones que se ejecutan en la

nube y se ofrecen bajo demanda a modo de servicio a los usuarios. Estas aplicaciones son totalmente escalables y evitan a los usuarios tener que instalar y mantener el software, ya que pueden acceder a él a través de Internet. La empresa distribuidora aporta el servicio de mantenimiento y soporte del software solicitado por el cliente y sus beneficios se basan en el concepto de pago por el uso. Un ejemplo de ello pueden ser las variadas aplicaciones que ofrece la empresa Google, como GMail [7] o Google Sites [8], todas ellas reunidas en un conjunto de aplicaciones que recibe el nombre de Google Apps [9].

La capa intermedia, que recibe el nombre de PaaS (*Platform as a Service*, Plataforma como un Servicio) se encarga de proporcionar la infraestructura y el entorno de ejecución necesario a las aplicaciones de la capa superior como un servicio más. En otras palabras, esta capa permite el desarrollo de aplicaciones Cloud evitándole al usuario tener que preocuparse de la gestión y administración de la infraestructura. Suele estar virtualizada, para poder garantizar la escalabilidad. Un ejemplo práctico puede ser Google App Engine [10].

Por último se encuentra la capa IaaS (*Infrastructure as a Service*, Infraestructura como un Servicio) que se corresponde con la capa inferior de la arquitectura. En ella se ubican los recursos físicos, tales como servidores, bases de datos, o discos de almacenamiento que se ofrecen como un servicio al usuario. También suelen utilizar técnicas de virtualización para asegurar potencia de cálculo cuando es requerida y conseguir reducir costes gracias a la utilización más eficiente de los recursos. Una muestra de esta capa sería Amazon EC2 [11].

Para finalizar este apartado sobre el término "Cloud Computing" se ha considerado oportuno tratar, de forma resumida, un caso de uso de esta tecnología: los juegos sociales.

Actualmente, los juegos sociales se han convertido en una nueva forma de jugar online en Internet. Este tipo de juegos busca crear grandes redes de usuarios que puedan interactuar entre ellos a través de la aplicación. En este sentido es importante

señalar que el número de usuarios activo suele ser muy inconstante. En consecuencia, la demanda de las infraestructuras que dan soporte a este tipo de aplicaciones es muy dinámica. Y es aquí donde la computación en la nube cobra protagonismo ya que, como se ha podido analizar anteriormente, ésta es capaz de acoplarse perfectamente a las demandas inconstantes de recursos de cómputo. Contar con infraestructuras propias que den soporte a este tipo de juegos supone un gran coste y puede ser muy arriesgado, puesto que nunca se puede asegurar que se podrá cubrir un pico de demanda de cómputo con éxito.

Por todo ello, las infraestructuras de tipo Cloud se están posicionando actualmente como una buena solución para este tipo de aplicaciones y empresas importantes en el sector, como Zynga [12], ya han comenzado a hacer uso de ellas.

1.4.2. GESTORES DE MÁQUINAS VIRTUALES

Los gestores de máquinas virtuales (*Virtual Infrastructure Managers*) son herramientas software que permiten crear y gestionar nubes, ya sean públicas, privadas o híbridas. Desacoplan la MV de la localización física disponible. Crean una capa de virtualización distribuida que se sitúa entre la infraestructura física existente y el servicio que se presta a los usuarios, como se puede apreciar en la Figura 1.4. Esta capa permite extender los beneficios de las plataformas de virtualización a múltiples recursos y permite gestionar de forma independiente los recursos físicos y los servicios.

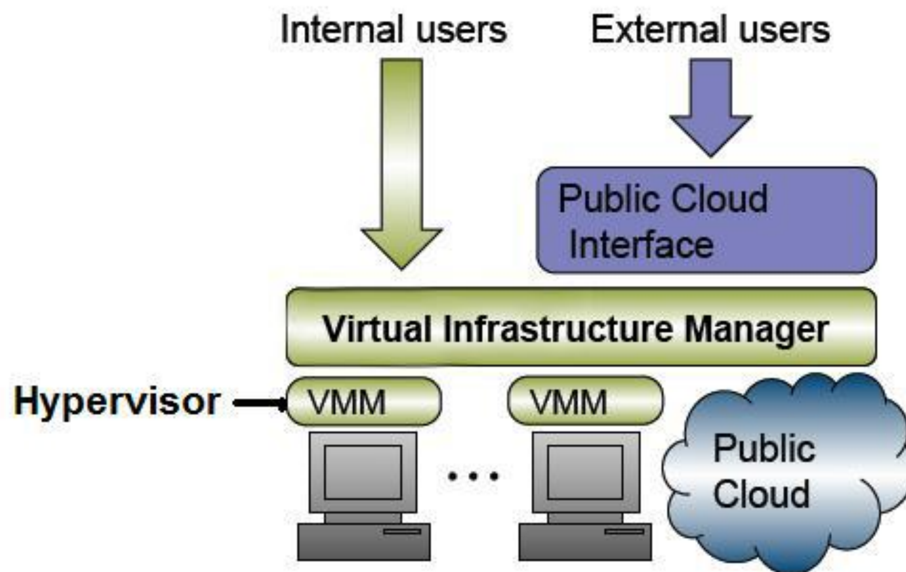

Figura 1.4 – Esquema de localización del GMVs

La utilización de un GMVs aporta ventajas tales como la gestión centralizada, el escalado y particionamiento de recursos dinámico y la provisión bajo demanda de MVs.

Algunos de los GMVs más conocidos actualmente son OpenNebula (gestor elegido para la realización del proyecto y que se analizará en el siguiente capítulo), Eucalyptus [13] y Abicloud, de la empresa Abiquo [14]. Los tres comparten una característica común: ser de código abierto.

1.4.3. HIPERVISORES

Los hipervisores o VMM (*Virtual Machine Monitor*, Monitor de Máquina Virtual) son herramientas software capaces de comunicarse con el sistema anfitrión, ya sea hardware o un SO (*Operative System*, sistema operativo), para interoperar entre un sistema de virtualización y el sistema físico. Conforma una pieza fundamental en la

tarea de ejecución de MVs. Su objetivo se basa en lograr la abstracción del hardware que requieren los SOs para permitir alojar una o más MVs en un mismo equipo.

Esta plataforma de virtualización se encarga de gestionar los cuatro recursos principales de un computador (CPU (*Central Processing Unit*, Unidad Central de Procesamiento), memoria, red y discos de almacenamiento) repartiendo dinámicamente dichos recursos entre todas las MVs definidas en el computador anfitrión.

Figura 1.5 – Arquitectura de trabajo de un Hipervisor

Existen dos tipos fundamentales de hipervisores:

- **Hipervisores de tipo 1 o *Bare-Metal*:** son capaces de ejecutarse directamente sobre el hardware real de la máquina sin necesidad de tener instalado ningún SO. Este tipo de Hipervisores son más eficientes que los de tipo 2, ya que consiguen dar un mayor rendimiento y escalabilidad, además de suponer menos sobrecarga para el computador. La desventaja es que no todo el hardware soporta este tipo de software. Ejemplos de este tipo de hipervisores son Xen [15], KVM [16] o VMware [17] ESXi.

- **Hipervisores de tipo 2 o *Hosted:*** necesitan un SO anfitrión para poder ejecutarse. Permiten la creación de MVs dentro del mismo SO. Al ejecutarse sobre un SO y no directamente sobre el hardware, el rendimiento de este tipo de hipervisores es menor que los de tipo 1. Suelen utilizarse en entornos de escritorio y no en grandes infraestructuras virtualizadas. Algunos de ellos son VirtualBox [18], VMware Player o Virtual PC.

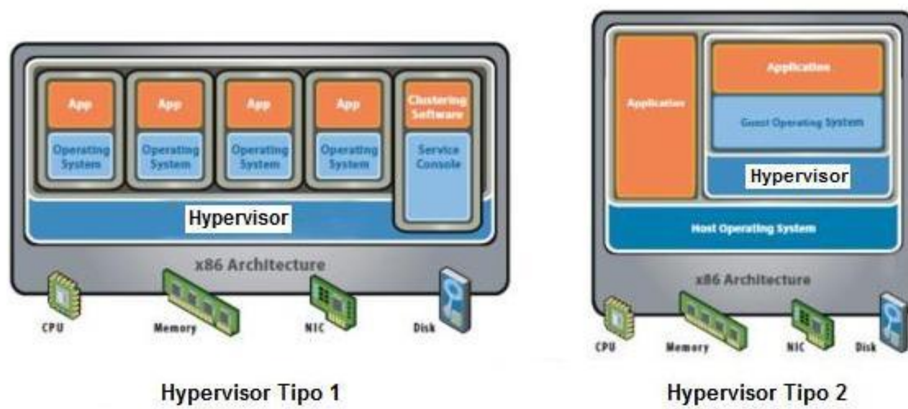

Figura 1.6 – Diferentes tipos de Hipervisores

2. TECNOLOGÍAS EMPLEADAS

En este apartado se analizan, una a una, las herramientas y tecnologías empleadas para la realización del proyecto.

2.1. JAVA

Java [19] es un lenguaje de programación orientado a objetos, desarrollado por la empresa Sun Microsystems (recientemente adquirida por otra empresa del sector, Oracle). Es multiplataforma y está desarrollado bajo la licencia GNU GPL (*General Public License*, Licencia Pública General).

Actualmente, este lenguaje de programación es uno de los más utilizados por su versatilidad, ya que Java se utiliza para crear todo tipo de aplicaciones, webs dinámicas, acceso a bases de datos, etc; por ser de código abierto y por ser independiente de la plataforma. Además, permite crear programas modulares y código reutilizable.

Para el desarrollo del código que integra el API se ha utilizado la versión 1.6.0_02 del JDK (*Java Development Kit*, Kit de Desarrollo de Java).

2.2. ECLIPSE GALILEO

Eclipse [20] es una plataforma de programación utilizada para crear entornos integrados de desarrollo (IDE, *Integrated Development Environment*). Este entorno de programación fue desarrollado originalmente por IBM (*International Business Machines*) como el sucesor de su familia de herramientas para VisualAge. Actualmente

es desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta el código abierto.

Eclipse Galileo [21] es una de las versiones más recientes de este IDE (la más reciente recibe el nombre de Eclipse Helios [22]), y es la que se escogió a la hora de realizar el proyecto puesto que, en el momento del comienzo del proyecto, era la última versión y daba soporte al *plug-in* para trabajar con el sistema de control de versiones Subversion [23] (SVN), necesario para poder comunicarse con el repositorio donde se alojaba el código fuente del proyecto.

2.3. SUBVERSION

Subversion, también conocido como svn, es un controlador de versiones empleado en la administración de archivos utilizados en el desarrollo de software. Es una herramienta multiplataforma que ayuda a que los desarrolladores lleven un seguimiento de los cambios en los ficheros de código fuente de su proyecto.

Esta herramienta permite, por ejemplo, obtener y comparar versiones anteriores de código o mantener una estructura de árbol, como se observa en la Figura 2.2, que le permite comportarse como un sistema de ficheros que evoluciona con cada conjunto de cambios. Subversion permite acceder al repositorio a través de la red, por lo que puede ser utilizado desde cualquier lugar.

Algunas de las operaciones más comunes que se pueden realizar sobre el repositorio mediante esta herramienta son *commit* (permite incorporar una nueva versión de archivos modificados localmente al repositorio) y *update* (utilizado para sincronizar la copia local con la del repositorio).

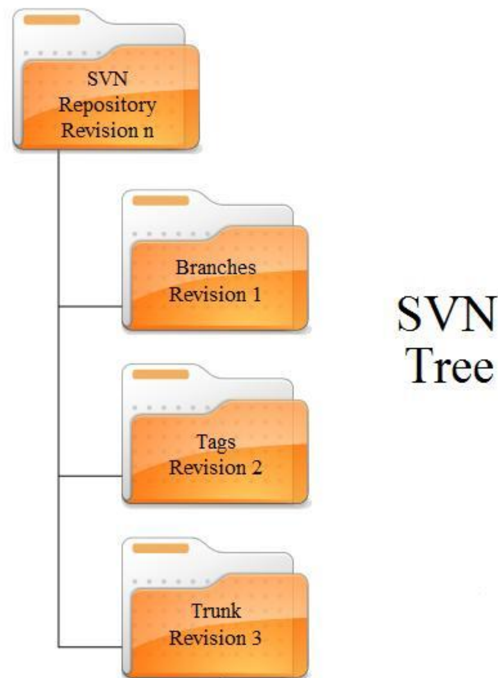

Figura 2.1 – Estructura de árbol de Subversion

Durante la realización del proyecto se ha utilizado esta herramienta para contar con una copia del código desarrollado para el API disponible en el repositorio. En el apéndice B se encuentra una guía de instalación de la herramienta Subversion en Eclipse Galileo.

2.4. XML-RPC

XML-RPC [24] es un protocolo de llamada a procedimiento remoto que utiliza XML para codificar los datos y HTTP (*Hypertext Transfer Protocol*, Protocolo de Transferencia de Hipertexto) para transmitirlos a través de Internet. Es un protocolo muy simple pero que permite trabajar con estructuras de datos complejas que pueden transmitirse, procesarse y ser devueltas como respuesta.

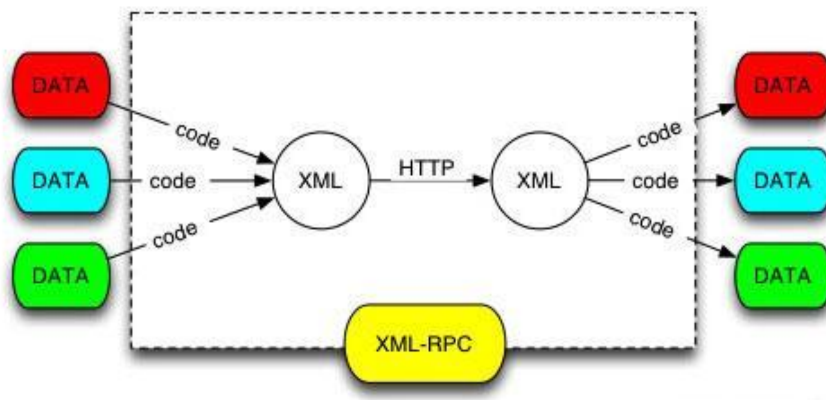

Figura 2.2 – Esquema de trabajo del protocolo XML-RPC

Este protocolo ha sido adaptado, mediante distintas implementaciones, a muchos de los sistemas operativos actuales y a distintos lenguajes de programación, como Java, C, C++, Python, Delphi o Perl, y su adaptación al resto de lenguajes y entornos está en auge.

El GMVs para el cual se va a desarrollar la interfaz de alto nivel, cuenta con su propio API que utiliza el protocolo XML-RPC para invocar sus operaciones. Y como Java también admite trabajar con este protocolo, este va a ser el camino a seguir para comunicarse con el servidor de OpenNebula desde el API en Java.

2.5. XPATH

El lenguaje XPath [25] (*XML Path Language*) permite buscar información dentro de un fichero XML y navegar entre etiquetas y atributos de manera sencilla mediante la construcción de expresiones que recorren y procesan el fichero. Es uno de los elementos principales del estándar XSLT (*Extensible Stylesheet Language Transformation*, transformación del lenguaje extensible de hojas de estilo) propuesto por el W3C (*World Wide Web Consortium*).

XPath incluye:

- Una sintaxis para definir las partes de un documento XML.
- Un conjunto de expresiones para seleccionar esas partes.
- Un conjunto de funciones estándar para el tratamiento de cadenas, valores numéricos, etc.

Cuando un documento XML es procesado por un XPath [26], se construye un árbol de nodos. Así se facilita la selección de partes del documento. En la Figura 2.3 se puede observar un ejemplo formado por el fichero de XML y su representación en forma arbórea.

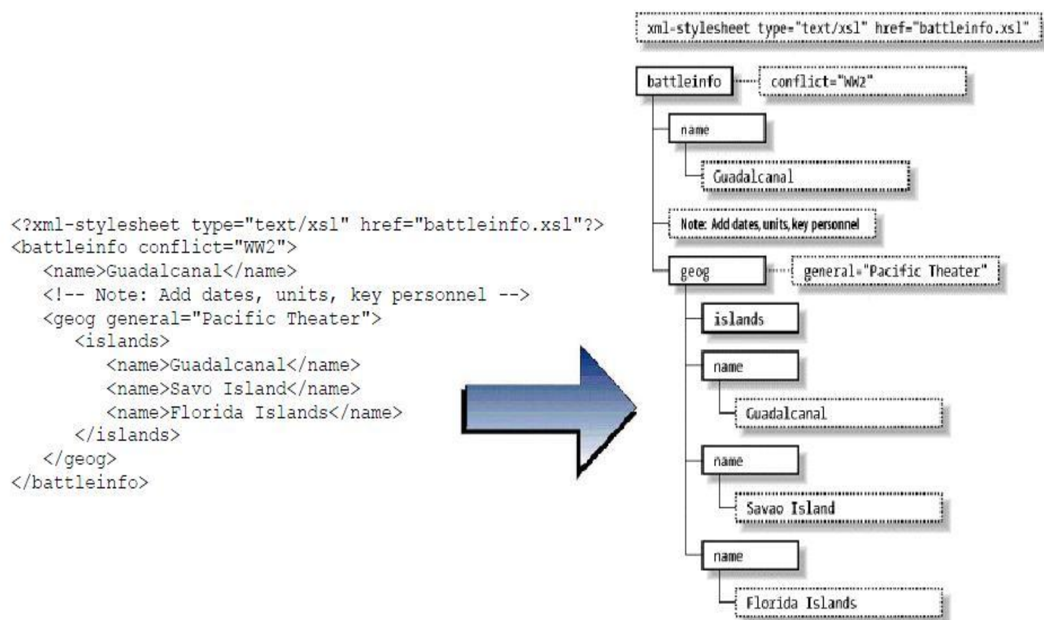

Figura 2.3 – Ejemplo de fichero XML y su representación en forma de árbol

Este lenguaje es una pieza fundamental en el desarrollo del proyecto, ya que permite procesar los resultados obtenidos tras las invocaciones a las operaciones que el servidor de OpenNebula ofrece y, a partir de ellos, construir los componentes de

alto nivel que facilitan la tarea al programador de aplicaciones para infraestructuras de tipo Cloud.

2.6. LOG4J

Log4j [27] *(logs for java)* es una herramienta de código abierto desarrollada en Java por la fundación Apache, que permite centralizar y administrar los mensajes y avisos de la aplicación. Evita el uso del *"System.out.println("");"* [28] y aporta ventajas sobre éste tales como la capacidad de habilitar y deshabilitar ciertos logs mediante la categorización de los mensajes de acuerdo al criterio del programador.

La configuración de salida se realiza mediante un fichero XML o en formato *Java Properties*, generalmente llamado *log4j.properties*.

Log4J [29] tiene por defecto 5 niveles de prioridad para los mensajes de logs:

- **DEBUG**: se utiliza para escribir mensajes de depuración.
- **INFO**: utilizado para mensajes informativos sobre el estado de la aplicación.
- **WARN**: se usa para mensajes de alerta sobre eventos que se desea mantener constancia, pero que no afectan el correcto funcionamiento del programa.
- **ERROR**: se utiliza para avisar de que ha ocurrido un error durante la ejecución de la aplicación. Generalmente estos eventos afectan al programa pero lo dejan seguir funcionando.
- **FATAL**: utilizado para avisar de mensajes críticos del sistema.

Adicionalmente a los anteriores 5 niveles, existen 2 más que se utilizan en el archivo de configuración:

•**ALL**: habilita todos los logs. Es el nivel más bajo posible.

•**OFF**: deshabilita todos los logs. Es el nivel más alto posible.

Un ejemplo de uso en la aplicación de esta herramienta se encuentra en la Figura 2.4.

```
import java.util.*;
import java.net.MalformedURLException;
...
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;           (1)

public class OpenNebulaClient
{
    private XmlRpcClient client;
    private XmlRpcClientConfigImpl config;
    private String SESSION;
    ...                                                      (2)
    Logger log = Logger.getLogger("OpenNebulaClient.class");
    ...
    log.debug("Trying to create a Virtual Machine");
    log.fatal("Error: " + msg + " : " + ex.getMessage());   (4)
    ...

    public static void main(String[] args) throws Exception{

        PropertyConfigurator.configure("log4j.properties"); (3)
     ...
    }
}
```

Figura 2.4 – Ejemplo de uso de Log4J

Para utilizar esta herramienta se deben importar las clases necesarias de Log4J en el código (en concreto la clase *Logger* y la clase *PropertyConfigurator*) (1). Seguidamente, se define una variable estática del tipo *org.apache.log4j.Logger* con el nombre de la clase que va a escribir en el registro (2). Finalmente se configura el objeto *Logger* con la clase *org.apache.log4j.PropertyConfigurator* indicando donde se encuentra el fichero de

configuración *log4j.properties* (3). Ya se pueden escribir mensajes mediante la variable creada (4) escogiendo el nivel de prioridad.

2.7. OPENNEBULA

OpenNebula es un GMVs de código abierto, desarrollado por el grupo de Arquitectura de Sistemas Distribuidos de la Universidad Complutense de Madrid. Permite crear Clouds privados, públicos e híbridos, y es capaz de gestionar almacenamiento, red y tecnologías de virtualización permitiendo la creación dinámica de MVs sobre infraestructuras Cloud. Sobre este GMVs se basa el API Java desarrollado.

Este GMVs está diseñado para soportar varios usuarios, pero sólo existe uno que puede llevar a cabo todas las operaciones que OpenNebula ofrece de forma satisfactoria, el propietario de la cuenta *oneadmin*. Éste será el administrador del sistema, el único capaz de llevar a cabo acciones como la gestión del resto de usuarios.

OpenNebula define una MV mediante una plantilla, que incluye datos como la capacidad de memoria y CPU, la red virtual a la que pertenece, el sistema operativo o la imagen en disco. Un ejemplo de plantilla válida se expone en el subcapítulo 3.2.3.

Cada MV en OpenNebula se identifica por un número único, el identificador (VID). Además, el usuario puede asignar un nombre a ésta en la plantilla; el nombre predeterminado para cada MV es su identificador.

OpenNebula define que el ciclo de vida de toda MV incluye las siguientes etapas:

- ***Pending***: por defecto una MV se inicia en este estado, en espera de un recurso que ejecutar.

- ***Hold:*** el usuario ha declarado la MV y no será puesta en marcha hasta que se libere.

- ***Prolog:*** el sistema transfiere los archivos de la MV (imágenes de disco y el archivo de recuperación).

- ***Running:*** en este estado, la MV se está ejecutando.

- ***Migrate:*** la MV está migrando de un recurso a otro. Puede realizarse de dos formas: *livemigration* (la MV se transfiere a otro recurso sin que se aprecie tiempo de inactividad) o la migración en frío, donde se guarda la actual MV y los archivos transfieren al nuevo recurso.

- ***Epilog:*** en esta fase se eliminan los archivos de la máquina utilizada para alojar la MV y se realiza una copia de las imágenes de disco que hay que guardar.

- ***Stopped:*** la MV se detiene. Su estado se ha guardado y ha sido transferido de vuelta junto con las imágenes de disco.

- ***Suspended:*** estado similar al anterior, pero los archivos permanecen en el recurso remoto para reiniciar más tarde la MV.

- ***Failed:*** la máquina virtual falló como consecuencia de algún error.

- ***Unknown:*** la MV no pudo ser localizada, puesto que se encuentra en un estado desconocido.

- ***Done:*** la máquina virtual finaliza. Las MVs en este estado no serán mostradas en la lista de MVs creadas, pero se mantienen en la base de datos a efectos contables.

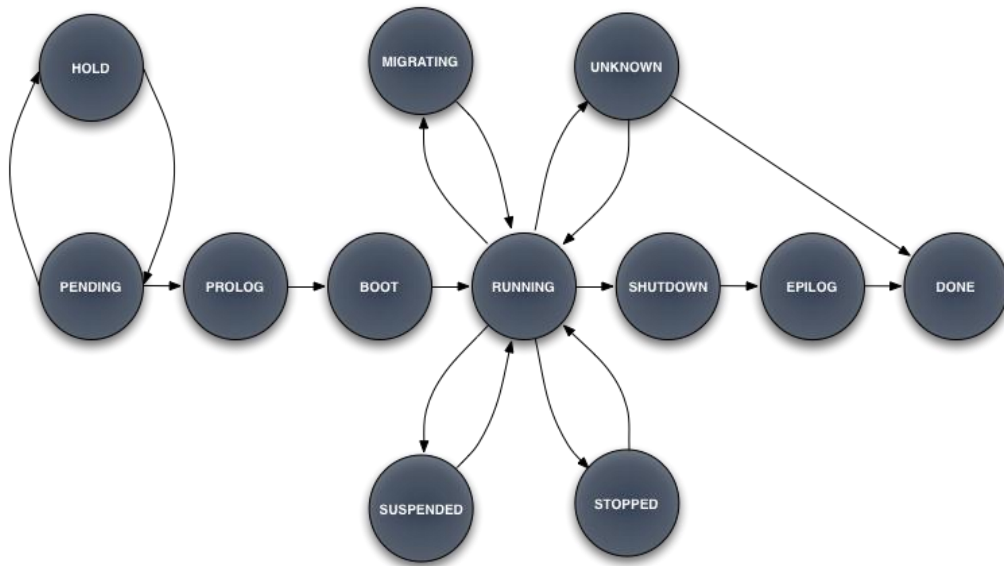

Figura 2.5 – Ciclo de vida de una MV

3. DESARROLLO DEL API

Este capítulo conforma el grueso de la memoria, ya que analiza detalladamente el API desarrollado para el proyecto. Está compuesto de 6 subcapítulos en los que se estudian aspectos como las operaciones destinadas a la gestión de MVs o los métodos destinados a gestionar las cuentas de usuario.

3.1. ASPECTOS GENERALES DEL API

En este subcapítulo de la memoria se comienza a analizar el API desarrollado en el proyecto. Concretamente se estudian los aspectos comunes o generales de la interfaz, como son la excepción propia creada para el control de los problemas que puedan surgir durante la ejecución de las operaciones del API y la clase que integra la gestión, mediante el uso de XPath, necesaria para procesar correctamente las respuestas de OpenNebula.

3.1.1. ESQUEMA DE CLASES

Antes de comenzar a analizar aspectos concretos del API, es necesario conocer la estructura de las clases que lo conforman.

El proyecto está compuesto por un paquete, denominado *opennebula*, que incluye 4 clases:

- ***OpenNebulaClient*:** es la clase principal del API, que incluye todas las operaciones que éste ofrece.

- ***ONEVMTemplate***: esta clase permite crear objetos de tipo *ONEVMTemplate* para construir las plantillas necesarias para crear MVs de forma más cómoda.

- ***OneVMCluster***: esta clase es la encargada de dar soporte a las operaciones con clusters de MVs.

- ***CloudException***: clase encargada de definir la excepción propia utilizada en el API.

opennebula también incluye otros dos paquetes:

- ***result***: incluye las clases encargadas de tratar el XML resultante de las invocaciones a OpenNebula y de construir los objetos de alto nivel que facilitan la interacción con el gestor.

- ***test***: aquí se encuentran las clases utilizadas para probar el correcto funcionamiento del API.

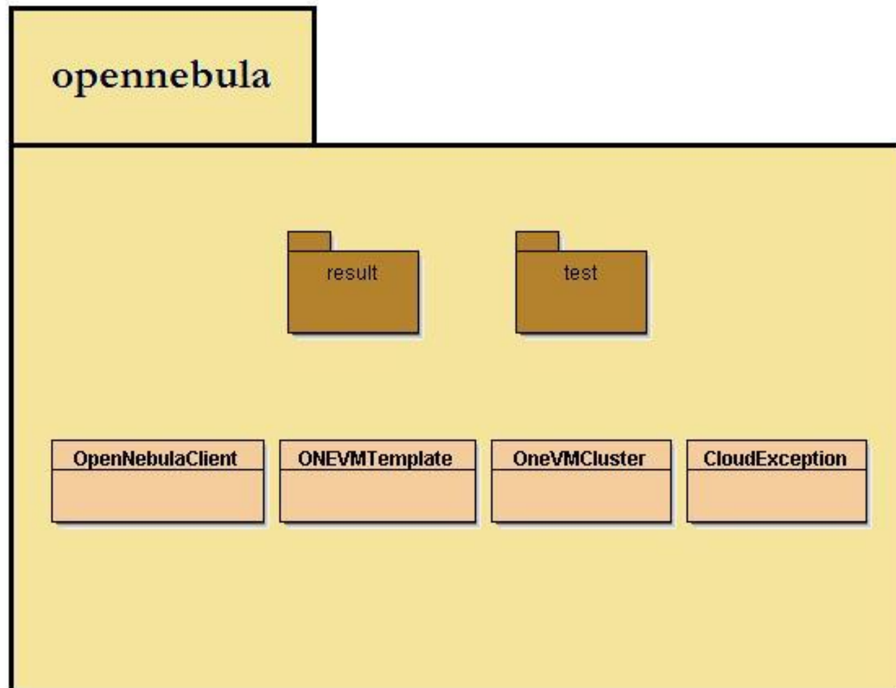

Figura 3.1 – Estructura del paquete *opennebula*

El paquete *result* está formado por 5 clases encargadas de procesar los XML resultantes de las invocaciones a las operaciones que ofrece OpenNebula:

- ***CloudOneResult***: clase que contiene los aspectos comunes en el procesamiento de los resultados en formato XML.

- ***VMInfoResult***: define los objetos que se obtienen como resultado de invocar las operaciones de información sobre MVs definidas en la clase *OpenNebulaClient*.

- ***HostInfoResult***: define los objetos que se obtienen como resultado de invocar las operaciones de información sobre hosts (máquinas) definidas en la clase *OpenNebulaClient*.

- ***VNInfoResult*:** define los objetos que se obtienen como resultado de invocar las operaciones de información sobre redes virtuales definidas en la clase *OpenNebulaClient*.

- ***UserInfoResult*:** define los objetos que se obtienen como resultado de invocar las operaciones de información sobre usuarios definidas en la clase *OpenNebulaClient*.

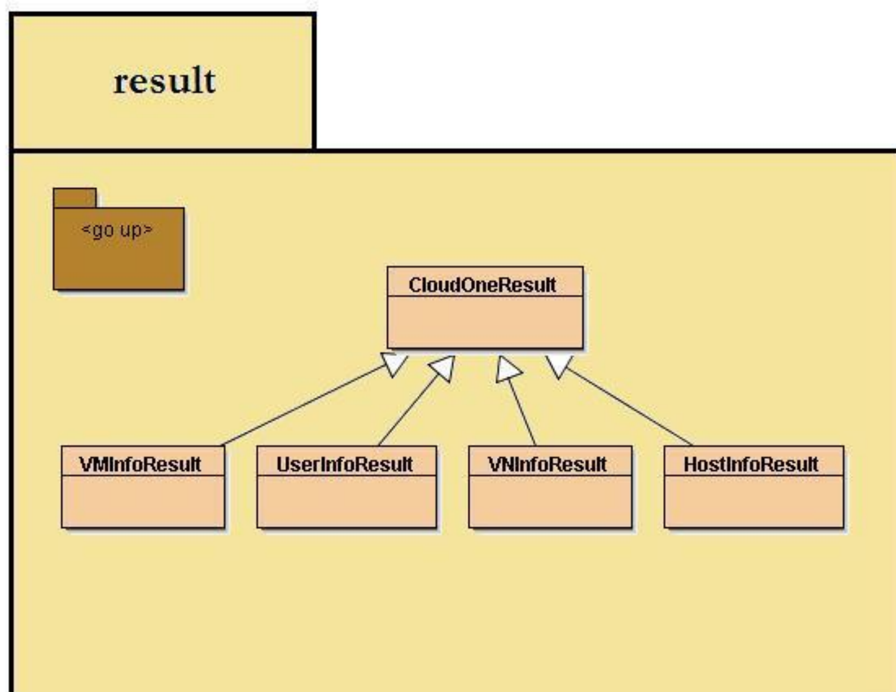

Figura 3.2 – Estructura del paquete *result*

Por último, el paquete *test* contiene las clases utilizadas para la realización de las pruebas del API. También incluye el caso de uso.

- ***TestOpenNebulaClient*:** testea el comportamiento de la clase *OpenNebulaClient*.

- ***CasoDeUsoOpenNebulaClient*:** contiene el caso de uso desarrollado para ver un ejemplo de funcionamiento del API. En el Apéndice A se encuentra expuesto.

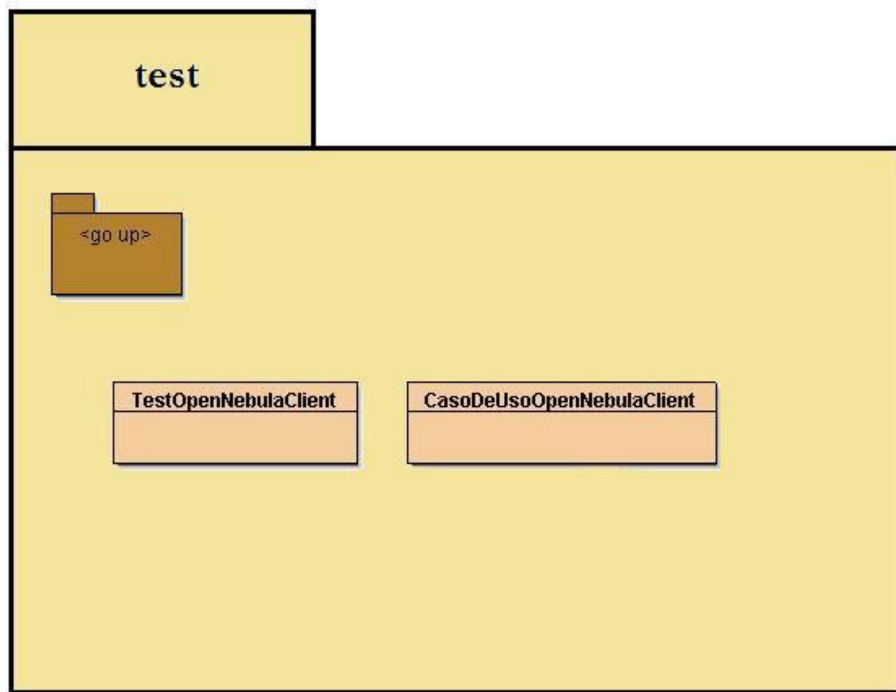

Figura 3.3 – Estructura del paquete *test*

3.1.2. LA EXCEPCIÓN CLOUDEXCEPTION

Para poder gestionar correctamente los problemas imprevistos que puedan surgir durante la ejecución de las distintas operaciones de que dispone la interfaz de alto nivel que se está construyendo, se ha optado por definir una excepción propia que recibe el nombre de *CloudException*.

Mediante el uso de excepciones en Java se consigue crear aplicaciones capaces de responder y recuperarse, en la medida de lo posible, ante errores inesperados que pueden aparecer en cualquier momento durante la ejecución de las mismas.

Las excepciones son objetos de tipo *Exception*, clase que hereda de *Throwable*, que a su vez hereda de *Object*, convirtiéndose en objetos que pueden ser lanzados por los métodos. El esquema de clases se puede observar en la Figura 3.4.

La excepción *CloudException* es definida por el usuario, es decir, no pertenece al estándar de Java pero hereda de la clase *Exception*, permitiendo contar con una excepción propia creada exclusivamente para el API.

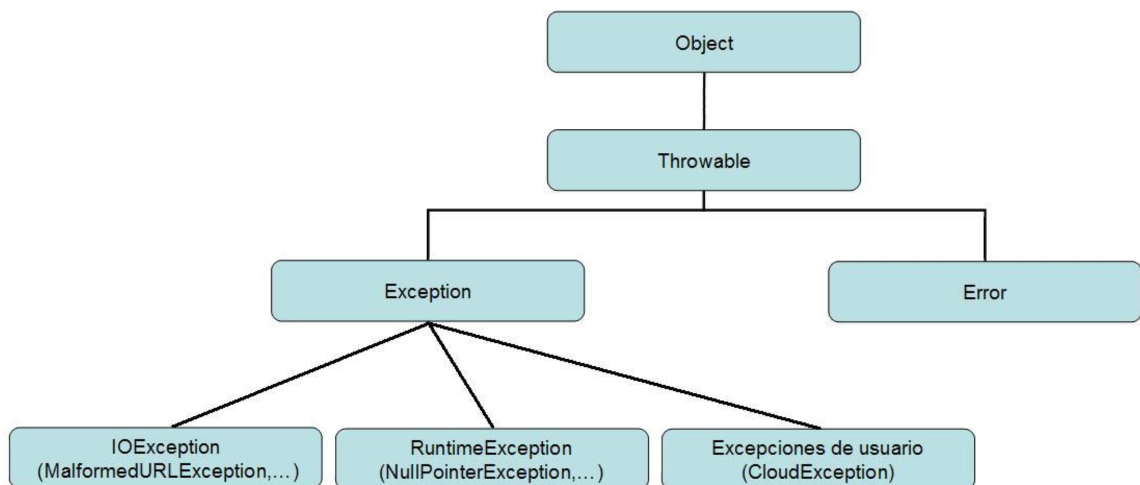

Figura 3.4 – Esquema de las excepciones en Java

Esta excepción la pueden lanzar todos los métodos de la clase *OpenNebulaClient* para avisar de que, o bien la invocación remota al servidor del gestor ha fallado por motivos asociados con la programación de entornos distribuidos (caída de servicios, fallos en la red, etc.), o bien el fallo viene provocado por los parámetros con los que se ha hecho la invocación (permisos insuficientes del usuario que la realiza, identificador de una máquina virtual (MV, *Virtual Machine*) inexistente, etc.).

Cualquier aplicación que realice llamadas al API puede capturar esta excepción y analizar el mensaje que contiene para conocer el motivo por el cual fue lanzada. También puede acceder a la excepción que provocó el error, ya que *CloudException* cuenta con dos constructores, uno de ellos recibe un mensaje y el otro, además del mensaje, la excepción que se recibió al producirse el error. Mediante el encadenamiento de excepciones es posible conocer con detalle el origen de una excepción, así como su camino dentro de la secuencia de invocaciones de métodos.

3.1.3. LA CLASE CLOUDONERESULT

CloudOneResult es una clase que pertenece al paquete *result*, y que contiene los aspectos comunes para el tratamiento de los resultados obtenidos, en formato XML, tras invocar las operaciones que ofrece el GMVs OpenNebula.

En esta clase se hace uso del lenguaje XPath para poder alcanzar su objetivo. Para ello se han definido dos atributos, *factory* y *xpath*, necesarios para crear un objeto de tipo *XPath* que permitirá evaluar expresiones XPath sobre el XML. Cabe señalar que para trabajar con objetos de este tipo es imprescindible importar la librería *javax.xml.xpath*.

Otro atributo importante de esta clase es *xmlResult*, que contiene en forma de String el XML sobre el que hay que realizar la evaluación de las expresiones.

La clase está formada por 6 métodos:

- **El constructor, *CloudOneResult*:** recibe el String devuelto por OpenNebula y lo asigna a la variable destinada a ello. Además inicializa los atributos para trabajar con XPath.

- ***evaluate*:** este método recibe la expresión a evaluar, en formato String, y devuelve el valor obtenido tras la evaluación, también en formato String.

Las expresiones deben tener la forma /"nombre_etiqueta"/"nombre etiqueta". En la Figura 3.5 se puede observar un ejemplo.

Los componentes de alto nivel que devuelve el API se crean a partir de las distintas invocaciones a este método, accediendo a todos los atributos disponibles en el XML.

```
***Operation getUserInfo***

<USER>
        <ID>2</ID>
        <NAME>amcaar</NAME>
        <PASSWORD>911c9c3fd86910130413d771087afd431af76b7</PASSWORD>
        <ENABLED>1</ENABLED>
</USER>
```

La expresión /USER/ID obtendría como resultado, tras ser evaluado con XPath, "2"

La expresión /USER/ENABLED obtendría como resultado "1"

Figura 3.5 – Ejemplo de expresiones válidas para XPath

- ***pool***: este método es invocado por las operaciones situadas en la clase *OpenNebulaClient* que obtienen la información de un grupo de objetos (ya sean MVs, hosts, usuarios o redes virtuales (VNs)). Como resultado de su invocación se obtiene un vector de números enteros que se corresponden con los identificadores de los objetos de tipo *elementName* (parámetro de entrada de este método). Este vector será imprescindible para poder construir posteriormente un *ArrayList* con los objetos de alto nivel correspondientes con los identificadores obtenidos.

- ***numberOfAttributes***: este método devuelve un número entero con el total de atributos existentes con la misma etiqueta en el XML. Esta

etiqueta la recibe como un parámetro en formato String. Es útil para conocer, por ejemplo, el número de máquinas virtuales que hay desplegadas, o el número de usuarios registrados.

- ***getTagValue***: el método, al igual que los dos anteriores, se emplea en las operaciones de *pool* y es capaz de devolver un String con el valor de una etiqueta concreta, que recibe como parámetro de ese mismo tipo.

- ***toSring***: el método devuelve un String con el XML original recibido del GMVs. De esta forma siempre se puede acceder al resultado original.

3.1.4. El MÉTODO EXTRACTRESULT

En la clase *OpenNebulaClient* podemos encontrar todas las operaciones que ofrece el API y además un método, *extractResult*, utilizado por todas estas operaciones para poder procesar correctamente los resultados obtenidos tras las invocaciones al gestor con el que se está trabajando.

Analizando el API basado en XML-RPC que ofrece OpenNebula se puede apreciar que todos los resultados obtenidos tras la ejecución de los métodos que ofrece siguen un patrón común. Este resultado siempre está formado por:

- Un parámetro de tipo booleano que toma el valor verdadero (*true*) cuando la operación se ha finalizado con éxito, y toma el valor falso (*false*) cuando el método no finalice como se espera.

- Un parámetro de tipo String, que puede estar vacío o contener un mensaje. Este mensaje puede ser de error o puede contener la información solicitada en operaciones de tipo *info*.

Adicionalmente, las operaciones que sirven para crear una nueva MV, VN (red virtual, *Virtual Network*), host o usuario, pueden devolver un parámetro de tipo entero que contiene el identificador del objeto creado si la operación concluye con éxito. Si no, este parámetro será un mensaje de error contenido en un String. En todo caso, todas las operaciones devuelven siempre 2 parámetros.

Por este patrón común, se ha considerado oportuno la construcción de un método capaz de analizar los distintos parámetros de salida de las operaciones del API XML-RPC [30] para detectar si la operación ha transcurrido sin problemas o se ha producido algún error. Además, la construcción de este método permite dotar de eficiencia y reusabilidad al código ya que no hay que repetir para cada operación que ofrece el API las mismas líneas de código para el estudio de los resultados, sino que esta funcionalidad está extraída de los métodos y se realiza en otra parte.

El método *extractResult* puede lanzar la excepción *CloudException* cuando detecta que la ejecución de la operación no se realizó correctamente. Para detectarlo comprueba si la variable booleana tiene un valor falso. En este caso se lanzará una *CloudException*. Si la variable booleana tiene un valor verdadero, significa que la operación se realizó con éxito, pero es necesario comprobar si se trataba de una llamada a un método de creación o de cualquier otro, puesto que el tipo del segundo parámetro vendrá determinado por este factor. Para ello, se realiza una conversión de tipo a entero del segundo parámetro. Si la operación se realiza con éxito significa que el método que se invocó fue el de creación, y *extractResult* devuelve el identificador. Cuando salta una excepción al intentar hacer la conversión, se sobreentiende que el segundo parámetro se corresponde con un String que, o bien no tendrá contenido, o bien contendrá la información solicitada. En este caso el valor entero devuelto por el método que estamos analizando en este apartado, será ignorado por el que lo invocó.

3.2. OPERACIONES DESTINADAS A LA GESTIÓN DE MÁQUINAS VIRTUALES

En este subcapítulo, se exponen las operaciones disponibles en el API que están destinadas a la gestión de MVs. Además se define el componente de alto nivel creado para este tipo de operaciones y una clase creada específicamente facilitar para la creación de MVs, *ONEVMTemplate*.

3.2.1. OPERACIONES DISPONIBLES EN LA CLASE OPENNEBULACLIENT

Como ya se ha mencionado en capítulos anteriores, la clase *OpenNebulaClient* incluye todas las operaciones que ofrece el API. En este apartado se expone cada operación destinada a la gestión de MVs por separado, analizando la funcionalidad de cada una.

El API dispone de un total de 16 operaciones destinadas a la gestión de MVs. Algunas se corresponden directamente con sus semejantes en el API XML-RPC (versión 1.4) que ofrece OpenNebula, y otras se han añadido por diversos motivos. En la siguiente tabla se puede ver, de forma resumida, el perfil de cada operación:

Nombre del método	Parámetros de entrada	Parámetros de salida	Correspondencia directa con las operaciones del API XML-RPC
createVM	Template de la MV	Id de la máquina creada	Sí (*one.vm.allocate*)
submitVM	Template de la MV	Objeto *VMInfoResult*	No
deployVM	Id de host y MV	Vacío *(void)*	Sí (*one.vm.deploy*)
getVMInfo	Id de MV	Objeto *VMInfoResult*	Sí (*one.vm.info*)
shutdownVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
holdVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
releaseVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
stopVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
cancelVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
suspendVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
resumeVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
restartVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)
finalizeVM	Id de MV	Vacío *(void)*	No* (*one.vm.action*)

migrateVM	Id de host y MV, booleano *livemigration*	Vacío *(void)*	Sí (*one.vm.migrate*)
vmPoolInfo	Vacío *(void)*	ArrayList de objetos *VMInfoResult*	Sí (*one.vmpool.info*)
vmPoolInfoAll	Vacío *(void)*	ArrayList de objetos *VMInfoResult*	Sí (*one.vmpool.info*)

*****Estas operaciones se corresponden directamente con el método one.vm.action, pero no existe en el API XML-RPC una operación para cada una de ellas.

Tabla 3.1 – Métodos destinados a la gestión de MVs

A continuación se describe la funcionalidad que ofrece cada método:

- ***createVM:*** el método invoca a su correspondiente en el API XML-RPC, *one.vm.allocate*, creando una MV cuyas características se especifican en el parámetro de entrada *template*. Si la operación finaliza satisfactoriamente, el método retorna el número identificador de la MV creada. Si, por el contrario, la operación falla, se transmite la excepción *XmlRpcException* y devuelve un número negativo.

- ***submitVM:*** este método complementa la funcionalidad del anterior ya que, tras realizar una llamada a *createVM*, se busca la información de la MV recién creada y se crea un objeto de alto nivel de tipo *VMInfoResult* que es el que devuelve este método. La obtención de la información se realiza mediante la invocación al método *getVMInfo*. Si durante la ejecución del método ocurre algún problema, se lanza la excepción *CloudException*.

- ***deployVM:*** este método permite desplegar una MV en un host determinado. Necesita que el invocante le indique el identificador tanto del host destino como de la MV a desplegar. Internamente realiza una llamada al método *one.vm.deploy* del API XML-RPC de que dispone OpenNebula. Si se produce algún error durante su ejecución, el método

lanza la excepción *CloudException*, indicando el motivo por el cual se ha abortado la ejecución.

- ***getVMInfo:*** esta operación obtiene la información de la MV correspondiente con el identificador que recibe. Es capaz de devolver un objeto de alto nivel, de tipo *VMInfoResult*, al que se le podrán consultar todos los atributos incluidos en el XML que devuelve la operación *one.vm.info* (incluida en el API XML-RPC del gestor). Lanza la excepción *CloudException* en el caso de que se produzca algún error.

- ***shutdownVM:*** este método apaga la MV cuyo identificador corresponde con el que recibe como parámetro de entrada. Realiza una llamada interna al método *one.vm.action* del API XML-RPC indicándole como parámetro de entrada el nombre de la acción a realizar, en este caso "shutdown". Como todos los demás métodos de que dispone el API desarrollado, si surge algún error de comunicación con el servidor de OpenNebula o si el resultado obtenido de este último no es el esperado, el método lanza la excepción *CloudException*.

- ***holdVM:*** el método invoca a *one.vm.action* indicándole que la operación que se desea ejecutar es "hold". Con ello, este método permite mantener el estado de la MV correspondiente con el identificador que se recibe como parámetro de entrada. Esta operación es útil para inicializar una MV y esperar a ponerla en marcha posteriormente mediante la invocación al método *releaseVM*. En el caso de que ocurra un error se lanza una *CloudException*.

- ***releaseVM:*** para poner en marcha (liberar) una MV se puede invocar a este método que consigue, mediante la invocación a *one.vm.action* con el parámetro de entrada "release", liberar la MV cuyo identificador es igual al recibido como parámetro de entrada. Si, por algún motivo, surge un problema durante su ejecución, la excepción *CloudExceptión* será lanzada.

- ***stopVM:*** este método permite parar la MV. Realiza una llamada interna a la operación *one.vm.action* con el parámetro "stop" que indica a esta operación la acción que tiene que llevar a cabo. Como el resto de operaciones que se basan en la invocación a one.vm.action, precisa del identificador de la máquina que se desea parar. Nuevamente, si ocurre algún problema será lanzada la excepción *CloudException*.

- ***cancelVM:*** mediante la invocación a este método se puede cancelar la MV. Para ello este método se encarga de realizar una llamada interna a la operación one.vm.action tomando el parámetro que indica la acción a realizar el valor "cancel". Si surge algún problema de conexión durante la invocación o el resultado obtenido es erróneo, se aborta la ejecución del método lanzando una *CloudException*.

- ***suspendVM:*** suspende a la MV cuyo identificador se corresponde con el que recibe este método como parámetro de entrada. Internamente, realiza una llamada al método one.vm.action del API XML-RPC de OpenNebula indicándole como parámetro de entrada que la acción que se desea llevar a cabo es "suspend". Si la invocación a este método falla o el resultado obtenido no es el esperado, se lanza la excepción *CloudException*.

- ***resumeVM:*** este método reanuda la ejecución de la MV cuyo identificador es idéntico al recibido como parámetro de entrada. Para ellos invoca a *one.vm.action* indicando que la acción a realizar sea "resume". Si el resultado obtenido no es el esperado porque el identificador no sea válido, por ejemplo, o la llamada no se pueda realizar con éxito, se aborta la ejecución del método lanzando una *CloudException* que contendrá en su mensaje el motivo.

- ***restartVM:*** esta operación permite reiniciar la MV indicándole el identificador de la misma. Realiza una invocación remota al método

one.vm.action indicando que la acción que debe realizar es "restart". Si ocurre algún problema durante su ejecución, se lanza la excepción *CloudException*.

- ***finalizeVM:*** mediante la invocación a este método se puede finalizar la MV cuyo identificador se corresponde con el recibido por el método como parámetro de entrada. Invoca internamente a la operación *one.vm.action* para que ésta realice la acción "finalize". De nuevo, si ocurre algún problema será lanzada la excepción *CloudException*.

- ***migrateVM:*** esta operación es capaz de migrar una MV al host que se le indica como destino (mediante su identificador). Para ello precisa recibir los identificadores de los dos objetos involucrados en la operación, la MV y el host; y un booleano que le indica si se desea realizar *livemigration*. Si ocurre algún error durante su ejecución esta es abortada lanzando la excepción *CloudException*.

- ***vmPoolInfo:*** este método recupera la totalidad o parte de la información de las MVs existentes. La cantidad de información devuelta depende del usuario que la invoque. Sólo el usuario cuyas credenciales coincidan con las de *oneadmin* podrá acceder a la información de todas las MVs creadas, mientras que un usuario cualquiera tendrá acceso a las máquinas que él creó. Internamente realiza una llamada al método *one.vmpool.info*, existente en el API XML-RPC que ofrece OpenNebula. Devuelve un ArrayList de objetos *VMInfoResult*. Si se produce algún contratiempo durante su ejecución, el método responderá lanzando la excepción *CloudException*.

- ***vmPoolInfoAll:*** si se desea obtener la información de todas las máquinas virtuales desplegadas puede invocarse a este método. Pero para obtener la información de la totalidad de las MVs es necesario que las credenciales de usuario correspondan con *oneadmin*, es decir, con el

administrador. De lo contrario la ejecución concluirá con el lanzamiento de la excepción *CloudException*. Al igual que el método anterior, se basa en una llamada a *one.vmpool.info*, pero con parámetros de entrada diferentes y retorna un ArrayList de objetos *VMInfoResult*.

Cabe señalar que todas las operaciones explicadas anteriormente utilizan la herramienta Log4j para poder avisar del estado del API en cada momento. Además, tras recibir la respuesta de OpenNebula, invocan al método *extractResult* (analizado en el subcapítulo anterior) para analizar el resultado obtenido.

También es importante destacar que la operación *vmPoolInfo* devuelve un ArrayList de objetos *VMInfoResult* donde todos los atributos de los objetos toman valor, pero si se analiza detenidamente el XML resultante tras la invocación a *one.vmpool.info* (en el Apéndice C se puede consultar un ejemplo) se puede observar que este XML no incluye el valor de muchos de los atributos disponibles. Es por ello que la estrategia seguida en este método es conseguir todos los identificadores de las MVs mediante el método *pool* de la clase *CloudOneResult*, y para cada una invocar al método *getVMInfo* que, como se ha mencionado anteriormente, realiza una llamada a *one.vm.info* cuyo XML resultante sí contiene la información de todos los atributos.

3.2.2. EL OBJETO DE ALTO NIVEL VMINFORESULT

Para poder acceder a todos los atributos incluidos en el XML resultante de las invocaciones a los métodos *one.vm.info* y *one.vmpool.info* de forma sencilla, se ha considerado oportuno la creación de un objeto de alto nivel que facilite el acceso a estos atributos desde cualquier programa que necesite trabajar con el API.

La clase *VMInfoResult* es la encargada de definir este objeto. También incluye las operaciones de acceso a todos los atributos. Hereda de la clase *CloudOneResult*, permitiéndole así hacer uso de todos sus métodos.

En la Tabla 3.2 se pueden observar los distintos métodos de que dispone la clase, además de su correspondencia con el resultado obtenido en formato XML.

Nombre del método	Tipo de parámetro de salida	Correspondencia con el XML
constructor	*VMInfoResult*	No procede
getID	int	/VM/ID
getUID	int	/VM/UID
getName	String	/VM/NAME
getLastPoll	long	/VM/LAST_POLL
getStateNumber	int	/VM/STATE
getState	String	/VM/STATE*
getLCMStateNumber	int	/VM/LCM_STATE
getLCMState	String	/VM/LCM_STATE*
getStime	long	/VM/STIME
getEtime	long	/VM/ETIME
getDeployID	String	/VM/DEPLOY_ID
getMemory	long	/VM/MEMORY
getCPU	long	/VM/CPU
getNetTX	int	/VM/NET_TX
getNetRX	int	/VM/NET_RX
getFiles	String	/VM/TEMPLATE/CONTEXT/FILES
getContextTarget	String	/VM/TEMPLATE/CONTEXT/TARGET
getTemplateCPU	long	/VM/TEMPLATE/CPU
IsDiskReadonly	boolean	/VM/TEMPLATE/DISK/READONLY
getDiskSource	String	/VM/TEMPLATE/DISK/SOURCE
getDiskTarget	String	/VM/TEMPLATE/DISK/TARGET
getGraphicsListenIP	String	/VM/TEMPLATE/GRAPHICS/LISTEN
getGraphicsType	String	/VM/TEMPLATE/GRAPHICS/TYPE
getTemplateMemory	long	/VM/TEMPLATE/MEMORY
getBridge	String	/VM/TEMPLATE/NIC/BRIDGE
getIP	String	/VM/TEMPLATE/NIC/IP
getMAC	String	/VM/TEMPLATE/NIC/MAC
getNetwork	String	/VM/TEMPLATE/NIC/NETWORK
getVirtualNetworkID	int	/VM/TEMPLATE/NIC/VNID
getOSBoot	String	/VM/TEMPLATE/OS/BOOT
getOSRoot	String	/VM/TEMPLATE/OS/ROOT
getRank	String	/VM/TEMPLATE/RANK
getRequirements	String	/VM/TEMPLATE/REQUIREMENTS
getSEQ	long	/VM/HISTORY/SEQ
getHostName	String	/VM/HISTORY/HOSTNAME
getHID	int	/VM/HISTORY/HID

getPStime	long	/VM/HISTORY/PSTIME
getPEtime	long	/VM/HISTORY/PETIME
getRStime	long	/VM/HISTORY/RSTIME
getREtime	long	/VM/HISTORY/RETIME
getEStime	long	/VM/HISTORY/ESTIME
getEEtime	long	/VM/HISTORY/EETIME
getReason	int	/VM/HISTORY/REASON

*****La correspondencia no es directa.

Tabla 3.2 – Métodos disponibles en la clase *VMInfoResult*

Todos los anteriores métodos menos, obviamente, el constructor, basan su funcionamiento en la invocación al método *evaluate* de la clase padre (*CloudOneResult*) con un String como parámetro de entrada que contiene la expresión XPath a evaluar (expresión que se corresponde con la columna "Correspondencia con el XML" de la Tabla 3.2). Como este método siempre devuelve un String y el principal objetivo de esta clase es facilitar la tarea al programador, cada método está encargado de realizar una conversión de tipo de datos si se considera oportuno.

3.2.3. LA CLASE ONEVMTEMPLATE

Como se ha comentado anteriormente, la operación capaz de crear una nueva MV precisa de un parámetro de entrada de tipo String en el que se le indique la plantilla (*template*) de la máquina.

Como se puede observar en la Figura 3.6, el String es bastante tedioso de construir. Por ello se ha considerado oportuno facilitar esta tarea al programador y ofrecerle la oportunidad de poder crear la plantilla de una forma más cómoda mediante la construcción de un objeto de alto nivel.

```
String Template =
        "NAME = ttylinux\n" +
        "CPU = 1\n" +
        "MEMORY = 512\n" +
        "OS = [ boot = \"hd\", root = \"sda\" ]\n" +
        "DISK = [ source = \"/srv/cloud/disk0.qcow2\", target = \"hda\", readonly =\"no\" ]\n" +
        "NIC = [ NETWORK = \"Publica\"]\n" +
        "GRAPHICS = [ type = \"vnc\", listen = \"127.0.0.1\" ]\n" +
        "REQUIREMENTS = \"CPUSPEED > 1000\"\n";
```

Figura 3.6 – Ejemplo de *template* válido para la creación de una MV

La clase encargada de definir y dar soporte a este nuevo objeto es *ONEVMTemplate*. Está formada por métodos consultores y modificadores de los atributos que puede contener un *template*, y el método *toString*, capaz de transformar la información que contiene el objeto en un String válido para la creación de MVs en OpenNebula.

Tras el análisis de distintas plantillas válidas de MVs se ha podido comprobar que existen atributos obligatorios y otros opcionales en la creación del *template*. Es por ello que la clase dispone de dos constructores válidos para crear el objeto de alto nivel:

- El primero recibe como parámetros de entrada los atributos considerados obligatorios. El resto de atributos se inicializan con su valor por defecto para evitar problemas más adelante.

- El segundo no precisa de ningún parámetro de entrada, inicializando todos los atributos del objeto con los valores por defecto.

En la Tabla 3.3 se pueden observar todos los atributos que conforman el *template* así como si son de tipo obligatorio u opcional. También se muestra el tipo de dato de cada atributo.

Nombre del atributo	Tipo de dato	Obligatorio/Opcional
name	String	Obligatorio
cpu	double	Obligatorio
memory	int	Obligatorio
disk	String	Obligatorio

disk2	String	Opcional
nick	String	Obligatorio
os	String	Opcional
graphics	String	Opcional
requirements	String	Opcional
context	String	Opcional
rank	String	Opcional
features	String	Opcional

Tabla 3.3 – Atributos de un objeto *ONEVMTemplate*

3.3. OPERACIONES DESTINADAS A LA GESTIÓN DE HOSTS

En este subcapítulo se analizan, de forma análoga al subcapítulo anterior, las operaciones disponibles en el API destinadas a la gestión de los Hosts que dan soporte a las MVs. También se define el componente de alto nivel creado expresamente para este tipo de operaciones, *HostInfoResult*.

3.3.1. OPERACIONES DISPONIBLES EN LA CLASE OPENNEBULACLIENT

La clase *OpenNebulaClient* ofrece 5 métodos destinados a la gestión de los Hosts. Cada uno de ellos se corresponde directamente con su semejante en el API XML-RPC que OpenNebula ofrece. Todos informan de su estado a través de la herramienta Log4j y analizan los resultados de OpenNebula invocando al método *extractResult*. A continuación se analiza la funcionalidad de cada uno:

- ***createHost***: este método es capaz de crear un Host indicándole su nombre. También necesita recibir como parámetros de entrada el gestor de MVs, el gestor de información, el administrador y una variable booleana que indica si el Host pertenece al dominio administrativo o no.

Realiza una llamada remota a la operación *one.host.allocate*, perteneciente al API XML-RPC de OpenNebula. Con el identificador del nuevo Host recibido tras la invocación, se invoca al método *getHostInfo* para poder devolver un objeto de alto nivel de tipo *HostInfoResult*. Si durante la realización de la llamada ocurre algún problema o la respuesta recibida no es la esperada, el método aborta su ejecución y lanza la excepción *CloudException*.

- ***getHostInfo*:** mediante la ejecución de esta operación se puede obtener un objeto de alto nivel, de tipo *HostInfoResult*, que permite acceder a toda la información del Host cuyo identificador se corresponde con el que el método recibe como parámetro. Internamente realiza una llamada al método *one.host.info* del cual, si todo trascurre correctamente, se recibe la información del Host en formato XML. Si el resultado no es el esperado o la invocación remota sufre algún contratiempo, el método finaliza lanzando una *CloudExcepion*.

- ***deleteHost*:** para eliminar un Host de la infraestructura gestionada se debe invocar a este método. Para lograr su objetivo necesita recibir como parámetro de entrada el identificador del Host que se quiere borrar y con él, se invoca a la operación *one.host.delete* del API XML-RPC. Si surge algún problema durante su ejecución se lanza la excepción *CloudException*.

- ***enableHost*:** esta operación permite habilitar o deshabilitar un Host, dependiendo de lo que se le indique en el parámetro booleano que recibe en su invocación. También necesita recibir el identificador del Host al que se desea aplicar la operación. Puede lanzar la excepción *CloudException* si se produce algún error durante su ejecución.

- ***hostPoolInfo*:** este método permite obtener la información de todos los Hosts existentes. Devuelve un ArrayList de objetos HostInfoResult para poder acceder de forma sencilla a todos sus atributos. Para ello, en

primer lugar se realiza una invocación remota a la operación *one.hostpool.info* y, mediante el método *pool* de la clase *CloudOneResult*, se obtiene un vector con los identificadores de los Hosts. En segundo lugar se invoca al método *getHostInfo* para obtener los objetos *HostInfoResult*. Si durante su ejecución surge algún problema, el método aborta su ejecución lanzando la excepción *CloudException* que contendrá el motivo por el cual el método no ha alcanzado su objetivo.

En la Tabla 3.4 aparece el perfil de cada método y la correspondencia con su semejante en el API XML-RPC de OpenNebula.

Nombre del método	**Parámetros de entrada**	**Parámetros de salida**	**Correspondencia directa con las operaciones del API XML-RPC**
createHost	nombre del Host, del gestor de MVs, del gestor de información, del administrador y booleano dominio administrativo	Objeto *HostInfoResult*	Sí (*one.host.allocate*)
getHostInfo	Id del Host	Objeto *HostInfoResult*	Sí (*one.host.info)*
deleteHost	Id del Host	Vacío (*void*)	Sí (*one.host.delete)*
enableHost	Id del Host y booleano habilitar/deshabilitar	Vacío (*void*)	Sí (*one.host.enable)*
hostPoolInfo	Vacío (*void*)	ArrayList de objetos *HostInfoResult*	Sí (*one.hostpool.info)*

Tabla 3.4 – Métodos destinados a la gestión de Hosts

3.3.2. EL OBJETO DE ALTO NIVEL HOSTINFORESULT

Los objetos de alto nivel definidos en la clase *HostInfoResult* permiten acceder de forma sencilla a todos los atributos disponibles en las respuestas, en formato XML, de los métodos *one.host.info* y *one.hostpool.info* (disponibles en el API XML-RPC que ofrece OpenNebula). Por ejemplo, se puede acceder al número de MVs que alberga realizando una llamada al método *getNumberOfVMs* sobre un objeto de tipo *HostInfoResult*; sin la necesidad de procesar el XML que puede resultar tedioso.

La clase hereda de *CloudOneResult*, permitiéndole basar todos sus métodos consultores de los atributos en la llamada al método *evaluate* (disponible en la clase padre).

En la Tabla 3.5 se pueden observar todos los métodos que se ofrecen para este tipo de componente, su salida y su correspondencia con el resultado recibido en formato XML.

Nombre del método	**Tipo de parámetro de salida**	**Correspondencia con el XML**
constructor	*HostInfoResult*	No procede
getID	int	/HOST/ID
getName	String	/HOST/NAME
getState	String	/HOST/STATE*
getStateNumber	int	/HOST/STATE
getIM_MAD	String	/HOST/IM_MAD
getVM_MAD	String	/HOST/VM_MAD
getTM_MAD	String	/HOST/TM_MAD
getLastMonTime	long	/HOST/LAST_MON_TIME
getHID	int	/HOST/HOST_SHARE/HID
getDiskUsage	long	/HOST/HOST_SHARE/DISK_USAGE
getMemoryUsage	long	/HOST/HOST_SHARE/MEM_USAGE
getCPUUsage	long	/HOST/HOST_SHARE/CPU_USAGE
getMaxDisk	long	/HOST/HOST_SHARE/MAX_DISK
getMaxMemory	long	/HOST/HOST_SHARE/MAX_MEM
getMaxCPU	long	/HOST/HOST_SHARE/MAX_CPU
getFreeDisk	long	/HOST/HOST_SHARE/FREE_DISK
getFreeMemory	long	/HOST/HOST_SHARE/FREE_MEM
getFreeCPU	long	/HOST/HOST_SHARE/FREE_CPU

getUsedDisk	long	/HOST/HOST_SHARE/USED_DISK
getUsedMemory	long	/HOST/HOST_SHARE/USED_MEM
getUsedCPU	long	/HOST/HOST_SHARE/USED_CPU
getNumberOfVMs	int	/HOST/HOST_SHARE/RUNNING_VMS
getArchitecture	String	/HOST/TEMPLATE/ARCH
getCPUSpeed	long	/HOST/TEMPLATE/CPUSPEED
getHypervisor	String	/HOST/TEMPLATE/HYPERVISOR
getModelName	String	/HOST/TEMPLATE/MODELNAME
getTotalCPU	long	/HOST/TEMPLATE/TOTALCPU
getTotalMemory	long	/HOST/TEMPLATE/TOTALMEMORY
getNetRX	int	/HOST/TEMPLATE/NETRX
getNetTX	int	/HOST/TEMPLATE/NETTX

*La correspondencia no es directa.

Tabla 3.5 – Métodos disponibles en la clase *HostInfoResult*

3.4. OPERACIONES DESTINADAS A LA GESTIÓN DE REDES VIRTUALES

Este subcapítulo trata sobre las operaciones disponibles en el API destinadas a la gestión de VNs. De manera análoga a los dos subcapítulos anteriores, también se profundiza en la funcionalidad y las operaciones que ofrece la clase *VNInfoResult,* capaz de crear los objetos de alto nivel que contienen los atributos de las redes virtuales.

3.4.1. OPERACIONES DISPONIBLES EN LA CLASE OPENNEBULACLIENT

Como ya se ha mencionado en otros capítulos, la clase *OpenNebulaClient* incluye todas las operaciones que ofrece el API. Entre ellas se encuentran las que están destinadas a la gestión de VNs, que se analizan en este apartado.

En concreto, el API dispone de 4 operaciones destinadas a la gestión de VNs. Cada una de ellas está directamente relacionada con su equivalente en el API XML-

RPC que ofrece OpenNebula, como puede observarse en la columna "*Correspondencia directa con las operaciones del API XML-RPC*" de la Tabla 3.6.

Nombre del método	**Parámetros de entrada**	**Parámetros de salida**	**Correspondencia directa con las operaciones del API XML-RPC**
createVN	plantilla de la nueva VN	Objeto VN*InfoResult*	Sí (*one.vn.allocate*)
getVNInfo	Id de la VN	Objeto *VNInfoResult*	Sí (*one.vn.info)*
deleteVN	Id de la VN	Vacío (*void*)	Sí (*one.vn.delete)*
vnPoolInfo	Vacío (*void*)	ArrayList de objetos *VNInfoResult*	Sí (*one.vnpool.info)*

Tabla 3.6 – Métodos destinados a la gestión de VNs

Seguidamente se analizan una a una las operaciones sobre VNs disponibles en el API. Cabe señalar que, al igual que el resto de operaciones que ofrece el API, utilizan Log4j para indicar su estado y, tras la invocación remota a la operación correspondiente del API XML-RPC, se realiza una llamada al método *extractResult* para analizar el resultado obtenido. Las operaciones disponibles son:

- ***createVN***: este método permite crear una nueva red virtual, pasándole como parámetro de entrada la plantilla de ésta. Realiza una invocación remota al método *one.vn.allocate* del API XML-RPC de OpenNebula. Puede ocurrir algún error durante su invocación, por ejemplo de conexión, o puede que la respuesta del método no sea la esperada; en estos casos, *createVN* es capaz de detectar el problema y finalizar su ejecución lanzando una *CloudException* informando del error.

- ***getVNInfo***: para obtener la información de una red virtual, se puede invocar a este método con el identificador de la VN de la que se desea información. El método se encarga de realizar una invocación remota a su semejante (*one.vn.info*) en el API XML-RPC con el identificador de la

VN. Si se detecta algún problema durante la llamada o en el resultado obtenido, se finaliza la ejecución lanzando una excepción de tipo *CloudException*.

- ***deleteVN***: esta operación permite eliminar una red virtual. Precisa que se le indique como parámetro de entrada el identificador correspondiente con la VN que se desea borrar. Internamente se realiza una llamada remota a la operación *one.vn.delete* del API XML-RPC que ofrece el GMVs con el que se está trabajando. Como el API está dotado de mecanismos para detectar problemas durante este tipo de invocaciones remotas, si ocurre algún imprevisto se aborta la ejecución lanzando la excepción *CloudException*. El método también puede responder de esta forma si el resultado de la operación no es el esperado.

- ***vnPoolInfo***: este método permite obtener la totalidad o parte de la información de las VNs disponibles en la infraestructura. La cantidad de información recibida dependerá de las credenciales de usuario con las que se invoque. Mientras que un usuario cualquiera sólo recibirá información de las VNs que haya creado, el administrador (*oneadmin*) tendrá la posibilidad de visualizar la información completa. Devuelve un ArrayList de objetos *VNInfoResult* tras invocar de forma remota a la operación *one.vnpool.info*, extraer los identificadores de cada VN y obtener el objeto de alto nivel para cada uno mediante el método *getVNInfo*, analizado anteriormente. Si ocurre algún error durante su ejecución, ésta es abortada lanzando la excepción *CloudException*.

3.4.2. EL OBJETO DE ALTO NIVEL VNINFORESULT

Como el resto de componentes de alto nivel estudiados hasta ahora, el objetivo principal de la creación del objeto *VNInfoResult* es facilitar al programador el acceso a todos los atributos incluidos en la respuesta en formato XML de OpenNebula.

La clase *VNInfoResult* es la encargada de definir este tipo de objeto. Está formada por 7 métodos cuyo perfil se recoge en la Tabla 3.7. Como el resto de clases encargadas de definir los objetos de alto nivel disponibles en el API, hereda de la clase *CloudOneResult*, permitiéndole hacer uso de todas sus operaciones. Por ello, todos los métodos pueden realizar una invocación al método *evaluate* para obtener el valor de una determinada etiqueta del resultado en formato XML.

Nombre del método	Tipo de parámetro de salida	Correspondencia con el XML
constructor	Objeto *VNInfoResult*	No procede
getName	String	/VNET/NAME
getType	String	/VNET/TYPE
getID	int	/VNET/ID
getUID	int	/VNET/UID
getBridge	String	/VNET/BRIDGE
getTotalLeases	int	Número de etiquetas LEASE

Tabla 3.7 – Métodos disponibles en la clase *VNInfoResult*

3.5. OPERACIONES DESTINADAS A LA GESTIÓN DE USUARIOS

En este subcapítulo se encuentran las operaciones disponibles en el API que están destinadas a la gestión de usuarios. También se define el componente de alto nivel creado para este tipo de operaciones User*InfoResult* y las operaciones disponibles en la clase que lleva el mismo nombre.

3.5.1. OPERACIONES DISPONIBLES EN LA CLASE OPENNEBULACLIENT

La clase *OpenNebulaClient* ofrece un total de 4 operaciones destinadas a gestionar a los usuarios de la infraestructura. La Tabla 3.8 recoge el perfil de cada operación y su correspondencia con las operaciones del API XML-RPC que ofrece OpenNebula.

Nombre del método	**Parámetros de entrada**	**Parámetros de salida**	**Correspondencia directa con las operaciones del API XML-RPC**
createUser	nombre de usuario y contraseña	Objeto *UserInfoResult*	Sí (*one.user.allocate*)
getUserInfo	Id del usuario	Objeto *UserInfoResult*	Sí (*one.user.info)*
deleteUser	Id del usuario	Vacío (*void*)	Sí (*one.user.delete)*
userPoolInfo	Vacío (*void*)	ArrayList de objetos*UserInfoResult*	Sí (*one.userpool.info)*

Tabla 3.8 – Métodos destinados a la gestión de usuarios

Analicemos la funcionalidad de cada método:

- ***createUser*:** mediante la invocación a esta operación es posible crear un nuevo usuario. Para ello el método requiere el nombre y la contraseña como parámetros de entrada. Internamente, se realiza una invocación remota al método *one.user.allocate*. Se ha de tener en cuenta que sólo el administrador (cuyas credenciales se corresponden con *oneadmin*) puede llevar a cabo esta operación con éxito. Los usuarios normales no tiene permiso para ello. Si no existen permisos, la invocación remota falla o el resultado de esta invocación no es el esperado, se aborta la ejecución del método lanzando una *CloudException* que contendrá el motivo del error.

- ***getUserInfo*:** para obtener la información de un usuario determinado, es posible invocar a este método indicándole el identificador del usuario del que queremos la información. Previamente, es necesario comprobar que

las credenciales de usuario coinciden con las del administrador si se quiere ejecutar la operación con éxito. Con el identificador se invoca al método *one.user.info* del API XML-RPC de OpenNebula para obtener el XML que después se procesará y con su información se construirá un objeto de alto nivel de tipo UserInfoResult que será devuelto por el método. Si se produce algún error durante su ejecución, será lanzada la excepción *CloudException*.

- ***deleteUser*:** este método permite eliminar un usuario. Es necesario contar con las credenciales del administrador para realizar la operación con éxito. Realiza una invocación remota al método *one.user.delete* del API XML-RPC que ofrece OpenNebula con el identificador del usuario que se desea borrar. Si las credenciales del usuario no son las correctas o la llamada no transcurre con normalidad, será lanzada la excepción *CloudException*.

- ***userPoolInfo*:** esta operación permite acceder a la información de todos los usuarios registrados en la infraestructura. Nuevamente es necesario ser el administrador para poder llevar a cabo la operación. Tras la invocación remota al método *one.userpool.info*, se obtienen los identificadores de cada usuario (con el método *pool* de la clase *CloudOneResult*) y se invoca, para cada uno, el método *getUserInfo*, construyendo un ArrayList de objetos *UserInfoResult* que es devuelto por esta operación. Si se produce algún error durante su ejecución, ésta se abortará lanzando una nueva *CloudException*.

3.5.2. EL OBJETO DE ALTO NIVEL USERINFORESULT

Con el objetivo de acceder fácilmente a todos los atributos de los usuarios disponibles en la respuesta en formato XML de OpenNebula y siguiendo la misma

estrategia que con el resto de operaciones del API, se ha definido un objeto de alto nivel para cumplir esta labor, y que recibe el nombre de *UserInfoResult*.

Este objeto está definido en la clase que porta el mismo nombre, y que ofrece 5 métodos cuyo perfil recoge la Tabla 3.9. Esta clase hereda de *CloudOneResult*, que contiene las acciones comunes en el tratamiento de los resultados de OpenNebula en formato XML.

Nombre del método	**Tipo de parámetro de salida**	**Correspondencia con el XML**
constructor	Objeto *UserInfoResult*	No procede
getName	String	/USER/NAME
getID	int	/USER/ID
isEnabled	Boolean	/USER/ENABLED
getPassword	String	/USER/PASSWORD

Tabla 3.9 – Métodos disponibles en la clase *UserInfoResult*

3.6. OPERACIONES DESTINADAS A LA GESTIÓN DE CLUSTERS DE MVs

Este subcapítulo se dedica a analizar las operaciones disponibles en el API que están destinadas a la gestión de clusters de MVs. También se expone la funcionalidad de la clase que da soporte a estas operaciones, *OneVMCluster*.

3.6.1. OPERACIONES DISPONIBLES EN LA CLASE OPENNEBULACLIENT

Como se ha citado anteriormente en este documento, no existe soporte para este tipo de operaciones en el API Java que OpenNebula ofrece, y tampoco es posible hallarlo en el caso del API XML-RPC. Pero, tras analizar el uso que se le iba a dar al API desarrollado, se considera necesario este apoyo ya que suele ser habitual trabajar con grupos de MVs con la misma plantilla (*template*) para lograr un objetivo común. Si bien es cierto que se podrían crear varias MVs con la misma plantilla, una a una,

también lo es el hecho de que si se puede realizar esta tarea con una sola instrucción se facilita el trabajo del programador, objetivo primordial que intenta alcanzar este API.

En la clase *OpenNebulaClient* se encuentran disponibles un total de 12 operaciones cuyo objetivo es permitir la gestión de clusters de MVs. El perfil de todas ellas queda plasmado en la Tabla 3.10.

Nombre del método	Parámetros de entrada	Parámetros de salida
createCluster	Número de MVs que forman el cluster y la plantilla de éstas.	Objeto *OneVMCluster*
decrementCluster	Objeto *OneVMCluster* y el número de MVs a decrementar	Objeto *OneVMCluster*
incrementCluster	Objeto *OneVMCluster*, el número de MVs a incrementar y la plantilla de la nuevas MVs.	Objeto *OneVMCluster*
finalizeCluster	Objeto *OneVMCluster*	Vacío (*void*)
shutdownCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
holdCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
releaseCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
stopCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
cancelCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
suspendCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
resumeCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*
restartCluster	Objeto *OneVMCluster*	Objeto *OneVMCluster*

Tabla 3.10 – Métodos destinados a la gestión de clusters de MVs

Seguidamente se analizan las operaciones recogidas en la anterior tabla pero, para no repetir lo ya explicado en anteriores subcapítulos, se ha optado por agrupar la explicación de los métodos capaces de realizar una acción determinada con el cluster de MVs, ya que su explicación se corresponde con sus semejantes en el caso de la gestión de MVs, solo que ahora con un grupo de éstas.

- ***createCluster*:** mediante este método es posible inicializar un número de MVs concreto, que se recibe como parámetro de entrada. También es

necesario para llevar a cabo la operación, recibir una plantilla válida con la que serán creadas todas las MVs. Internamente se realizan tantas llamadas a la operación *submitVM* como MVs se deseen crear. El método devuelve un objeto de alto nivel OneVMCluster, al que se le pueden consultar el número de MVs que forman el cluster y los identificadores de todas ellas. Si durante su ejecución ocurre algún problema, se aborta la ejecución del método y se lanza la excepción *CloudException*.

- ***decrementCluster*:** este método se encarga de decrementar la cantidad de MVs existente en el cluster. Requiere recibir como parámetros el objeto de alto nivel que contiene el cluster que se desea disminuir y el número de MVs a decrementar. Como todas las MVs son iguales, es decir, tienen todas el mismo *template*, no importa las que sean eliminadas, luego mediante la llamada al método finalizeVM se eliminan las MVs y se actualizan los valores del objeto *OneVMCluster*,que es devuelto por *decrementCluster*. La excepción *CloudException* puede ser lanzada si ocurre algún error durante su ejecución.

- ***incrementCluster*:** para incrementar el número de MVs que forman el cluster, es necesario recibir como parámetros de entrada el objeto de alto nivel *OneVMCluster*, el número de MVs a incrementar y la plantilla de éstas. Mediante la invocación al método *submitVM* tantas veces como el número de MVs a incrementar indique, se crean las nuevas MVs. Seguidamente se actualizan los valores de los dos atributos del objeto *OneVMCluster*. Este objeto se devuelve como parámetro de salida. Si surge algún inconveniente durante su ejecución, se lanza una nueva *CloudException* con el motivo del error.

- **resto de operaciones:** para realizar cualquier tipo de acción de las que se disponía en la gestión de MVs (reiniciar, parar, finalizar…) sobre un cluster, hay disponibles en el API 9 operaciones destinadas a ello. El

funcionamiento de todas ellas es común: realizan tantas llamadas a sus métodos afines disponibles para gestionar una sola MV como máquinas tenga el cluster. Por ejemplo, el método *restartCluster* invocará a *restartVM* varias veces, cada una de ellas con el identificador de una de las MVs que forman el cluster. Los identificadores se obtienen del atributo del objeto *OneVMCluster* encargado de contenerlos en forma de vector y que se recibe como parámetro en cada método. Todas las operaciones pueden lanzar una *CloudException* si ocurre algún problema durante su ejecución.

3.6.2. LA CLASE ONEVMCLUSTER

La clase encargada de definir el objeto que permite gestionar un grupo de MVs a la vez recibe el nombre de OneVMCluster. Cuenta con dos atributos, el número de MVs que forman el cluster y un ArrayList con los identificadores de todas las MVs.

Cuenta con dos métodos consultores y otros dos modificadores, uno por cada atributo con el que cuenta la clase. Éstos son utilizados por los métodos que ofrece el API para la gestión de clusters de MVs para alcanzar sus objetivos.

Tambien contiene un método que devuelve la información del cluster: cuántas máquinas lo forman y cuáles son sus identificadores. El método recibe el nombre de *toString*.

4. CONCLUSIONES

Para cerrar este documento, en este último capítulo se exponen las conclusiones que la autora extrae tras la realización del PFC.

Los objetivos planteados al comienzo se han podido llevar a cabo de forma satisfactoria. Se ha construido un API en Java que facilita la tarea a los programadores encargados de realizar aplicaciones que requieran la intervención con el GMVs OpenNebula. Y, lo más importante, de forma sencilla. Mediante la creación de los objetos de alto nivel se consigue acceder de forma simple a todos los atributos de éstos, cosa imposible de llevar a cabo en el API en Java que OpenNebula ofrece a día de hoy.

Además, tras analizar el uso que se le iba a dar al API en el ámbito científico, se optó por dotarlo de operaciones que soportaran la gestión de clusters de MVs ya que, en este ámbito, suele ser más usual lanzar varias máquinas virtuales con la misma plantilla para que desarrollen entre todas un objetivo común, que trabajar con MVs de forma aislada. Con ello se cumple otro de los objetivos planteados inicialmente.

Pero para llegar al punto donde actualmente se encuentra el PFC, primeramente ha sido necesario un período de documentación sobre el paradigma de la computación en Cloud, concepto absolutamente desconocido en un principio y que a día de hoy resulta totalmente familiar. Se han dedicado semanas de investigación desechando mucha información contradictoria sobre este paradigma, ya que muchos son los que opinan sobre el tema pero pocos los que lo hacen con rigurosidad.

También ha supuesto un pequeño esfuerzo el hecho de trabajar con un entorno de programación sobre el que no se había trabajado hasta el momento. Pero la experiencia sobre este tipo de IDE ha resultado satisfactoria, tanto que, en futuros

trabajos, se opte por la elección de este tipo de entornos más sofisticados para trabajar y no sobre otros más simples con los que se había trabajado hasta ahora.

El hecho de usar la herramienta Subversión y de disponer de un repositorio donde almacenar el trabajo realizado, además de Internet, también ha facilitado la realización del trabajo desde casa, una ventaja analizando el contexto temporal sobre el que se ha llevado a cabo la realización del proyecto.

Otro aspecto, en este caso puramente de desarrollo, ha sido la necesidad de analizar y hacer un correcto uso de un par de herramientas que se han empleado en el API y que se desconocían hasta ahora: Log4j y XPath, pero su uso no ha supuesto ningún problema remarcable.

Desde el punto de vista personal de la autora, el proyecto se planteó como un desafío personal, ya que no quería realizar un trabajo cualquiera, sino que deseaba aprovechar esta oportunidad para colaborar en un proyecto más relevante en el ámbito de la investigación. Por ello se optó por la realización del trabajo que en este documento se expone, colaborando en el proyecto PAID-06-09-2810. Es reconfortante saber que los resultados obtenidos van a servir para un proyecto de mayor repercusión y que no se va a quedar simplemente almacenado. Además, los conocimientos aprendidos y la experiencia ganada durante el desarrollo del proyecto, han servido para tomar la decisión de continuar la formación académica por la rama de la Computación de Altas Prestaciones.

APÉNDICE A

DESCRIPCIÓN DE UN CASO DE ESTUDIO

En este apéndice de la memoria se muestra un caso de estudio realizado para observar el funcionamiento del API desarrollado. Seguidamente se muestra el código desarrollado para este caso de estudio:

```
package org.grycap.cloud.opennebula.test;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.grycap.cloud.opennebula.ONEVMTemplate;
import org.grycap.cloud.opennebula.OneVMCluster;
import org.grycap.cloud.opennebula.OpenNebulaClient;

public class CasoDeUsoOpenNebulaClient {

 public static void main(String[] args) {

  PropertyConfigurator.configure("log4j.properties");
  Logger log =
Logger.getLogger(CasoDeUsoOpenNebulaClient.class);

  ONEVMTemplate template = new ONEVMTemplate();
  template.setName("opaljeos");
  template.setCPU(1);
  template.setMemory(512);
  template.setOS("[ boot = \"hd\", root = \"sda\" ]");
  template.setDisk("[ source =
\"/srv/cloud/images/jeos/kvm_jeos_opal" +
    "/disk0.qcow2\", " + "target = \"hda\", readonly =\"no\"
]");
  template.setNic("[ NETWORK = \"Publica\"]");
  template.setGraphics("[ type = \"vnc\", listen =
\"127.0.0.1\" ]");
  template.setRequirements("\"CPUSPEED > 1000\"");
  template.setContext("[ files =
\"/srv/cloud/images/cntxtlzr/init.sh"
    + "/srv/cloud/images/cntxtlzr/cntxtlzr.tgz\", target =
\"hdb\" ]");
```

```
try{
      OpenNebulaClient onc = new
OpenNebulaClient("http://dellblade01.itaca.upv.es" +
 ":2633/RPC2","amcaar:318ccc3fef86513000492de450870fd433af76bd"
);

OneVMCluster cluster = onc.createCluster(2,
template.toString());
      log.info(cluster.toString());
      onc.incrementCluster(cluster, 1, template.toString());
      log.info(cluster.toString());
      onc.decrementCluster(cluster, 1);
      log.info(cluster.toString());
      onc.finalizeCluster(cluster);
      log.info(cluster.toString());
      onc.vmPoolInfo();

   }catch(Exception ex){
    log.fatal("An exception occurred: " + ex.getMessage());
   }

 }

}
```

En este ejemplo de uso del API se ha creado un objeto ONEVMTemplate para la posterior creación de MVs. Se han modificado varios de sus atributos hasta crear la plantilla deseada. Seguidamente se ha construido un objeto OpenNebulaClient sobre el que se pueden invocar todas las operaciones que ofrece el API.

Se ha optado por mostrar un ejemplo de las operaciones destinadas a la gestión de clusters por ser las más sofisticadas, ya que internamente también hacen uso de otras operaciones ofrecidas. Como se puede observar en el código, se crea un cluster con 2 MVs. Seguidamente se amplía a una más para luego decrementarla. Finalmente se apagan todas las MVs que forman el cluster.

Para observar el estado del cluster tras cada operación, se invoca al método *toString* sobre el objeto *OneVMCluster*.

A continuación se puede observar la salida producida por este pequeño programa:

```
Java - Eclipse SDK
File  Edit  Navigate  Search  Project  Run  Window  Help
Team Synchr...
Problems  @ Javadoc  Declaration  Console
<terminated> CasoDeUsoOpenNebulaClient [Java Application] C:\Program Files\Java\jre1.6.0_02\bin\javaw.exe (17/08/2010 15:59:23)
DEBUG 2010-08-17 15:59:24,653 [class]  - Trying to create cluster
DEBUG 2010-08-17 15:59:24,655 [class]  - Trying to create a Virtual Machine
INFO  2010-08-17 15:59:25,907 [class]  - Operation createVM performed successfully
DEBUG 2010-08-17 15:59:25,907 [class]  - Trying to get VM info with id = 507
INFO  2010-08-17 15:59:26,045 [class]  - Operation getVMInfo performed successfully
DEBUG 2010-08-17 15:59:26,045 [class]  - getVMInfo: Obtained XML result: <VM><ID>507</ID><UID>2</UID><NAME>opaljeos</NAME><LAST_POLL>0</LAST_POLL><ST
DEBUG 2010-08-17 15:59:26,087 [class]  - Trying to create a Virtual Machine
INFO  2010-08-17 15:59:27,381 [class]  - Operation createVM performed successfully
DEBUG 2010-08-17 15:59:27,381 [class]  - Trying to get VM info with id = 508
INFO  2010-08-17 15:59:27,522 [class]  - Operation getVMInfo performed successfully
DEBUG 2010-08-17 15:59:27,522 [class]  - getVMInfo: Obtained XML result: <VM><ID>508</ID><UID>2</UID><NAME>opaljeos</NAME><LAST_POLL>0</LAST_POLL><ST
INFO  2010-08-17 15:59:27,533 [CasoDeUsoOpenNebulaClient]  - The cluster consists of 2 VMs, whose identifiers are: 507 508
DEBUG 2010-08-17 15:59:27,533 [class]  - Trying to increment cluster
DEBUG 2010-08-17 15:59:27,533 [class]  - Trying to create a Virtual Machine
INFO  2010-08-17 15:59:28,898 [class]  - Operation createVM performed successfully
DEBUG 2010-08-17 15:59:28,898 [class]  - Trying to get VM info with id = 509
INFO  2010-08-17 15:59:29,034 [class]  - Operation getVMInfo performed successfully
DEBUG 2010-08-17 15:59:29,035 [class]  - getVMInfo: Obtained XML result: <VM><ID>509</ID><UID>2</UID><NAME>opaljeos</NAME><LAST_POLL>0</LAST_POLL><ST
INFO  2010-08-17 15:59:29,042 [CasoDeUsoOpenNebulaClient]  - The cluster consists of 3 VMs, whose identifiers are: 507 508 509
DEBUG 2010-08-17 15:59:29,042 [class]  - Trying to decrement cluster
DEBUG 2010-08-17 15:59:29,042 [class]  - Trying to finalize VM with id = 509
INFO  2010-08-17 15:59:29,313 [class]  - Operation finalizeVM performed successfully
DEBUG 2010-08-17 15:59:29,313 [class]  - the VM with id = 509 was finalized
INFO  2010-08-17 15:59:29,313 [CasoDeUsoOpenNebulaClient]  - The cluster consists of 2 VMs, whose identifiers are: 507 508
DEBUG 2010-08-17 15:59:29,313 [class]  - Trying to finalize cluster
DEBUG 2010-08-17 15:59:29,313 [class]  - Trying to finalize VM with id = 508
INFO  2010-08-17 15:59:29,623 [class]  - Operation finalizeVM performed successfully
DEBUG 2010-08-17 15:59:29,623 [class]  - the VM with id = 508 was finalized
DEBUG 2010-08-17 15:59:29,623 [class]  - Trying to finalize VM with id = 507
INFO  2010-08-17 15:59:29,836 [class]  - Operation finalizeVM performed successfully
DEBUG 2010-08-17 15:59:29,836 [class]  - the VM with id = 507 was finalized
INFO  2010-08-17 15:59:29,836 [CasoDeUsoOpenNebulaClient]  - The cluster consists of 0 VMs, whose identifiers are:
DEBUG 2010-08-17 15:59:29,836 [class]  - Trying to obtain VM pool information
INFO  2010-08-17 15:59:29,917 [class]  - Operation vmpoolInfo performed successfully
DEBUG 2010-08-17 15:59:29,917 [class]  - vmpoolInfo: Obtained XML result: <VM_POOL></VM_POOL>
DEBUG 2010-08-17 15:59:29,920 [class]  - El número de objetos VM es: 0
```

Figura A – Salida obtenida tras la ejecución del caso de estudio

APÉNDICE B

INSTALACIÓN DE SUBVERSION EN ECLIPSE GALILEO

En este apartado se exponen los pasos a seguir para poder instalar Subversion en Eclipse Galileo:

1. Ir a Help/Install New Software...
2. Agregar
    1. Galileo -> http://download.eclipse.org/releases/galileo
    2. Subversive SVN Connectors -> http://community.polarion.com/projects/subversive/download/eclipse/2.0/galileo-site/
3. de Galileo agregar
    1. Colaboration/Subversive SVN Team Provider (Incubation) 0.7.8.I20090506-1500
4. Instalar y reiniciar el IDE.
5. Repetir el proceso agregando de Subversive SVN Connectors:
    1. Subversive SVN Connectors/Subversive SVN Connectors 2.2.0.I20090505-1500
    2. Subversive SVN Connectors/Native JavaHL 1.5 Implementation (Optional) 2.2.0.I20090505-1500
    3. Subversive SVN Connectors/ SVNKit 1.2.2 Implementation (Optional) 2.2.0.I20090505-1500
6. Reiniciar el IDE.

APÉNDICE C

EJEMPLOS DE RESULTADOS XML DEVUELTOS POR OPENNEBULA

En este apéndice se encuentran los ejemplos de los resultados obtenidos tras la invocación las operaciones información de MVs del API XML-RPC de OpenNebula en formato XML.

Resultado obtenido tras la invocación al método *one.vm.info*:


```
<VM>
    <ID>200</ID>
    <UID>1</UID>
    <NAME>opaljeos</NAME>
    <LAST_POLL>1273159657</LAST_POLL>
    <STATE>6</STATE>
    <LCM_STATE>0</LCM_STATE>
    <STIME>1273156531</STIME>
    <ETIME>1273159701</ETIME>
    <DEPLOY_ID>one-200</DEPLOY_ID>
    <MEMORY>524288</MEMORY>
    <CPU>0</CPU>
    <NET_TX>0</NET_TX>
    <NET_RX>0</NET_RX>
    <TEMPLATE>
        <CONTEXT>
                <FILES>/srv/cloud/images/cntxtlzr/cntxtlzr.tgz</FILES>
                <TARGET>hdb</TARGET>
        </CONTEXT>
        <CPU>1</CPU>
        <DISK>
            <READONLY>no</READONLY>
            <SOURCE>/srv/cloud/jeos/kvm_jeos_opal/disk0.qcow2</SOURCE>
            <TARGET>hda</TARGET>
        </DISK>
        <GRAPHICS>
            <LISTEN>127.0.0.1</LISTEN>
            <TYPE>vnc</TYPE>
        </GRAPHICS>
```

```
        <MEMORY>512</MEMORY>
        <NAME>opaljeos</NAME>
        <NIC>
            <BRIDGE>br0</BRIDGE>
            <IP>158.42.167.64</IP>
            <MAC>40:10:10:10:10:21</MAC>
            <NETWORK>Publica</NETWORK>
            <VNID>27</VNID>
        </NIC>
        <OS>
            <BOOT>hd</BOOT>
            <ROOT>sda</ROOT>
        </OS>
        <RANK>FREECPU</RANK>
        <REQUIREMENTS>CPUSPEED > 1000</REQUIREMENTS>
        <VMID>200</VMID>
    </TEMPLATE>
    <HISTORY>
        <SEQ>0</SEQ>
        <HOSTNAME>dellblade03</HOSTNAME>
        <HID>3</HID>
        <STIME>1273156561</STIME>
        <ETIME>1273159697</ETIME>
        <PSTIME>1273156561</PSTIME>
        <PETIME>1273156564</PETIME>
        <RSTIME>1273156564</RSTIME>
        <RETIME>1273159697</RETIME>
        <ESTIME>1273159697</ESTIME>
        <EETIME>1273159697</EETIME>
        <REASON>0</REASON>
    </HISTORY>
</VM>
```


Resultado obtenido tras la invocación al método *one.vmpool.info*:


```
<VM_POOL>
    <VM>
        <ID>136</ID>
        <UID>2</UID>
        <USERNAME>amcaar</USERNAME>
        <NAME>ttylinux</NAME>
        <LAST_POLL>0</LAST_POLL>
        <STATE>2</STATE>
```

```
        <LCM_STATE>0</LCM_STATE>
        <STIME>1272878238</STIME>
        <ETIME>0</ETIME>
        <DEPLOY_ID></DEPLOY_ID>
        <MEMORY>0</MEMORY>
        <CPU>0</CPU>
        <NET_TX>0</NET_TX>
        <NET_RX>0</NET_RX>
    </VM>

    <VM>
        <ID>137</ID>
        <UID>2</UID>
        <USERNAME>amcaar</USERNAME>
        <NAME>ttylinux</NAME>
        <LAST_POLL>0</LAST_POLL>
        <STATE>2</STATE>
        <LCM_STATE>0</LCM_STATE>
        <STIME>1272878826</STIME>
        <ETIME>0</ETIME>
        <DEPLOY_ID></DEPLOY_ID>
        <MEMORY>0</MEMORY>
        <CPU>0</CPU>
        <NET_TX>0</NET_TX>
        <NET_RX>0</NET_RX>
    </VM>
</VM_POOL>
```

BIBLIOGRAFÍA Y REFERENCIAS

[1] Miller, Michael: *Cloud Computing, Web-Based Applications That Change the Way You Work and Collaborate Online*. Indianapolis, Que, (2008). ISBN: 978-0768686227.

[2] Sotomayor, B., Montero, R.S., Llorente, I.M., Foster, I.:*Capacity leasing in cloud systems using the opennebula engine*. In: Workshop on Cloud Computing and its Applications 2008 (CCA08).

[3] Ruby. http://www.ruby-lang.org/es/

[4] Phyton. http://www.python.org/

[5] API Java de OpenNebula (2010). http://opennebula.org/doc/oca/java/

[6] Laurent, S. St., Johnston, J., Dumbill, E.: *Programming Web Services with XML-RPC*. O'Reilly (2001). ISBN: 0-596-00119-3.

[7] GMail. http://www.google.com/apps/intl/es/business/gmail.html

[8] Google Sites. http://www.google.com/apps/intl/es/business/sites.html

[9] Granneman, S.: *Google Apps Deciphered: Compute in the Cloud to Streamline Your Desktop*. Prentice Hall (2008). ISBN: 978-0-13-700776-9.

[10] Google App Engine. http://code.google.com/intl/es-ES/appengine/

[11] Amazon EC2. http://aws.amazon.com/ec2/

[12] Ejemplo de uso de Cloud Computing, Zynga. http://www.rightscale.com/customers/zynga-grows-to-1-social-gaming-site-with-rightscale.php

[13] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L.,Zagorodnov, D.: *The eucalyptus open-source cloud-computing system*. In: Proceedings of 9th IEEE International Symposium on Cluster Computing and the Grid. (2009).

[14] Abiquo. http://www.abiquo.com/

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield. Xen and the Art of Virtualization. Proceedings of the nineteenth ACM symposium on Operating systems principles, pp 164-177, Bolton Landing, NY, USA (2003).

[16] Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: *KVM: The linux virtual machine monitor.* In: OLS '07: The 2007 Ottawa Linux Symposium, pp. 225–230. (2007).

[17] VMware. http://www.vmware.com

[18] Virtual Box. http://www.virtualbox.org/

[19] Java. http://www.java.com/es/

[20] Holzner, S., Rosenblatt, B.: *Eclipse: A Java Developer's Guide.* O' Reilly Media (2004). ISBN: 978-0596006419.

[21] Eclipse Galileo. http://www.eclipse.org/galileo/

[22] Eclipse Helios. http://www.eclipse.org/helios/

[23] Nagel, William A.: *Subversion version control: using the Subversion version control system in development projects.* Prentice Hall Professional Technical Reference (2005). ISBN: 0131855182.

[24] XML-RPC. http://www.xmlrpc.com/

[25] Simpson, John E.: *XPath and XPointer.* O'Reilly (2002). ISBN: 0-596-002912

[26] Watt, A. *XPath essentials.* Wiley (2002). ISBN: 0-471-20548-6

[27] Perry, J. Steven: *Log4j.* O'Reilly Media (2009). ISBN: 978-1449379735.

[28] Don't Use System.out.println! Use Log4j. Vipan Singla (2001) http://www.vipan.com/htdocs/log4jhelp.html

[29] Gulcu, Ceky: *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java*. QOS.ch (2003). ISBN-13: 978-2970036906.

[30] API XML-RPC 1.4 OpenNebula. http://www.opennebula.org/documentation:rel1.4:api