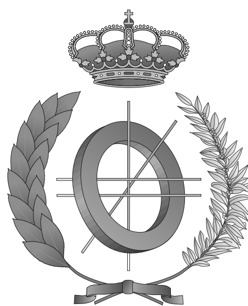

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

IceCloud: Diseño e implementación de un servicio autónomo para gestión y despliegue de aplicaciones distribuidas sobre un grid heterogéneo

Laura Núñez Villa

Septiembre, 2014

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

IceCloud: Diseño e implementación de un servicio autónomo para gestión y despliegue de aplicaciones distribuidas sobre un grid heterogéneo

Autor: Laura Núñez Villa
Director: Dr. David Villa Alises

Septiembre, 2014

Laura Núñez Villa

Ciudad Real – Spain

E-mail: Laura.Nunez2@alu.uclm.es
Web site:

© 2014 Laura Núñez Villa

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Se permite la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.3 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Una copia de esta licencia esta incluida en el apéndice titulado «GNU Free Documentation License».

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Resumen

Actualmente, asistimos a una revolución tecnológica basada en el concepto emergente de «computación en la nube» (*Cloud Computing*). Este concepto ha desencadenado un cambio en la forma de realizar las operaciones fundamentales de muchas empresas y usuarios particulares. Aunque para muchas personas la informática «en la nube» es un término confuso, cada día aumenta la demanda de servicios y recursos a través de Internet y las plataformas que ofrecen acceso flexible a recursos de computación son cada vez más frecuentes y completas.

En este Proyecto Fin de Carrera se realizará una aproximación al concepto de Cloud Computing y se presentará el desarrollo de una herramienta que ofrezca una plataforma de Cloud-Computing del tipo *Infraestructura como Servicio*.

Esta herramienta, llamada *IceCloud*, ofrece un entorno distribuido heterogéneo que se muestra al usuario como una serie de nodos de cómputo diversos en los que ejecutar sus aplicaciones de forma dinámica, permitiendo la adquisición y liberación de esos recursos según las necesidades de cada momento.

Índice general

Resumen V

Índice general VI

Índice de cuadros IX

Índice de figuras X

Índice de listados XII

Listado de acrónimos XIV

Agradecimientos XV

1. Introducción 1

1.1. Estructura del documento . . . 2

2. Antecedentes 4

2.1. Computación en la nube . . . 4

2.1.1. Definición . . . 5

2.1.2. Arquitectura . . . 5

2.1.3. Modelos de servicio . . . 6

2.1.4. Beneficios y Desventajas . . . 6

2.1.5. Algunas plataformas de Cloud Computing . . . 8

2.2. Sistemas distribuidos . . . 10

2.2.1. Modelo distribuido orientado a objetos . . . 11

2.2.2. El middleware ZeroC ICE . . . 11

2.3. IceGrid . . . 16

2.3.1. Arquitectura y componentes . . . 17

2.3.2. Despliegue de aplicaciones . . . 18

2.3.3. Distribución de aplicaciones . . . 23

3. Objetivos **25**
3.1. Objetivo general 25
3.2. Objetivos específicos 25
3.2.1. Identificación de características de los nodos 25
3.2.2. Descripción de las aplicaciones 26
3.2.3. Monitorización del grid 26
3.2.4. Computación elástica 26

4. Método de trabajo y herramientas **27**
4.1. Metodología de trabajo 27
4.1.1. Prototipado evolutivo 27
4.2. Herramientas 29
4.2.1. Lenguajes de programación 29
4.2.2. Herramientas hardware 30
4.2.3. Software 30

5. Desarrollo del proyecto **32**
5.1. Especificación de requisitos 32
5.2. Pruebas 32
5.3. Proceso de desarrollo 33
5.3.1. Iteración 1: Obtención de información del nodo y categorías 33
5.3.2. Iteración 2: Descripción de aplicaciones 35
5.3.3. Iteración 3: Despliegue de aplicaciones 39
5.3.4. Iteración 4: Asignación dinámica de nodos 54
5.3.5. Iteración 5: Elasticidad 67
5.3.6. Iteración 6: Inclusión de plantillas de servidores 72
5.3.7. Iteración 7: Creación de una interfaz de usuario 80
5.3.8. Iteración 8: Creación de una aplicación de uso real 84

6. Resultados **86**
6.1. Aplicación de la herramienta 86
6.2. Costes y recursos 88
6.3. Repositorio 89

7. Conclusiones y trabajo futuro **90**
7.1. Conclusiones y objetivos alcanzados 90
7.2. Trabajo futuro 91

A. Referencia IceGrid XML **93**

A.1. Adapter Descriptor Element . . . 93
A.2. Allocatable Descriptor Element . . . 94
A.3. Application Descriptor Element . . . 95
A.4. DbEnv Descriptor Element . . . 95
A.5. DbProperty Descriptor Element . . . 96
A.6. Description Descriptor Element . . . 97
A.7. Directory Descriptor Element . . . 97
A.8. Distrib Descriptor Element . . . 97
A.9. IceBox Descriptor Element . . . 98
A.10. IceGrid Descriptor Element . . . 98
A.11. Load-Balancing Descriptor Element . . . 98
A.12. Log Descriptor Element . . . 99
A.13. Node Descriptor Element . . . 99
A.14. Object Descriptor Element . . . 100
A.15. Parameter Descriptor Element . . . 101
A.16. Properties Descriptor Element . . . 101
A.17. Property Descriptor Element . . . 102
A.18. Replica-Group Descriptor Element . . . 103
A.19. Server Descriptor Element . . . 103
A.20. Server-Instance Descriptor Element . . . 104
A.21. Server-Template Descriptor Element . . . 105
A.22. Service Descriptor Element . . . 106
A.23. Service-Instance Descriptor Element . . . 106
A.24. Service-Template Descriptor Element . . . 107
A.25. Variable Descriptor Element . . . 108

Bibliografía **109**

Índice de cuadros

5.1. Opciones para el atributo *allocationRule* 57
5.2. Primer término del atributo compuesto *allocationRule* 57
5.3. Operadores para atributo compuesto *allocationRule* 57

6.1. Evolución de los servidores 88
6.2. Número total de frames renderizados por servidor 88
6.3. Total líneas de código por lenguaje 89

Índice de figuras

2.1. Cloud Computing 4
2.2. Arquitectura de un sistema cloud 6
2.3. Modelos de Servicio 7
2.4. Capas hardware y software 11
2.5. Estructura Cliente-Servidor 12
2.6. Distribución con IcePatch2 16
2.7. Aplicación IceGrid simple 17
2.8. Ventana inicial de IceGrid GUI 20
2.9. Descriptor de aplicación en IceGrid Admin 20
2.10. Aplicación en IceGrid Admin 22
2.11. Proceso de distribución de archivos 24

4.1. Diagrama de flujo del modelo evolutivo 27
4.2. Paradigma de prototipado evolutivo 28

5.1. Diagrama de secuencia para la obtención del fichero IceGrid eXtensible Markup Language (XML) 38
5.2. Diagrama de clases de AdapterDescriptor 41
5.3. Diagrama de clases de ApplicationDescriptor 42
5.4. Diagrama de clases de CommunicatorDescriptor 43
5.5. Diagrama de clases de DbEnvDescriptor 43
5.6. Diagrama de clase de DistributionDescriptor 43
5.7. Diagrama de clase de NodeDescriptor 44
5.8. Diagrama de clase de ObjectDescriptor 45
5.9. Diagrama de clase de PropertyDescriptor 45
5.10. Diagrama de clase de PropertySetDescriptor 45
5.11. Diagrama de clase de ReplicaGroupDescriptor 46
5.12. Diagrama de clase de ServerInstanceDescriptor 47
5.13. Diagrama de clase de ServiceInstanceDescriptor 48

5.14. Diagrama de clase de TemplateDescriptor 48
5.15. Diagrama de secuencia del renderizado. 85

6.1. Estructura de la aplicación de renderizado 86
6.2. Evolución de los servidores durante la ejecución 88

Índice de listados

2.1. Ejemplo de interfaz en SLICE 15
2.2. Ejemplo de aplicación en XML 23
5.1. SLICE de NodeInfoData. Iteración 1 33
5.2. Interfaz Slice que implementa `NodeInfoServer` 34
5.3. Ejemplo de fichero de categorías 35
5.4. Ejemplo de aplicación IceCloud con servidor 36
5.5. SLICE de la interfaz ICRegistry 37
5.6. Ejemplo de especificación de categorías 55
5.7. `NodeInfoData` y `ExtendedNodeInfo` 56
5.8. Slice del descriptor de una aplicación 58
5.9. Slice del descriptor de un nodo definido por categoría 58
5.10. Slice del descriptor de un servidor 59
5.11. Ejemplo de especificación de categorías 68
5.12. Aplicación con `category-server-template` 73
5.13. SLICE de ICRegistry. 81
5.14. SLICE de la aplicación de renderizado. 84
6.1. Estimación de costes del proyecto 89
A.1. Ejemplo de uso del elemento `adapter` 94
A.2. Ejemplo de uso del elemento `application` 95
A.3. Ejemplo de uso del elemento `dbenv` 96
A.4. Ejemplo de uso del elemento `dbproperty` 96
A.5. Ejemplo de uso del elemento `description` 97
A.6. Ejemplo de uso del elemento `distrib` 97
A.7. Ejemplo de uso del elemento `icebox` 98
A.8. Ejemplo de uso del elemento `load-balancing` 99
A.9. Ejemplo de uso del elemento `log` 99
A.10. Ejemplo de uso del elemento `node` 100
A.11. Ejemplo de uso del elemento `object` 101

A.12. Ejemplo de uso del elemento parameter 101
A.13. Ejemplo de uso del elemento properties 102
A.14. Ejemplo de uso del elemento property 103
A.15. Ejemplo de uso del elemento replica-group 103
A.16. Ejemplo de uso del elemento server 105
A.17. Ejemplo de uso del elemento server-instance 105
A.18. Ejemplo de uso del elemento server-template 106
A.19. Ejemplo de uso del elemento service 106
A.20. Ejemplo de uso del elemento service-instance 107
A.21. Ejemplo de uso del elemento service-template 107
A.22. Ejemplo de uso del elemento variable 108

Listado de acrónimos

GNU	GNU is Not Unix
AMI	Amazon Machine Image
API	Application Programming Interface
COCOMO	COnstructive COst MOdel
CPU	Central Processing Unit
EC2	Elastic Cloud Computing
FTP	File Transfer Protocol
GAE	Google App Engine
GCE	Google Compute Engine
GPL	GNU General Public License
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
Ice	Internet Communications Engine
PFC	Proyecto Fin de Carrera
PaaS	Platform as a Service
RAM	Random-Access Memory
S3	Simple Storage Service
SaaS	Software as a Service
Slice	Specification Language for Ice
SSL	Secure Sockets Layer
XML	eXtensible Markup Language
FTP	File Transfer Protocol
UML	Unified Modeling Language
W3C	World Wide Web Consortium

Agradecimientos

Quiero aprovechar estas líneas para dar las gracias a todas aquellas personas que han hecho posible la finalización de estos estudios, con su apoyo, colaboración y paciencia.

En primer lugar, a mis padres, por su esfuerzo y su cariño incondicional. Gracias por vuestro apoyo, vuestros consejos y vuestros ánimos, tanto en los buenos momentos como en los no tan buenos. También me gustaría agradecer al resto de mi familia la ayuda que me han ofrecido a la hora de superar cualquier obstáculo.

A Antonio, por estar siempre ahí, por entenderme y darme fuerzas cuando más lo necesitaba.

Agradezco a David Villa, la oportunidad que me ha brindado con la realización de este proyecto, el tiempo invertido y los conocimientos que me ha permitido adquirir con su dedicación.

Por último, gracias también a todas las personas que he conocido a lo largo de los años de carrera, por los buenos ratos, las noches en vela y las inquietudes compartidas.

Laura Núñez Villa

A mis padres, con admiración y cariño
A Antonio, por su apoyo y comprensión

Introducción

En 1969, Leonard Kleinrock, uno de los creadores de ARPANET, la red precursora de Internet, auguró que en un futuro, con el desarrollo de las redes de computadores, ciertas «utilidades de computación» estarían presentes en los hogares y oficinas del mismo modo que ya lo estaban los servicios eléctricos y telefónicos [Kle69].

Hoy en día, nos estamos acercando a esa visión. Bajo el paradigma emergente de Cloud Computing (o computación «en la nube») se pretende que los consumidores puedan acceder a un conjunto de recursos de computación (servidores, aplicaciones, almacenamiento...) ubicados en cualquier parte del mundo, de una manera cómoda y bajo demanda.

Una infraestructura *cloud* es un conjunto de recursos hardware y software que hace posible un servicio accesible a través de Internet mediante el cual el cliente puede decidir los recursos que necesita, solicitar una *asignación dinámica* y automática (*elástica*) de los mismos, ejecutar aplicaciones escalables sobre ellos y pagar únicamente por el tiempo y cantidad de uso de dichos recursos. Actualmente, empresas punteras en el sector de los servicios informáticos como Amazon o Google ofrecen sus propias plataformas de Cloud Computing.

El concepto de Cloud Computing engloba principalmente tres modelos de servicio:

- **El software como servicio, SAAS**: Permite a los clientes acceder a las aplicaciones que se encuentran ejecutando en una infraestructura cloud.
- **La plataforma como servicio, PAAS**: Ofrece a los clientes la posibilidad de desplegar sus propias aplicaciones en una infraestructura cloud, sin tener control sobre los elementos de la infraestructura (redes, servidores, sistema operativo...).
- **La infraestructura como servicio, IAAS**: Permite a los clientes ejecutar y desplegar sus aplicaciones en una infraestructura sobre la que es posible decidir ciertos aspectos.

El objetivo del presente proyecto es el diseño y la implementación de una plataforma de Cloud Computing del tipo IAAS, usando para ello el middleware de comunicaciones ZeroC ICE. Con su servicio IceGrid, este middleware proporciona la posibilidad de desplegar aplicaciones en un sistema distribuido (grid[1]), especificando explícitamente la estructura del mismo

[1] Un grid se puede definir básicamente como una red de computadores (de relativo bajo coste), posiblemente con diferente hardware y software, con la finalidad de procesar una tarea que requiere gran cantidad de recursos.

y los servidores alojados en cada nodo del grid. IceGrid no es un servicio cloud puesto que la definición del grid es estática y previa al arranque del sistema.

Este proyecto aborda el proceso mediante el cual se subsanan esas limitaciones de IceGrid para crear un servicio Infrastructure as a Service (IAAS) a partir de él, permitiendo describir las aplicaciones con independencia de la arquitectura del grid y así conseguir una *asignación dinámica* dotando al sistema de la **elasticidad** necesaria para que la asignación de recursos crezca o decrezca de forma automática en función de los recursos disponibles y las necesidades de la aplicación.

1.1 Estructura del documento

A continuación se detalla la estructura de este documento, describiendo brevemente el contenido de cada uno de los capítulos que lo componen.

Capítulo 2: Antecedentes

En este capítulo se realiza una introducción al concepto de Cloud Computing y sus diferentes modelos de servicio, presentando también algunas de las plataformas de computación «en la nube» más utilizadas actualmente.

Además, se definen las nociones sobre sistemas distribuidos que son necesarias para la comprensión del resto del documento y se explica en qué consiste el middleware ZeroC ICE, especialmente su servicio IceGrid.

Capítulo 3: Objetivos

El principal objetivo de este proyecto es el diseño e implementación de una plataforma de Cloud Computing que permita desplegar aplicaciones sobre un grid heterogéneo.

En este capítulo se detallarán los objetivos específicos que se pretenden alcanzar a lo largo del desarrollo del proyecto.

Capítulo 4: Método de trabajo y herramientas

La metodología elegida para el desarrollo de este proyecto es el prototipado incremental.

En este capítulo se describe la metodología seguida, además de especificar las herramientas, tanto hardware como software, utilizadas para la elaboración de este proyecto.

Capítulo 5: Desarrollo del proyecto

En este capítulo se encuentra detallado el trabajo realizado en cada una de las iteraciones del proyecto.

Capítulo 6: Resultados

Se describen los resultados obtenidos tras el desarrollo del sistema, así como el esfuerzo realizado, estimaciones del esfuerzo y costes del proyecto, etc.

Capítulo 7: Conclusiones y trabajo futuro

En este capítulo se resumen las conclusiones extraídas a lo largo de la elaboración de este PFC y se aportan varias propuestas de trabajo futuro para enriquecer y completar el trabajo aquí realizado.

Antecedentes

2

En este capítulo, se realizará una presentación de los conceptos que han servido como base teórica para la realización de este proyecto. Para ello, se definirá el concepto de Cloud Computing, haciendo referencia a distintas plataformas de computación elástica actualmente en el mercado. Además, se estudiarán las tecnologías utilizadas en el diseño y desarrollo de IceCloud, que serán citadas en capítulos posteriores y son claves para la comprensión de su funcionamiento.

2.1 Computación en la nube

El concepto emergente de «Cloud Computing» o «computación en la nube» es un paradigma que pretende que los consumidores puedan acceder a un conjunto de recursos (servidores, aplicaciones, almacenamiento...) ubicados en cualquier parte del mundo, de una manera cómoda y bajo demanda. Esos recursos, desde la infraestructura física o virtual hasta el software, son vistos por los consumidores como un servicio.

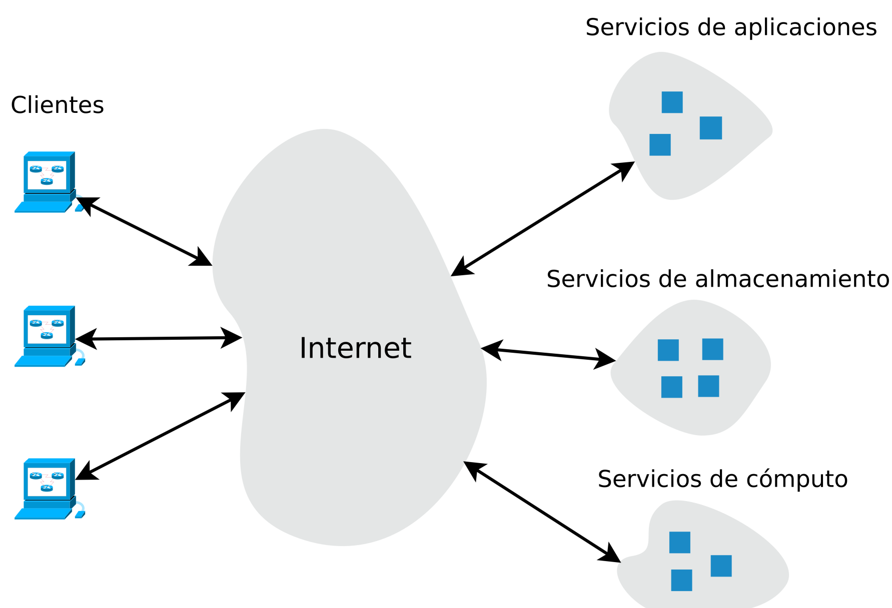

Figura 2.1: Cloud Computing

2.1.1 Definición

No existe un claro consenso a la hora de concretar qué es un Cloud, pero basándonos en diferentes definiciones [BYV+09, FZRL08, COS09], podemos concluir que una «nube» es un tipo de sistema distribuido a gran escala, en el cual se ofrecen bajo demanda y a través de Internet una serie de servicios dinámicamente escalables. Hay cinco características esenciales que forman parte de la definición de Cloud Computing [nis11]:

- **Autoservicio bajo demanda:** El usuario puede decidir los recursos que necesita, como el número de servidores o almacenamiento, automáticamente, sin precisar de interacción humana con los proveedores.
- **Acceso generalizado a través de la red:** Los servicios son accesibles a través de Internet, mediante mecanismos que favorezcan el acceso a diferentes dispositivos.
- **Reserva de recursos:** Los recursos están disponibles para múltiples usuarios, a los que se les asignan diferentes recursos físicos y virtuales dependiendo de sus necesidades. El usuario no no tiene conocimiento de la localización exacta de los recursos utilizados, pero puede especificar sus preferencias a grandes rasgos.
- **Elasticidad:** El usuario ve los recursos como ilimitados, y puede demandar que se le asigne una cantidad mayor o menor de los mismos en cualquier momento, en algunos casos automáticamente. El sistema debe adaptarse rápidamente a esos requerimientos de crecimiento y decrecimiento para obtener los recursos necesarios, y únicamente los necesarios, en función de las condiciones de funcionamiento en un entorno cambiante (por ejemplo, cantidad de usuarios conectados a una aplicación web).
- **Medidas de consumo:** El tiempo y cantidad de recursos usados deben poder ser monitorizados y optimizados, con el fin de ofrecer transparencia en el caso de modelos de negocio que tarifican en función del uso de los mismos.

2.1.2 Arquitectura

La arquitectura de un sistema cloud se puede dividir en cuatro capas (ver figura 2.2):

- Capa de recursos físicos: contiene los recursos a hardware: CPU, almacenamiento, red...
- Capa de virtualización: contiene los recursos encapsulados, que se muestran al usuario final y a las capas superiores de forma integrada.
- Capa de plataforma: añade una serie de herramientas, middleware y servicios para proveer una plataforma de desarrollo y/o despliegue de aplicaciones.
- Capa de aplicación: contiene aplicaciones que se pueden ejecutar en el sistema cloud.

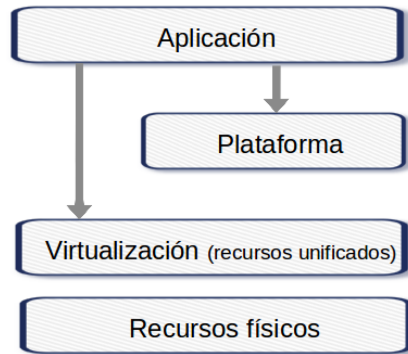

Figura 2.2: Arquitectura de un sistema cloud

2.1.3 Modelos de servicio

Mediante Cloud Computing se ofrecen básicamente tres modelos de servicio (ver figura 2.3):

El software como servicio, SAAS (Software as a Service)
Permite a los usuarios ejecutar aplicaciones que corren en una infraestructura Cloud. Los usuarios acceden a esas aplicaciones mediante una interfaz, como un navegador web. Los usuarios no controlan ni administran ningún aspecto de la infraestructura, exceptuando algunos ajustes limitados de la aplicación. Un ejemplo de SAAS son los servidores de correo web.

La plataforma como servicio, PAAS (Platform as a Service)
Este modelo permite a los usuarios desplegar aplicaciones adquiridas o creadas por ellos mismos usando una serie de lenguajes y herramientas soportadas por el proveedor. El usuario tiene el control sobre las aplicaciones desplegadas, pero no sobre el resto de la infraestructura (red, servidores, almacenamiento...)

La infraestructura como servicio, IAAS (Infrastructure as a Service)
Este tipo de modelo es el de menor nivel de abstracción, ya que el usuario tiene control sobre recursos de procesamiento, bases de datos y otros aspectos de la infraestructura Cloud, permitiéndole desplegar y ejecutar cualquier tipo de software. Esto no significa que tenga acceso directamente a los recursos físicos (ver figura 2.2), sino sobre los recursos virtualizados ofrecidos por el Cloud. Este es el tipo de servicio que se pretende ofrecer en este proyecto.

2.1.4 Beneficios y Desventajas

Las ventajas que ofrecen los sistemas de Cloud Computing son especialmente útiles en tres casos particulares:

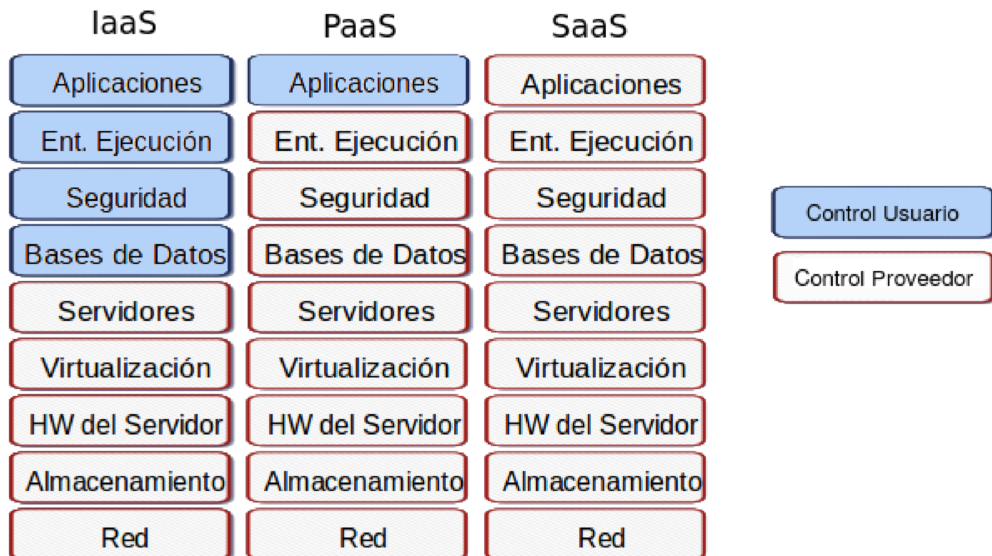

Figura 2.3: Modelos de Servicio

- Cuando la demanda de un servicio varía con el tiempo. Por ejemplo, en sistemas que cuentan con un pico de carga unos pocos días al mes, cuando el resto del tiempo estarían infrautilizados, Cloud Computing permite *alquilar* por ese tiempo los recursos necesarios, en lugar de adquirirlos definitivamente, lo que supone un ahorro de coste económico.
- Cuando de antemano no se conoce la demanda del sistema. Por ejemplo, en webs en las que el número de visitantes puede variar drásticamente.
- En casos en los que es necesario el procesamiento de grandes lotes de datos que es posible procesar paralelamente. Se podrían usar 1000 máquinas de un sistema cloud durante una hora, en lugar de una sola máquina durante 1000 horas, asumiendo que el paralelismo de la aplicación fuera absoluto.

Aunque hay corrientes que consideran que Cloud Computing suplantará en el futuro la computación local por parte de los clientes, existen ciertas desventajas que indican que seguirá presente una coexistencia de ambos:

- Por razones de seguridad. Cloud Computing pone datos que pueden ser sensibles o críticos, a disposición de terceros.
- Los usuarios dependen de una conexión a la red para obtener sus resultados.
- Los avances tecnológicos, especialmente los procesadores multi-core, pueden posibilitar en el futuro grandes recursos de cómputo en cualquier computador, del orden de cientos de hilos a bajo coste, lo que eliminaría (o reduciría) la necesidad de solicitar recursos externos.

2.1.5 Algunas plataformas de Cloud Computing

Cloud Computing es una de las tendencias tecnológicas actuales. Organizaciones de todo el mundo, tanto académicas como industriales están investigando y desarrollando tecnologías e infraestructuras enfocadas a Cloud Computing. A continuación se muestran algunos ejemplos de plataformas desarrolladas.

Amazon Elastic Compute Cloud - EC2

Amazon EC2 [EC2] es un ejemplo significativo de infraestructura como servicio. Desarrollado por Amazon, permite a los usuarios ejecutar aplicaciones basadas en GNU/Linux y Windows Server.

Los usuarios pueden crear instancias de servidores (Amazon Machine Image (AMI)s) y subirlas a un servicio de almacenamiento (Simple Storage Service (S3)) [AS3] y, después, ejecutarlas, detenerlas o monitorizarlas.

Es posible elegir entre diferentes tipos de instancias, clasificadas según la combinación de algunas características hardware (Central Processing Unit (CPU), almacenamiento y memoria). Es posible indicar las preferencias de los usuarios para que las instancias crezcan o se eliminen dependiendo del uso de CPU (*auto-scaling*). Además, Amazon ofrece un conjunto de aplicaciones comerciales gratuitas que pueden ser ejecutadas en esas instancias (gestores de contenido, sistemas gestión de bases de datos, frameworks web, etc).

El servicio EC2 cobra al usuario según el tiempo (en horas) que se ejecuta una instancia y el tipo de instancia, y Amazon S3 según las transferencias de datos realizadas.

Google Cloud Platform

Google Cloud Platform [GCP] engloba una serie de productos orientados a Cloud Computing, que incluyen tanto SAAS, como PAAS e IAAS.

Google Services

En lo que se refiere a SAAS, Google ofrece una serie de aplicaciones tales como traductores, herramientas de predicción y análisis de tráfico web, o servidores DNS.

Google App Engine (GAE)

Google App Engine es un ejemplo de plataforma como servicio. Permite al usuario ejecutar aplicaciones web escritas en Python, Java, PHP y Go. Además, también soporta APIs para almacenamiento de datos, Google Accounts, manipulación de imágenes o servicio de correo. Las aplicaciones se pueden administrar mediante una consola web. Su uso es gratuito hasta 1 GB de almacenamiento y alrededor de los 5 millones de visitas por mes.

Google Compute Engine (GCE)

Google Compute Engine es el servicio IAAS de Google Cloud Platform. En él, los

clientes pueden especificar imágenes de máquinas virtuales de diferentes categorías dependiendo del hardware, elegir entre el sistema operativo entre un conjunto de sistemas GNU/Linux o Windows Server y asignar esas imágenes a instancias. Además, el usuario puede elegir entre varias localizaciones físicas de esas imágenes.

Las instancias pueden ser replicadas en caso de alto tráfico de red, para balancear la carga de servidores de contenido web o para optimizar el acceso HTTP basándose en la localización física.

Los usuarios pagan según el tiempo de uso de las máquinas (una primera fracción de 10 minutos, y el resto de 1 minuto) y la localización de las mismas, abonando aparte servicios extra como ciertos sistemas operativos o persistencia de los datos.

Microsoft Azure

Con Microsoft Azure [AZU], la empresa Microsoft ofrece tanto servicios de plataforma como de infraestructura.

Como plataforma, mediante *Azure RemoteApp*, permite ejecutar aplicaciones desarrolladas para Windows Server en diversos lenguajes (.NET, Node.js, Java, PHP y Python).
Como infraestructura, permite al usuario configurar máquinas virtuales con diferentes sistemas operativos, entre los que se incluyen Ubuntu, SUSE y Windows Server.
Tanto las aplicaciones ejecutadas en las máquinas virtuales, como las ejecutadas mediante *Azure RemoteApp* pueden ser escalables automáticamente (mediante programación o en función del uso de CPU de las máquinas virtuales).

Además, ofrece distintos entornos de desarrollo, servicios de gestión de base de datos, servicios de almacenamiento, herramientas para procesamiento multimedia y otro tipo de software para integrar en las aplicaciones y facilitar el desarrollo de las mismas.

CloudStack

Apache CloudStack es un software de código abierto que permite administrar una gran cantidad de máquinas virtuales que forman parte de una plataforma de Cloud Computing del tipo de infraestructura como servicio.

Soporta hipervisores como BareMeta, Hyper-V, KVM o Xenserver, entre otros.

Los grids CloudStack deben contener un servidor de administración (*CloudStack Management Server*), encargado de gestionar la comunicación entre las máquinas virtuales, y el almacenamiento. Este servidor ofrece una interfaz web para el control de las aplicaciones, y es el encargado de escalar los servidores ejecutados en el resto de máquinas del grid.

Además de las características propias de una plataforma de este tipo, ofrece un API compatible con Elastic Cloud Computing (EC2) [EC2] y S3 [AS3], lo que permite a los usuarios desplegar un modelo híbrido.

OpenStack

OpenStack [OST] se trata de un proyecto de software libre, al cual se han unido más de 150 empresas tales como Intel, AMD, Canonical o Cisco. Ofrece un servicio de tipo IAAS. Está compuesto por diferentes módulos, encargados del cómputo de la aplicación (Nova), el almacenamiento (Swift) o el sistema de direcciones de red (Quantum), entre otros.

Su componente principal, *Nova*, gestiona diferentes instancias de imágenes de máquinas virtuales (o incluso físicas, mediante el proyecto *Ironic*) y soporta distintos hipervisores entre los que se encuentran KVM o Xen. Este componente es el encargado de redimensionar las instancias para ofrecer escalabilidad al sistema. Mediante el módulo `nova-api` permite la comunicación con los usuarios o el resto de componentes de OpenStack, su módulo `nova-scheduler` planifica la ejecución de las instancias en los diferentes nodos del cloud y su módulo `nova-compute` ejecuta las instancias sobre un hipervisor.

2.2 Sistemas distribuidos

Un sistema distribuido es aquel en el que el procesamiento de la información se realiza sobre varios computadores que se comunican y coordinan mediante paso de mensajes [CJKB12] en lugar de hacerlo sobre una sola máquina.

Los sistemas distribuidos implican las siguientes consecuencias:

- Concurrencia. En un sistema distribuido, se pueden ejecutar varios procesos sobre diferentes computadores de la red. Una ventaja de los sistemas distribuidos es que esos procesos pueden compartir recursos asociados a la red, pero añade otras dificultades. Es posible que estos procesos necesiten comunicarse y coordinarse para controlar el acceso a dichos recursos, y esto es un punto importante a tener en cuenta en el desarrollo de aplicaciones distribuidas. Por otra parte, la concurrencia también facilita la escalabilidad del sistema, añadiendo más recursos según la demanda.
- Ausencia de reloj global. Cada máquina física dispone de su propio reloj, pero los relojes de diferentes máquinas no están sincronizados perfectamente, por lo que puede que se necesiten algoritmos adicionales de sincronización entre procesos basados en el paso de mensajes.
- Fallo parcial. Un sistema distribuido puede fallar parcialmente. Un recurso concreto puede dejar de funcionar, sin que esto afecte al funcionamiento del resto de servicios y, en consecuencia, un fallo parcial no tiene por qué significar el fallo completo del sistema, que podrá seguir activo con un funcionamiento degradado. El sistema sólo falla irremediablemente si falla la red que comunica los distintos recursos.

2.2.1 Modelo distribuido orientado a objetos

El modelo distribuido orientado a objetos se basa en una arquitectura en la cual los principales componentes del sistema son objetos, definidos por una interfaz mediante la cual especifican los servicios que proporcionan. Mediante esa interfaz, otros objetos realizan llamadas a esos servicios sin hacer distinción entre servidores (proveedores del servicio) y clientes (receptores del servicio) [Som05] y sin conocer el funcionamiento interno de los objetos.

Middleware

Un middleware es un sistema que ofrece herramientas para la ejecución de aplicaciones distribuidas. Añade una capa entre el sistema operativo y la aplicación (ver figura 2.4), de tal modo que enmascara ante las aplicaciones de los detalles de red, sistema operativo o la arquitectura del computador y hace posible que las aplicaciones se ejecuten en plataformas heterogéneas e incluso que los objetos de una misma aplicación estén implementados en distintos lenguajes de programación.

Las aplicaciones acceden a las funcionalidades del middleware a través de un API adaptado a los distintos lenguajes de programación permitidos. Los desarrolladores lo ven como una librería y un conjunto de herramientas para la gestión de las aplicaciones [PRP05]

Figura 2.4: Capas hardware y software

2.2.2 El middleware ZeroC ICE

ICE (Internet Communications Engine) es un middleware de comunicaciones orientado a objetos desarrollado por la empresa ZeroC bajo licencia GPL.

Ofrece múltiples ventajas para la implementación de aplicaciones sobre sistemas distribuidos heterogéneos, ya que soporta múltiples lenguajes (C++, Java, Python, Ruby, PHP, ActionScript, etc.) y está disponible para varios sistemas operativos. Ello conlleva la po-

sibilidad de utilizarlo en muchos tipos de plataformas, incluyendo dispositivos móviles, o incluirlo en navegadores web.

ICE ofrece un conjunto de APIs que son prácticamente idénticas para los distintos lenguajes, lo que agiliza el desarrollo de aplicaciones multi-lenguaje.

También incluye una serie de servicios útiles para la propagación de eventos, distribución del software, persistencia, despliegue y monitorización de las aplicaciones, etc.

A continuación se abordan un serie de conceptos que es necesario conocer para comprender el funcionamiento de ICE.

Clientes y servidores

La arquitectura de ICE se basa en una arquitectura cliente-servidor.

Los clientes son las entidades que solicitan servicios a objetos remotos, realizando invocaciones a sus métodos a través de un proxy.

Los servidores son las entidades que proporcionan servicios en respuesta a las solicitudes de los clientes.

Un mismo programa puede contener objetos (rol servidor) y hacer invocaciones a otros objetos remotos (rol cliente).

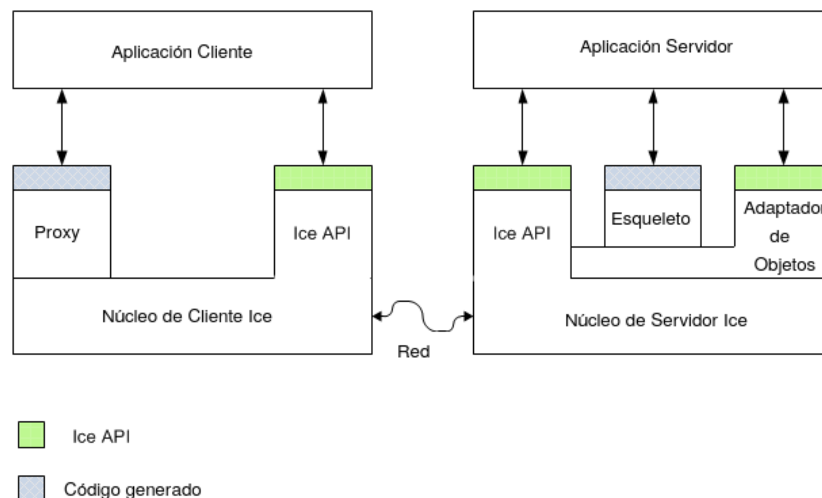

Figura 2.5: Estructura Cliente-Servidor

En la figura 2.5 vemos la estructura lógica interna de clientes y servidores.

- El núcleo ICE ofrece soporte para la comunicación remota. Su parte genérica se accede haciendo uso del API y es la misma para clientes y servidores (aunque los servidores hacen un uso extensivo de ella).
- El proxy ofrece una interfaz para que los clientes puedan invocar objetos y se encarga de serializar los datos para enviarlos por la red.

- El esqueleto es el encargado de recibir las invocaciones. Es la contrapartida funcional al proxy en el lado del cliente.
- El adaptador de objetos se encarga de crear los proxies para los objetos, y traducir las invocaciones de los clientes.

A continuación se explican con más detalle algunos de los conceptos nombrados anteriormente.

Objetos

En ICE, un objeto es una entidad con las siguientes características:

- Tiene un tipo y una identidad única, y está asociado a un adaptador de objetos, que contiene información de direccionamiento.
- Proporciona, al menos, una interfaz que especifica las operaciones que soporta y que son invocadas por los clientes. Esas operaciones tienen cero o más parámetros y un valor de retorno.

Proxies

Un proxy representa a un objeto (posiblemente remoto) en el espacio de direcciones de un cliente. Cuando los clientes quieren invocar una operación de un objeto, basta con que invoquen dicha operación en su proxy.

Los proxies encapsulan información de direccionamiento, sobre la identidad del objeto y sobre la interfaz del objeto.

ICE dispone de varios tipos de proxies aptos para diferentes situaciones:

- **Proxies directos (*direct proxies*)**
  En un proxy directo se especifica completamente la información del objeto (identidad y dirección asociada al servidor), incluyendo explícitamente el protocolo de transporte, la interfaz y el puerto.

  Su representación textual sería como la siguiente:

  ```
  Identity -t : tcp -h 127.0.0.1 -p 10000
  ```

  - **Identity** es la identidad del objeto
  - La opción **-t** es el método de acceso al objeto (*two-way*), mediante el que se indica que el cliente manda invocaciones y se espera una repuesta. Si el cliente no debe esperar respuesta, el método de acceso sería **-o** (*one-way*).
  - **tcp** es el identificador del protocolo de transporte. También puede ser **udp**.
  - Las opciones **-h 127.0.0.1 -p 10000** representan, respectivamente, la interfaz y el puerto donde escucha el servidor.

    - La unión de las opciones de los dos puntos anteriores también se conoce como *endpoint*.

- **Proxies indirectos (*indirect proxies*)** Los proxies indirectos no contienen información de direccionamiento.

  Pueden ser de dos tipos:

    - Proxies que representan objetos bien conocidos (*well-known objects*), accedidos directamente por su identidad, como en la cadena `ObjectIdentity` .
    - Proxies registrados en adaptadores bien conocidos. Como por ejemplo `ObjectIdentity@Adapter` . En este caso no importa que el objeto no sea un objeto bien conocido.

- **Proxies fijos (*fixed proxies*)**
  Están asociados a una conexión particular, y son destruidos al terminar dicha conexión. Los proxies fijos no pueden ser usados como parámetros al invocar operaciones y pueden ser usados tanto por parte del servidor como del cliente en conexiones bidireccionales.

Sirvientes (*servants*)

Un sirviente es la entidad en el lado del servidor encargada de proporcionar el comportamiento a las operaciones que pueden ser invocadas por los clientes. Se trata de una instancia de una clase registrada en el núcleo del servidor como la entidad que respalda a uno o más objetos. Los métodos de esa clase se corresponden con la interfaz definida para ese objeto.

Un sirviente puede respaldar a un solo objeto, en cuyo caso la identidad del objeto está implícita en el sirviente. También puede respaldar a varios objetos, y en ese caso, mantiene la identidad del objeto con cada petición.

Un único objeto puede tener múltiples sirvientes. Cuando un cliente invoque una operación sobre él, el núcleo de ejecución enviará la solicitud a uno de sus sirvientes. Si la invocación falla, será enviada a otro sirviente, y así sucesivamente. Se permite así crear sistemas redundantes, en los que la invocación sólo fallará si todos los sirvientes fallan.

Slice

Specification Language for Ice (SLICE) es el lenguaje proporcionado por ICE para definir los las interfaces de los objetos. Esta descripción es independiente del lenguaje en el que se programen dichas interfaces, por lo tanto permite desacoplar las interfaces de su implementación. ICE proporciona los compiladores necesarios para traducir las definiciones Slice a tipos de un lenguaje específico y APIs que el desarrollador puede usar para interaccionar con ICE. Esa traducción es conocida como *mapping*.

Los archivos que contienen las definiciones SLICE deben tener la extensión *.ice*.

Un ejemplo de interfaz definida en SLICE (Specification Language for Ice) se puede ver en el listado 2.1.

```
module Example {
  interface Hello {
    void puts(string message);
  };
};
```

Listado 2.1: Ejemplo de interfaz en SLICE

Servicios

IceBox

IceBox es un servidor que permite configurar componentes de las aplicaciones como servicios que son cargados y gestionados por él. Puede ser administrado remotamente.

Los distintos servicios deben implementar una interfaz de servicio IceBox. Se configuran como propiedades, y pueden ser dispuestos para optimizar la comunicación entre ellos.

IceGrid

IceGrid es el servicio de localización y despliegue de aplicaciones.

Como su uso es esencial en este proyecto, se explica más detalladamente en la siguiente sección (véase § 2.3)

IceStorm

IceStorm es el servicio encargado de la propagación de eventos.

Con una arquitectura de publicador-suscriptor, actúa de intermediario entre publicadores y suscriptores. Los publicadores envían datos a un canal de eventos mediante una invocación, y IceStorm se encarga de reenviarla a todos los suscriptores de ese canal.

IcePatch2

IcePatch2 es el servicio de distribución de archivos de ICE.

Se configura como un servidor al que se le indica el directorio donde se encuentra el código de la aplicación y permite enviar eficientemente actualizaciones de software a los distintos nodos, comprobando qué archivos han cambiado y remitiéndolos a aquellos clientes que lo necesiten, garantizando que la copia se hace de forma segura (ver figura 2.6).

Además IcePatch2 brinda una serie de herramientas para la compresión de archivos y el cálculo de checksums, y su tasa de transferencia es similar a la de File Transfer Protocol (FTP).

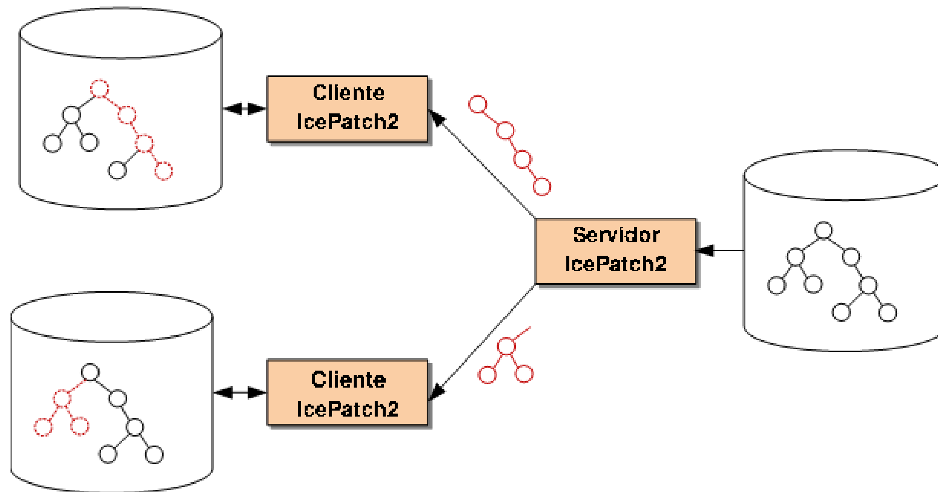

Figura 2.6: Distribución con IcePatch2

Glacier2

Glacier2 es un servicio de firewall que permite que servidores y clientes se comuniquen de forma segura. Ofrece herramientas de autenticación y filtros para controlar el acceso a los recursos, y permite enviar los datos cifrados mediante SSL.

Freeze

Freeze es un servicio que gestiona la persistencia de objetos en ICE. Mediante Freeze Evictor permite guardar fácilmente el estado de los objetos en una base de datos Berkeley[1], actualizarlos cuando se necesite y recuperarlos cuando se intente acceder a ellos.

Mediante Freeze Map se puede guardar otro tipo de datos que no sean objetos ICE.

Aunque Freeze abstrae a las aplicaciones del uso directo de la base de datos, nada impide que ésta sea accedida directamente si se requiere.

2.3 IceGrid

IceGrid es el servicio de localización y activación de aplicaciones de ICE. Es esencial en el desarrollo de este proyecto, por lo que se aborda con mayor profundidad que el resto de servicios, prestando especial detalle en lo referente al despliegue de las aplicaciones.

IceGrid establece una serie de mecanismos para configurar el funcionamiento de un grid, permitiendo la comunicación entre sus diferentes componentes.

A grandes rasgos, IceGrid proporciona:

- Servicio de localización: permite el uso de proxies indirectos (ver apartado 2.2.2).

[1] Berkeley DB es una biblioteca de manejo de base de datos con licencia libre, de alto rendimiento. Soporta múltiples datos para una misma clave y no soporta SQL.

- Activación de servidores bajo demanda: puede ocuparse de activar los servidores cuando se trate de acceder a un objeto alojado en él, de manera totalmente transparente para el cliente.
- Distribución de aplicaciones: proporciona una forma cómoda de distribuir el software a los diferentes nodos del grid, usando un servidor IcePatch2.
- Replicación y balanceo de carga: permite que haya varios servidores asociados a un objeto. Es posible configurarlo para que la decisión de qué servidor responde a las peticiones dependa de la carga de los nodos.
- Tolerancia a fallos: cuando un proxy tiene asociadas varios endpoints, y la invocación usando uno de ellos falla, IceGrid intenta invocar a los demás automáticamente.
- Administración: facilita herramientas de administración, para arrancar, parar, monitorizar y reconfigurar los servidores del grid.
- Despliegue: permite describir los servidores que serán desplegados en cada nodo mediante archivos XML.

2.3.1 Arquitectura y componentes

Un dominio IceGrid consta de un *IceGrid Registry* y cualquier número de nodos (ver figura 2.7).

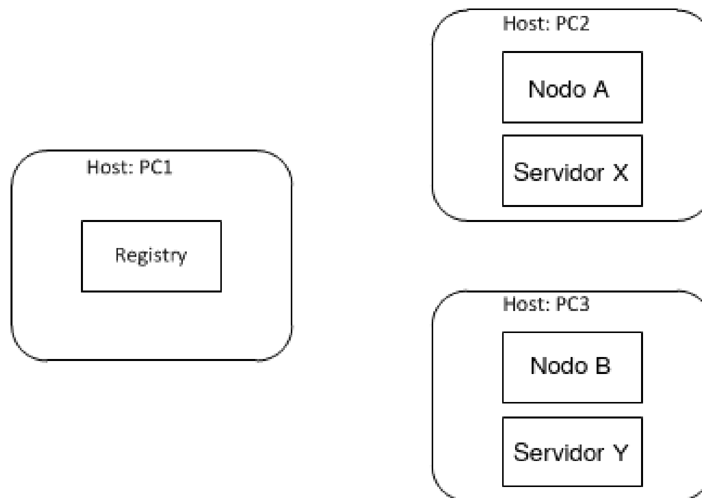

Figura 2.7: Aplicación IceGrid simple

- Cuando nos referimos a un *nodo* lo hacemos a una instancia de *IceGrid Node*. Puede haber varias instancias de *IceGrid Node* en un mismo computador. Un nodo no tiene por qué corresponder unívocamente con un computador.
- Un *servidor* identifica a un programa que se ejecuta en un nodo. Ese programa no tiene por qué implementar el rol de servidor, puede ser un cliente, o un programa que no necesite recibir ni realizar invocaciones remotas en absoluto.

- *IceGrid Registry* es un repositorio centralizado que contiene información sobre las aplicaciones desplegadas, los nodos del grid y los objetos *bien conocidos*, entre otros.

  La responsabilidad principal del Registry es la traducción de los proxies indirectos. Cuando un cliente intenta por primera vez usar un proxy indirecto, el entorno de ejecución de ICE se pone en contacto con el Registry, que convierte la información simbólica del proxy en endpoints.

  En principio puede parecer que el Registry es una simple tabla, pero entre sus tareas también se encuentra el solicitar el arranque de servidores automáticamente o seleccionar los nodos adecuados según su carga.

  En un grid sólo puede puede haber un Registry en ejecución, y es un componente clave en el sistema. Si el Registry o el nodo en el que se encuentra fallan, los clientes que tengan una conexión con un servidor podrán continuar, pero no será posible que establezcan nuevas conexiones si intentan usar proxies indirectos.

2.3.2 Despliegue de aplicaciones

Desplegar una aplicación IceGrid es describir la estructura de esa aplicación al Registry.

Dicha descripción puede incluir la siguiente información:

- **Grupos de réplica:** los grupos de réplica se usan para unificar bajo un mismo identificador único una colección de adaptadores de objetos.
- **Nodos:** Se debe especificar en la aplicación los nodos concretos de los que consta. En la descripción de los nodos se debe precisar el servidor o servidores que se ejecutarán en ese nodo.
- **Servidores:** La descripción de los servidores debe incluir su nombre (que debe ser único en todo el grid) y la ruta a su ejecutable. También debe especificar los adaptadores de objetos que crea.
- **Adaptadores de objetos:** Deben incluir información sobre sus endpoints y sobre los objetos bien conocidos registrados en ellos. Si un adaptador de objetos pertenece a un grupo de réplica, también debe indicar el identificador de ese grupo.
- **Objetos:** Los objetos bien conocidos se caracterizan únicamente por su identidad. Hay que especificarla para que el Registry mantenga una lista global de esos objetos.

IceGrid denomina la descripción de una aplicación y sus componentes con el término *descriptor*. Un *descriptor* de aplicación contiene un número de sub-descriptores (y sub-sub-descriptores) anidados. Los descriptores se corresponden con tipos Slice definidos en las interfaces del Registry.

Desplegar la aplicación requiere crear su descriptor en el Registry.

Para llevar a cabo la carga de un descriptor en el Registry se puede proceder de varias formas:

- Escribir una representación del descriptor en un archivo XML, y utilizar la herramienta administrativa de IceGrid para que lea ese archivo, genere el descriptor a partir de él y lo añada al Registry.
- Usar la interfaz gráfica para crear y cargar los descriptores interactivamente.
- Crear el descriptor desde un programa, y cargarlo mediante la interfaz administrativa del API de IceGrid.

Para desplegar una aplicación, es necesario que el Registry se esté ejecutando, pero no es preciso que los nodos que forman parte de la descripción de la aplicación también lo estén. Los nodos que se activen después de que la aplicación sea desplegada recibirán automáticamente la información necesaria. Después de desplegar una aplicación, se puede actualizar cuando se desee.

Creación de la aplicación desde IceGrid Admin

IceGrid proporciona una herramienta interactiva para la creación y gestión de aplicaciones llamada IceGrid Admin GUI (ver figura 2.8).

Mediante esta interfaz, podemos diseñar la aplicación y cargarla en el Registry.

La aplicación se puede crear de forma muy intuitiva. IceGrid Admin establece una equivalencia entre los descriptores que forman parte de la aplicación los representa en forma de nodos de un árbol cuya raíz es el descriptor de aplicación (ver figura 2.9).

En el siguiente apartado, se explica la estructura anidada de las aplicaciones, aplicándolo a su representación en XML. Cada uno de los elementos se corresponde con un nodo del árbol de IceGrid Admin.

Representación del descriptor en XML

Como hemos dicho anteriormente, la descripción de una aplicación se compone de descriptores anidados, y éstos se pueden representar como elementos XML. A continuación, mostramos un resumen de estos descriptores. En el anexo A se puede ver la descripción completa.

El elemento raíz del árbol XML es un elemento de tipo `icegrid`. Tiene un único hijo que es del tipo *descriptor de aplicación*. El descriptor de aplicación define una aplicación y se indica con la etiqueta `application`. Anidados en él podemos encontrar los siguientes elementos:

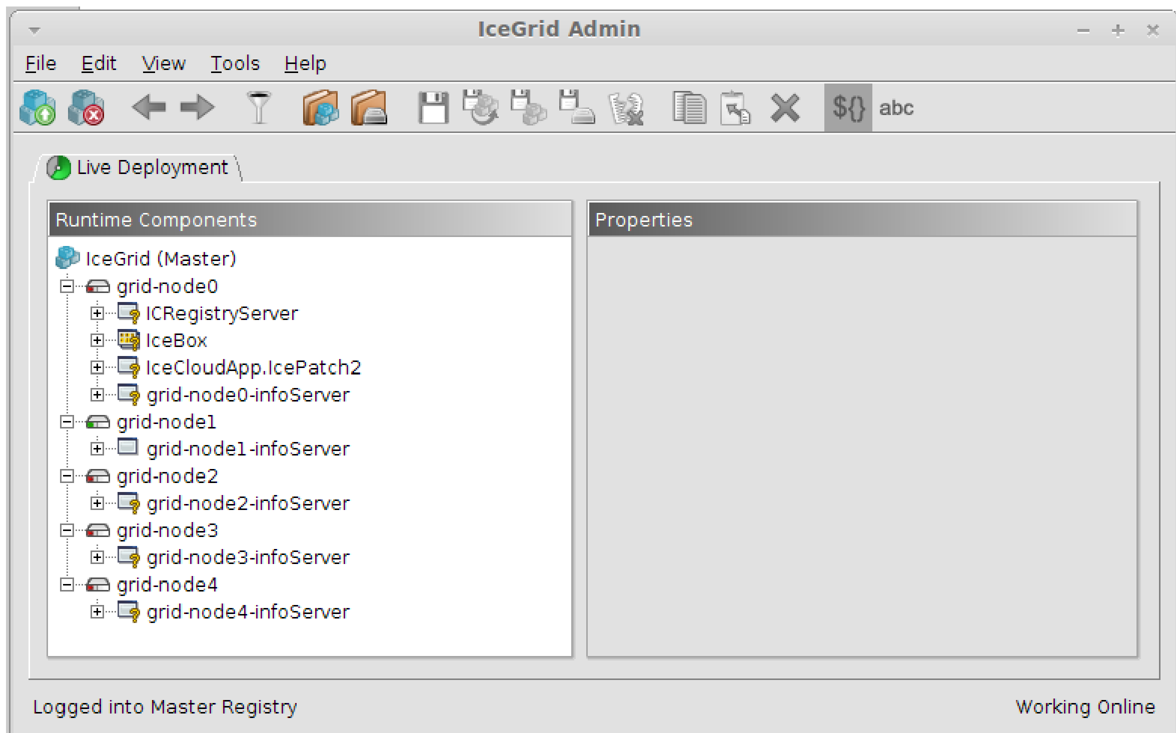

Figura 2.8: Ventana inicial de IceGrid GUI

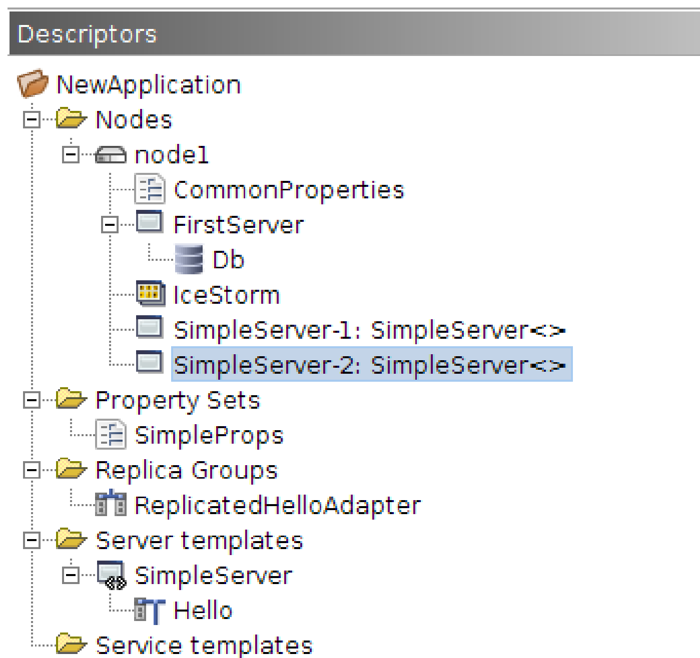

Figura 2.9: Descriptor de aplicación en IceGrid Admin

- **Descriptor de nodo:** Define un nodo IceGrid. Se deben especificar como hijos del nodo los servidores alojados en él. Se indica con la etiqueta `node`. Sus hijos pueden ser:

- **Descriptor de servidor:** Se indica con la etiqueta `server` y define a un servidor alojado en un nodo. Sus hijos pueden ser:
  - **Descriptor de adaptador:** Define un adaptador de objetos. Se indica con la etiqueta `adapter`. Puede tener hijos de tipo:
    - **Descriptor de objeto:** Se indica con la etiqueta `object` y describe un objeto bien conocido.
  - **Descriptor de entorno de base de datos:** Se indica con la etiqueta `dbenv` y su función es establecer la configuración para usar el servicio Freeze. Puede tener como hijos:
    - **Descriptor de propiedad de base de datos:** Se indica con la etiqueta `dbproperty` y define una propiedad de configuración para la base de datos.
  - **Descriptor de log:** Especifica el nombre de un fichero de log y se representa con la etiqueta `log`.
  - **Descriptor de distribución:** Se especifica con la etiqueta `distrib` y especifica los archivos que el servidor debe descargarse del servidor IcePatch2 y su proxy. Puede tener hijos de tipo:
    - **Descriptor de directorio:** Se presenta con la etiqueta `directory` e indica uno de los directorios incluidos en la distribución.
- **Descriptor de instancias:** Se especifica con la etiqueta `server-instance`. Y representa a un servidor creado a partir de una plantilla.
- **Descriptor IceBox:** Se especifica con la etiqueta `icebox` y representa a un servidor IceBox. Puede tener un hijos de tipo:
  - **Descriptor de servicio:** Describe un servicio IceBox. Puede tener hijos de tipo **descriptor de adaptador** `adapter`, que a su vez puede tener los mismos hijos que en el descriptor de servidor. Se define con la etiqueta `service`
  - **Descriptor de instancia de servicio:** Crea una instancia de una plantilla de servicio en el servidor IceBox. Se define con la etiqueta `service-instance`

- **Descriptor de grupo de réplica:** Se especifica con la etiqueta `replica-group` y sirve para agrupar diferentes adaptadores de objeto bajo un mismo adaptador virtual. Puede tener hijos de tipo:
  - **Descriptor de balanceo de carga:** Determina la política de balanceo de carga usada por el grupo de réplica y se especifica con la etiqueta `load-balancing`.
  - **Descriptor de objeto:** Con el que indica los objetos bien conocidos registrados en el grupo. Especificado con la etiqueta `object`.
- **Descriptor de plantilla de servidores:** Se especifica con la etiqueta `server-template` y sirver para definir una plantilla para los servidores, simplificando la tarea de desplegar varias instancias de un servidor con la misma definición. Puede tener hijos de tipo:

- **Descriptor de parámetros:** Determina un parámetro y se especifica con la etiqueta `parameter`.

- **Descriptor de plantilla de servicios:** Se especifica con la etiqueta `service-template` y sirver para definir una plantilla para los servicios, simplificando la tarea de desplegar varias instancias de un servicio con la misma definición.
- **Descriptor de variables:** Se especifica con la etiqueta `variable` y representa a una variable.

Además de los descriptores mencionados, podemos encontrar **descriptores de propiedades**, que determinan un conjunto de propiedades con la etiqueta `properties`, en los descriptores de aplicación, nodo, servidor, icebox, servicio, instancia de servicio e instancia de servidor.

También podemos encontrar **descriptores de descripción**, identificadoes con la etiqueta `description`, que aportan una descripción del elemento en los descriptores de aplicación, grupos de réplica, nodo, servidor, servicio, IceBox, adaptador y entorno de base de datos.

En los descriptores de IceBox, servidor y servicio, podemos encontrar **descriptores de propiedad** con la etiqueta `property`. Los nodos IceGrid generan un fichero de configuración para cada uno de sus servidores y servicios. Utilizando estos descriptores se pueden definir propiedades diferentes a las del archivo.

En el listado 2.2 podemos ver la descripción de una aplicación en XML (adaptada del manual de ICE [Zerb]), y en la figura 2.10 su equivalente representado en un árbol de IceGrid Admin.

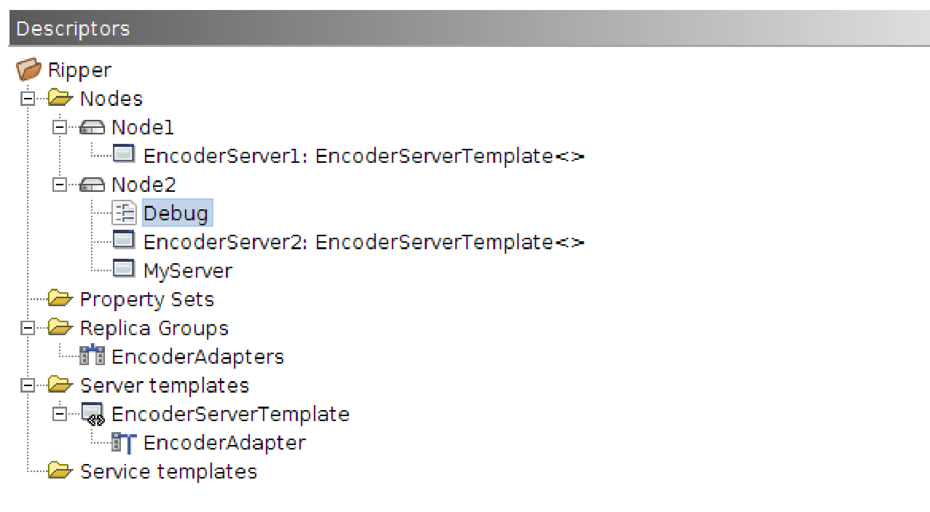

Figura 2.10: Aplicación en IceGrid Admin

```xml
<icegrid>
  <application name="Ripper">
    <replica-group id="EncoderAdapters">
      <load-balancing type="adaptive"/>
      <object identity="EncoderFactory"
      type="::Ripper::MP3EncoderFactory"/>
    </replica-group>
    <server-template id="EncoderServerTemplate">
      <parameter name="index"/>
      <parameter name="exepath" default="/opt/ripper/bin/server"/>
      <server id="EncoderServer${index}" exe="${exepath}"
        activation="on-demand">
        <adapter name="EncoderAdapter"
          replica-group="EncoderAdapters"
          endpoints="tcp"/>
      </server>
    </server-template>
    <node name="Node1">
      <server-instance template="EncoderServerTemplate"
      index="1"/>
    </node>
    <node name="Node2">
      <server-instance template="EncoderServerTemplate"
      index="2"/>
      <server id="MyServer" exe="./server">
      <properties>
        <properties refid="Debug"/>
          <property name="AppProperty" value="1"/>
        </properties>
      </server>
    </node>
  </application>
</icegrid>
```

Listado 2.2: Ejemplo de aplicación en XML

2.3.3 Distribución de aplicaciones

Podemos ampliar la definición de «despliegue» de una aplicación, detallando algunas tareas que conlleva el proceso:

- Crear los ficheros de configuración de IceGrid y preparar los directorios de datos en cada computador.
- Instalar los binaros de IceGrid y las librerías necesarias en cada nodo.
- Iniciar el Registry y los nodos necesarios.
- Distribuir los ejecutables, librerías y ficheros que vayan a necesitar los programas en los nodos adecuados.

Las tres primeras deben ser llevadas a cabo por el administrador, pero para realizar la última podemos usar IceGrid. Mediante un servidor IcePatch2 es posible configurar los servidores para enviar los archivos necesarios en cualquier momento. En la figura 2.11 podemos ver la secuencia de acciones que se realiza cuando el administrador despliega la aplicación.

En primer lugar se guarda el descriptor en el Registry. A continuación, el administrador solicita al Registry que distribuya los archivos necesarios. El Registry avisa a todos los nodos configurados para distribución en la aplicación para que comiencen el proceso de descarga. Todos los nodos IceGrid son clientes de IcePatch2, y actúan como tales: descargándose todo si no existía ninguna copia anterior de la distribución, o descargándose selectivamente los archivos que contienen algún cambio (ver figura 2.6).

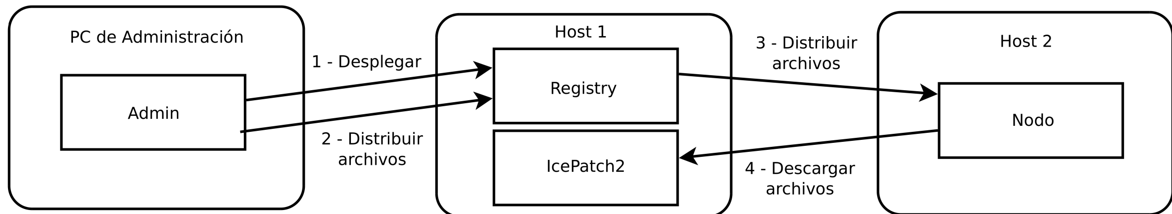

Figura 2.11: Proceso de distribución de archivos

Los servidores IcePatch2 incluyen las mismas características que los servidores genéricos IceGrid, incluyendo activación bajo demanda y administración remota. Es posible incluir más de un servidor IcePatch2 en cada aplicación con el fin de balancear la carga de trabajo cuando haya gran cantidad de archivos que distribuir.

Objetivos

En este capítulo se planteará el objetivo general de este proyecto, y se detallarán los objetivos específicos que se pretenden conseguir, con el fin de delimitar su alcance y describir su finalidad.

3.1 Objetivo general

El objetivo principal es diseñar e implementar una infraestructura de computación con posibilidad de adaptación automática de los recursos en función del uso y las necesidades de las aplicaciones ejecutadas (elasticidad).

La infraestructura debe asignar dinámicamente una colección de aplicaciones cualesquiera (correspondiente a una aplicación distribuida) a un conjunto de nodos con ciertas características independientes de la arquitectura del grid especificadas por el usuario. Dichas aplicaciones crecerán o decrecerán basándose en los recursos disponibles y los requisitos de la aplicación en un momento determinado.

Se utilizará para ello el middleware ZeroC Ice, que mediante su servicio IceGrid permite desplegar aplicaciones no elásticas en las que es necesario que el desarrollador conozca la arquitectura del grid y asigne los nodos explícitamente.

3.2 Objetivos específicos

A continuación se presentan los diferentes objetivos específicos que se pretenden alcanzar.

3.2.1 Identificación de características de los nodos

Será preciso estudiar e identificar las características de los nodos (hosts físicos) susceptibles de ser tomadas en cuenta a la hora de desplegar una aplicación desde el punto de vista del cliente.

Será necesario realizar una descripción de los nodos en función de esas características mediante un lenguaje formal.

Asimismo, habrá que plantear una forma de establecer grupos o categorías de nodos con unas características comunes.

3.2.2 Descripción de las aplicaciones

Será necesario establecer una forma de especificar las necesidades de recursos de cada aplicación, permitiendo al usuario configurar su aplicación según las características de los nodos y los grupos a los que pertenecen, en lugar de tener que especificar explícitamente los nodos de los que consta la aplicación como se hace con IceGrid (ver apartado 2.3.2).

El formato de esta descripción debe ser adecuado para generar después, considerando los componentes reales del grid, una descripción de la aplicación en el formato usado por IceGrid. Por lo tanto, también se deberá crear la herramienta encargada de procesar dicha descripción.

3.2.3 Monitorización del grid

Será necesario monitorizar el estado del grid y sus nodos para permitir que las aplicaciones se adapten en tiempo de ejecución a los cambios producidos en los recursos disponibles.

Puesto que IceGrid no ofrece la posibilidad de adaptar las aplicaciones automáticamente y las aplicaciones de IceCloud deben variar su configuración en función de los recursos disponibles en un momento determinado, se deberá crear una estructura que contenga las características de los nodos y aquellas variables que influyan en la selección de los nodos por parte del usuario, y procurar el mecanismo, mediante objetos distribuidos, para que los nodos puedan informar de su estado en un instante concreto.

Además, habremos de implementar una herramienta de monitorización activa que consulte periódicamente el estado de esos nodos.

3.2.4 Computación elástica

Otro de los objetivos es implementar los mecanismos necesarios que permitan la asignación dinámica de nodos (computación elástica). Es decir, que proporcionen al usuario la posibilidad de que su aplicación escale dinámicamente en tiempo de ejecución dependiendo de los recursos disponibles, creando nuevas instancias de sus servidores si fuese necesario, o liberando recursos mediante la detención de los servicios ejecutados en un momento dado, todo ello sin necesidad de modificar la aplicación y sin intervención alguna del usuario.

Se deberá crear un planificador que gestione el crecimiento y decrecimiento de las aplicaciones.

4 Método de trabajo y herramientas

En este capítulo se introduce la metodología elegida para el desarrollo del proyecto y se describen las herramientas, tanto hardware como software, utilizadas para realización del mismo

4.1 Metodología de trabajo

Al comenzar el proyecto, teníamos una idea general de los requisitos a desarrollar para implementar una plataforma de Cloud Computing, pero estos requisitos no eran detallados, por lo que se decidió optar por un modelo de desarrollo *evolutivo* con el que refinar los resultados progresivamente a lo largo de diferentes iteraciones.

El esquema 4.1 muestra el diagrama de flujo de un modelo evolutivo, en el que se generan varias versiones intermedias del sistema antes de obtener la versión final.

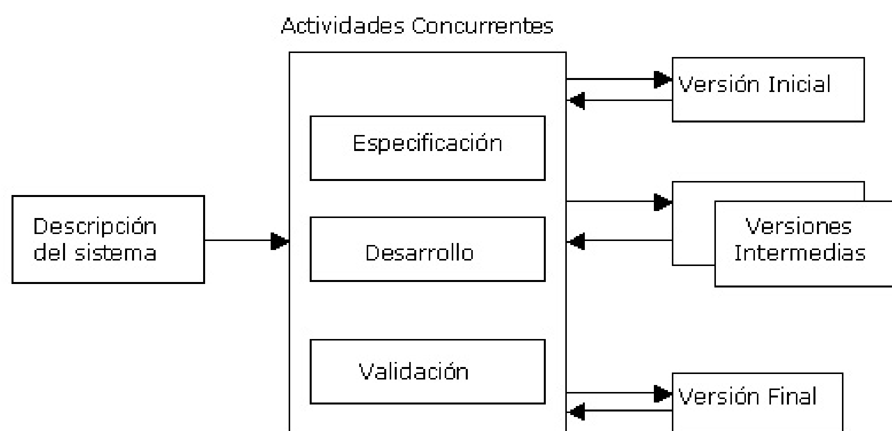

Figura 4.1: Diagrama de flujo del modelo evolutivo

4.1.1 Prototipado evolutivo

El *prototipado evolutivo* [Som05] es un paradigma basado en la realización de prototipos funcionales a lo largo del desarrollo del sistema hasta llegar a un producto final. Se parte de los requisitos mejor comprendidos y con mayor prioridad, y se van definiendo los detalles conforme avanza el desarrollo a través de diferentes versiones.

Con un paradigma de prototipado es esencial la comunicación con el cliente. Tras la elaboración de un prototipo, se entrega al cliente y éste evalúa los resultados, aportando retroalimentación para mejorar los requisitos [Pre10], sirviendo así como mecanismo para identificar los requisitos del software. El prototipo evaluado se modifica según los nuevos requisitos hasta transformase en el sistema real.

El esquema general de este paradigma se muestra en la figura 4.2. En cada iteración se diseña y construye un prototipo que es integrado en el sistema y puesto a disposición del cliente. Tras la entrega, se procede a realizar la siguiente iteración. Basándose en la evaluación del cliente se definen nuevas funcionalidades y se elabora el siguiente prototipo.

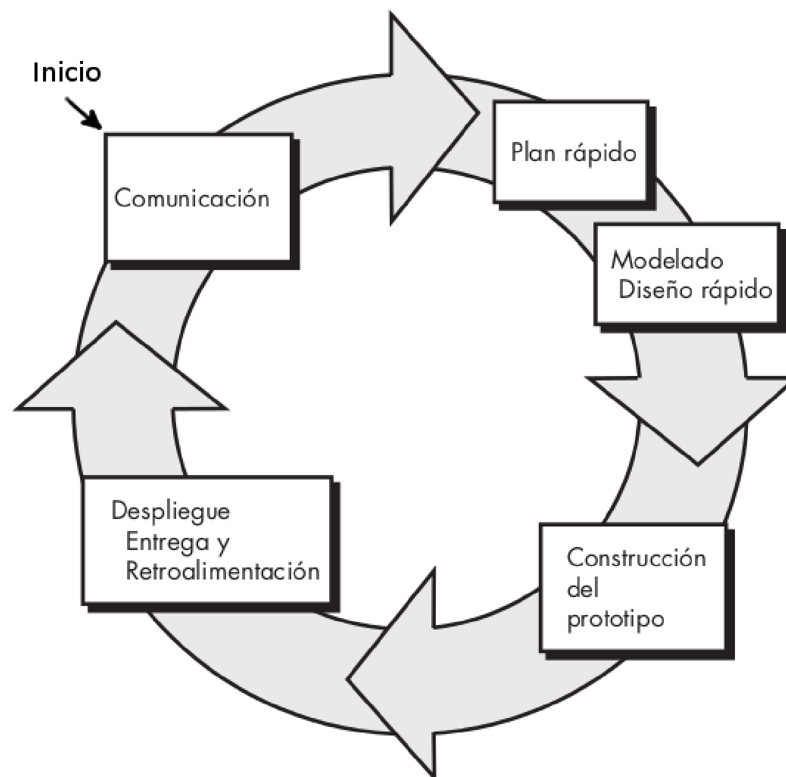

Figura 4.2: Paradigma de prototipado evolutivo

Las ventajas del prototipado evolutivo son las siguientes:

- Los clientes obtienen prototipos utilizables del sistema prácticamente desde el inicio del desarrollo, lo que aumenta la confianza en la obtención de un producto final que cumpla con los requisitos, ya que los prototipos aportan la sensación de sistema real que satisface las necesidades inmediatas del cliente.
- Al evaluar los prototipos intermedios, se facilita la tarea de identificar si los requisitos planteados son correctos y viables.
- Se minimiza el riesgo de errores, y éstos pueden solventarse más fácilmente. En cada iteración se realizan pruebas de los nuevos cambios y se comprueba que se siguen

cumpliendo los requisitos de las iteraciones anteriores. En caso de errores en una de las iteraciones, no se trasladan a las iteraciones posteriores.

- Las funcionalidades con mayor prioridad y más importantes son desarrolladas en las primeras fases, y son, por tanto, las más probadas a lo largo del proceso, lo que otorga robustez y fiabilidad al sistema.

No obstante, el prototipado evolutivo presenta generalmente dos problemas:

- Debido a la constante interacción con el cliente para introducir nuevos requisitos, los recursos del proceso de desarrollo son difíciles de estimar y no es fácil producir documentación que refleje cada versión del sistema.
- El mantenimiento del sistema puede ser problemático debido a la especificación de nuevas funcionalidades en su desarrollo puesto que los cambios continuos corrompen la arquitectura del software y hacen que introducir nuevos cambios sea cada vez más difícil y costoso.

Estas desventajas se hacen particularmente manifiestas en sistemas grandes. Para sistemas de pequeño y medio tamaño como el planteado en este PFC, el enfoque evolutivo no ofrece ningún problema, pero para sistemas grandes es recomendable usar un modelo híbrido que combine características de un modelo en cascada para desarrollar las partes del sistema bien conocidas y características del enfoque evolutivo para definir con más precisión los requisitos del sistema que presenten mayor incertidumbre.

4.2 Herramientas

4.2.1 Lenguajes de programación

Para la elaboración de este proyecto se han utilizado los siguientes lenguajes de programación:

Python Lenguaje de programación de alto nivel, interpretado, multiparadigma y orientado a objetos, desarrollado por Guido van Roosum. Ha sido el lenguaje para implementanción tanto del sistema, como de las pruebas realizadas debido a su sencillez y multitud de librerías.

Slice Lenguaje creado por ZeroC para definir las interfaces de los objetos y ofrecer independencia entre su especificación y la implementación de los mismos, permitiendo que dicha implementación pueda realizarse en distintos lenguajes.

XML (eXtensible Markup Language) Lenguaje de marcas desarrollado por World Wide Web Consortium (W3C) utilizado para almacenar datos de forma legible. Permite el intercambio de información estructurada entre diferentes plataformas. La estructura de un documento XML se compone de partes bien definidas representables en forma de árbol. Por ello, ha sido el lenguaje elegido para especificar categorías en las que

agrupar los nodos del sistema y para especificar las aplicaciones que deben ser desplegadas en IceCloud.

4.2.2 Herramientas hardware

Para el desarrollo de IceCloud se ha utilizado un equipo portátil con las siguientes características:

- Intel® Core™ i5-2450M. (2.50GHz x 4)
- 8GB RAM
- 500GB Disco duro.

4.2.3 Software

A continuación se detallan las herramientas y bibliotecas software que han sido necesarias para el desarrollo de este proyecto:

Herramientas de virtualización

- **Oracle VM VirtualBox** Es un software de virtualización desarrollado por Oracle que permite utilizar máquinas virtuales para simular máquinas reales. Ha sido utilizado para simular el grid sobre el que se ejecuta IceCloud.

Sistemas operativos

- **Linux Mint** Como principal sistema operativo en la realización de este proyecto se ha utilizado Linux Mint Mate 16 "Petra" con núcleo Linux 3.11.0-12-amd64 instalado en el equipo portátil indicado en la sección 4.2.2.
- **Debian GNU/Linux** En las máquinas virtuales utilizadas para formar parte del grid de IceCloud, se ha utilizado el sistema operativo Debian 7.0 "Wheezy" 32-bit con núcleo Linux 3.2.0-4-486.

Aplicaciones de desarrollo

- **ZeroC Ice** Middleware de comunicaciones orientado a objetos, multilenguaje y multiplataforma desarrollado por la empresa ZeroC [Zera]. Utilizado para realizar la comunicación entre clientes y servidores, y, especialmente como base para definir el grid y llevar a cabo el despliegue de aplicaciones mediante su servicio IceGrid.
- **Eclipse** Entorno de programación [Fou] utilizado para el desarrollo de la aplicación IceCloud. La versión utilizada ha sido 4.3 "Kepler", con el plugin **PyDev** [pyd] que ofrece un entorno de desarrollo de proyectos en Python para Eclipse.
- **PyUnit** (también conocido como **unittest**). Framework integrado en Eclipse mediante PyDev utilizado para la implementación y ejecución de pruebas en Python.

- **python-doublex** Framework de dobles de prueba para Python creado por David Villa [Vil]. Utilizado para la realización de los tests unitarios del sistema.
- **lxml** Biblioteca para el procesamiento de ficheros XML en Python [LXM]. Utilizada para procesar los archivos de definición de categorías y aplicaciones.
- **lshw** Herramienta que suministra información sobre el hardware de la máquina [LSH]. Con ella, es posible obtener información sobre la configuración de la memoria, la versión de CPU, firmware, caché, etc. Su salida se puede obtener en formato XML. Utilizada para obtener características de los nodos del grid.
- **Blender** Programa dedicado al modelado, iluminación, renderizado, animación y diseño de gráficos 3D. Se ha utilizado en la implementación de una aplicación distribuida para el renderizado de frames con la finalidad de probar IceCloud con una aplicación con utilidad real [BLE].

Documentación y gráficos

- **LaTeX** Lenguaje de marcado para composición y edición de documentos utilizado para la realización de esta memoria [CLM+03].
- **BibTex** Herramienta para la descripción de referencias para documentos escritos en LaTeX. Usada para crear las referencias de este documento.
- **Kile** Editor de documentos e intérprete de órdenes de archivos fuente TeX y LaTeX [PZH+].
- **Dia** Programa para la creación de diagramas, con soporte para modelado Unified Modeling Language (UML). Utilizado para la creación de algunos de los diagramas mostrados en este documento.
- **Gimp** Programa de manipulación de imágenes de GNU. Utilizado para la creación de algunas de las imágenes incluidas en esta memoria.

Control de versiones

- **Mercurial** Sistema de control de versiones distribuido. Usado para gestionar los cambios del código del proyecto y la documentación [MER14] [O'S], junto a un repositorio alojado en *BitBucket*[1].

[1] `https://bitbucket.org/arco_group/pfc.icecloud`

Desarrollo del proyecto

En este capítulo se describirá el proceso de desarrollo, detallando las decisiones tomadas en cada iteración, los prototipos resultantes y las pruebas realizadas para cada una de ellas.

5.1 Especificación de requisitos

Como ya comentamos en la sección 4.1, el paradigma elegido para el desarrollo de este proyecto ha sido **prototipado evolutivo**. Esto significa que los requisitos se han ido perfilando y añadiendo a lo largo de las distintas iteraciones. Los requisitos de partida, menos precisos y más generales, se muestran a continuación.

- Se utilizará el servicio *IceGrid*, que ofrecerá la base necesaria para el despliegue de aplicaciones.
- Los nodos deben poderse agrupar en distintas categorías según sus características físicas.
- Debe proponerse un mecanismo para describir aplicaciones.
- Los usuarios deberán poder añadir sus aplicaciones al sistema desarrollado.
- Los servidores de las aplicaciones no tendrán porqué estar ligados necesariamente a nodo de cómputo concreto, deben poder definirse en función de las categorías a las que pertenezcan los nodos.
- El sistema debe poder elegir dónde instanciar los servidores de las aplicaciones de entre los nodos de una misma categoría sin intervención del desarrollador de la aplicación.
- La aplicación debe ofrecer **elasticidad**. Los servidores deben tener la posibilidad de crecer y decrecer automáticamente según los cambios en el nodo.

5.2 Pruebas

Aunque algunas de las pruebas se han realizado sobre un grid real (virtualizado), para la elaboración de otras de las pruebas que se expondrán durante el proceso de desarrollo (ver sección 5.3) se han utilizado objetos simulados, ya que realizar pruebas sobre un grid real es

complejo y no aporta una diferencia significativa en cuanto a la validez del resultado de las mismas.

Los objetos simulados (dobles de prueba) son objetos que imitan el comportamiento de los objetos reales de forma controlada [MFC00]. Su implementación es más sencilla que la implementación del objeto real, y se usan, por ejemplo, para rellenar listas de parámetros, eludir parte de la implementación funcional o devolver respuestas predefinidas a las llamadas.

La estructura que se ha seguido para desarrollar las pruebas y para describirlas en este documento es conocida como patrón *Given - When - Then*:

- **Given** (Dado): El estado inicial en el que se encuentra el sistema sobre el que se producen los eventos.
- **When** (Cuando): Cuando se produce un evento determinado...
- **Then** (Entonces): Se comprueba que los resultados son los esperados.

5.3 Proceso de desarrollo

5.3.1 Iteración 1: Obtención de información del nodo y categorías

En la primera iteración, el objetivo es desarrollar un mecanismo para la creación de categorías en función de las características de los nodos, de tal modo que los usuarios puedan especificar los nodos de sus aplicaciones en función de esas categorías.

Diseño e implementación

Para esta primera aproximación, la única característica que consideraremos será la cantidad total de memoria RAM del computador donde se ejecuta el nodo. Para conseguir esta información, usaremos la herramienta `lshw`, que suministra información sobre la máquina. Con ella, es posible obtener, entre otros datos, información sobre la configuración de la memoria, y su salida puede ser especificada en formato XML.

Es necesario disponer de un servidor con un sirviente que respalde a un objeto que proporcione la información del nodo, y de un cliente que obtenga dicha información. Para representar esa información, se creó una estructura `NodeInfoData`, cuyo Slice aparece en el listado 5.1.

```
1 struct NodeInfoData {
2     int id;
3     long ram;                    // bytes
4 };
```

Listado 5.1: SLICE de NodeInfoData. Iteración 1

Los nodos tendrán como nombre `grid-nodeX`, siendo X un índice numérico identificativo del nodo.

Para llevar a cabo la comunicación de los datos del nodo, se ha implementado el servidor ICE `ServerNodeI`, que crea un adaptador de objetos `NodeInfoAdapter` y registra en él un objeto que implementa la interfaz `NodeInfo`, que se muestra en el listado 5.2. En cada uno de los nodos IceCloud, se ejecutará una instancia de este servicio. El proxy textual del objeto `NodeInfo` que describe las características del nodo `grid-nodeX` será:

`nodeinfoX@NodeInfoServerX.NodeInfoAdapter`

```
1 interface NodeInfo {
2     idempotent NodeInfoData getNodeInfo();
3 };
```

Listado 5.2: Interfaz Slice que implementa `NodeInfoServer`

También se ha diseñado e implementado en esta iteración un cliente que consulta el estado de los nodos. Dicho cliente se ha llamado `ClientNodeI`, y se encarga de obtener la estructura `NodeInfoData` del objeto `NodeInfo` de cada nodo.

Por otro lado, necesitamos una forma de representar las categorías. Las especificaremos en ficheros XML. Por cada característica a representar, incluiremos un elemento `feature` con un atributo `id` que identifique a la característica. Para cada categoría a definir, incluiremos un elemento `category` con un atributo `name` que la identifique y unos valores máximos y mínimos que deben cumplir los miembros de la categoría. Un ejemplo de fichero XML para describir categorías se muestra en el listado 5.3.

Para agrupar los nodos por categorías, se crea una función: `getNodesByCategory(nodesInfoDict, categories_path)` que, dados todos los nodos y el fichero de categorías, asigne al nodo las categorías cuyas características coincidan con las suyas. Esta función devuelve un diccionario con los identificadores de los nodos como clave y una lista de categorías a las que pertenece el nodo como valor.

Pruebas

En esta iteración se llevaron a cabo dos grupos de pruebas, para probar tanto los servidores como el cliente. Estas pruebas corresponden a código refactorizado que no aparece en la versión final.

- Por una parte, para comprobar que los objetos `NodeInfo` y los servidores que los respaldaban se comportaban de forma correcta, se llevó a cabo la siguiente prueba:

```xml
<?xml version="1.0"?>
<list>
  <feature id="memory">
    <category name="great_mem">
      <min_mem units="bytes"> 8000000000 </min_mem>
      <max_mem units="bytes"> 10000000000000000000 </max_mem>
    </category>
    <category name="big_mem">
      <min_mem units="bytes"> 200000000 </min_mem>
      <max_mem units="bytes"> 8000000000 </max_mem>
    </category>
    <category name="med_mem">
      <min_mem units="bytes"> 128000000 </min_mem>
      <max_mem units="bytes"> 200000000 </max_mem>
    </category>
    <category name="low_mem">
      <min_mem units="bytes"> 0 </min_mem>
      <max_mem units="bytes"> 128000000 </max_mem>
    </category>
  </feature>
</list>
```

Listado 5.3: Ejemplo de fichero de categorías

Dados:	Varios servidores `ServerNodeI`, ejecutándose en diferentes entornos (5 máquinas virtuales distintas), de características conocidas.
Cuando:	Un cliente obtiene solicita la información de cada nodo.
Entonces:	La información recibida corresponde con la real.

- Para probar el correcto funcionamiento de la función en el cliente que asigna categorías a los nodos, se realizó la siguiente prueba:

Dados:	Una lista de objetos NodeInfoData. Un fichero con una especificación de categorías.
Cuando:	Se obtiene el diccionario de categorías por nodo.
Entonces:	Las categorías asignadas a cada nodo son las correctas.

5.3.2 Iteración 2: Descripción de aplicaciones

Tras evaluar el prototipo obtenido en la iteración 1, se procede a añadir una nueva funcionalidad que permita al usuario describir las necesidades de sus propias aplicaciones.

Diseño e implementación

Es necesario establecer un formato para describir esas aplicaciones, con el cual sea posible configurar los servidores en función de las categorías de los nodos.

Optamos por el lenguaje XML, para poder aprovechar todas las características de IceGrid, basándonos en su formato para describir las aplicaciones.

Este formato está disponible en el anexo A, y será frecuentemente nombrado en este apartado.

La primera modificación realizada sobre el formato XML de IceGrid es el elemento raíz, que pasará de llamarse `icegrid` a `icecloud`. El resto de elementos se mantienen iguales, introduciendo además los elementos que iremos detallando en este apartado.

Para poder incluir los servidores en función de nodos definidos por su categoría, se añade al formato XML típico IceGrid (consultar anexo A) una modificación en el formato de sus nodos.

Una aplicación IceCloud podrá tener nodos con nombre, que se corresponderán con el nombre de los nodos IceGrid del grid, al igual que en las aplicaciones IceGrid, y nodos con categorías en lugar de nombre. Dentro de los nodos con categorías, se podrán especificar servidores, y éstos serán alojados en un nodo IceGrid (*icegridnode*) que cumpla las condiciones indicadas.

De esta forma, se podrán especificar aplicaciones como la del listado 5.4. En este ejemplo, vemos una aplicación con un nodo IceGrid especificado, y otro nodo definido por su categoría, que podría corresponderse o no con el *icegridnode* indicado anteriormente.

```
1  <icecloud>
2    <application name="MinimalApplication">
3      <node name="grid-node0">
4        <server id="Server0" activation="manual" exe="./server.py">
5        </server>
6      </node>
7      <node category="great-mem">
8        <server id="Server1" activation="manual" exe="./server.py">
9        </server>
10     </node>
11   </application>
12 </icecloud>
```

Listado 5.4: Ejemplo de aplicación IceCloud con servidor

Para tratar estos ficheros XML, se ha implementado un módulo llamado `Parser`, encargado de transformar los archivos XML que describen aplicaciones IceCloud en archivos que describan aplicaciones IceGrid. Este módulo debe cubrir los siguientes requisitos:

- Leer el fichero, y extraer los nodos definidos mediante categorías.
- Seleccionar el *icegridnode* correspondiente a cada nodo definido mediante categoría.

- Sustituir los nodos definidos mediante categorías por nodos IceGrid concretos. Para ello, deberá hacer uso de la función `getNodesByCategory(nodesInfoDict, categories_path)` creada en la iteración anterior.
- Eliminar nodos duplicados.
- Con la información procesada, crear un nuevo archivo XML con el formato de IceGrid.

Para llevar a cabo el procesamiento de los archivos XML, se ha utilizado la biblioteca *lxml*, que proporciona una API para el tratamiento de archivos XML y HTML, permitiendo buscar y seleccionar elementos teniendo en cuenta la estructura jerárquica de estos lenguajes.

Para coordinar los elementos de los que dispone de momento el prototipo, se crean los módulos `IceCloudManager`, y `ICRegistry`.

- El módulo **ICRegistry** es la evolución del cliente `ClientNodeI`, desarrollado en la anterior iteración. Sigue siendo un cliente, que se encarga de recopilar toda la información de todos los `ServerNodeI`, y a su vez implementa un sirviente de tipo ICRegistry, cuya interfaz se ve definida en el listado 5.5. Mediante el método `getAllNodesInfo()` suministra la información de todos los nodos, indicando el índice del nodo al que pertenece dicha información.
  El Registry de IceGrid ofrece una interfaz administrativa que permite a los clientes monitorizar, actualizar y recibir actualizaciones del Registry. Mediante ella, podemos obtener el nombre de todos los nodos. Para acceder a ella, es necesario crear una sesión. El módulo `ICRegistry` hace uso de una de esas sesiones.
- El módulo **IceCloudManager** implementa un cliente que consulta el método `getAllNodesInfo()` del ICRegistry. También incluye las funciones desarrolladas en la iteración anterior encargadas de obtener las categorías a partir del archivo y se ocupa de obtener el XML de IceGrid mediante un objeto `Parser` partiendo de la información obtenida con los métodos anteriores. El procedimiento para crear dicho archivo se muestra en el diagrama de secuencia 5.1, en el que se obvia la obtención de los proxies correspondiente al modelo de objetos distribuidos.

```
1 dictionary<int, NodeInfoData> NodeInfoDataMap;

3 interface ICRegistry {
4   idempotent NodeInfoDataMap getAllNodesInfo();
5 };
```

Listado 5.5: SLICE de la interfaz ICRegistry

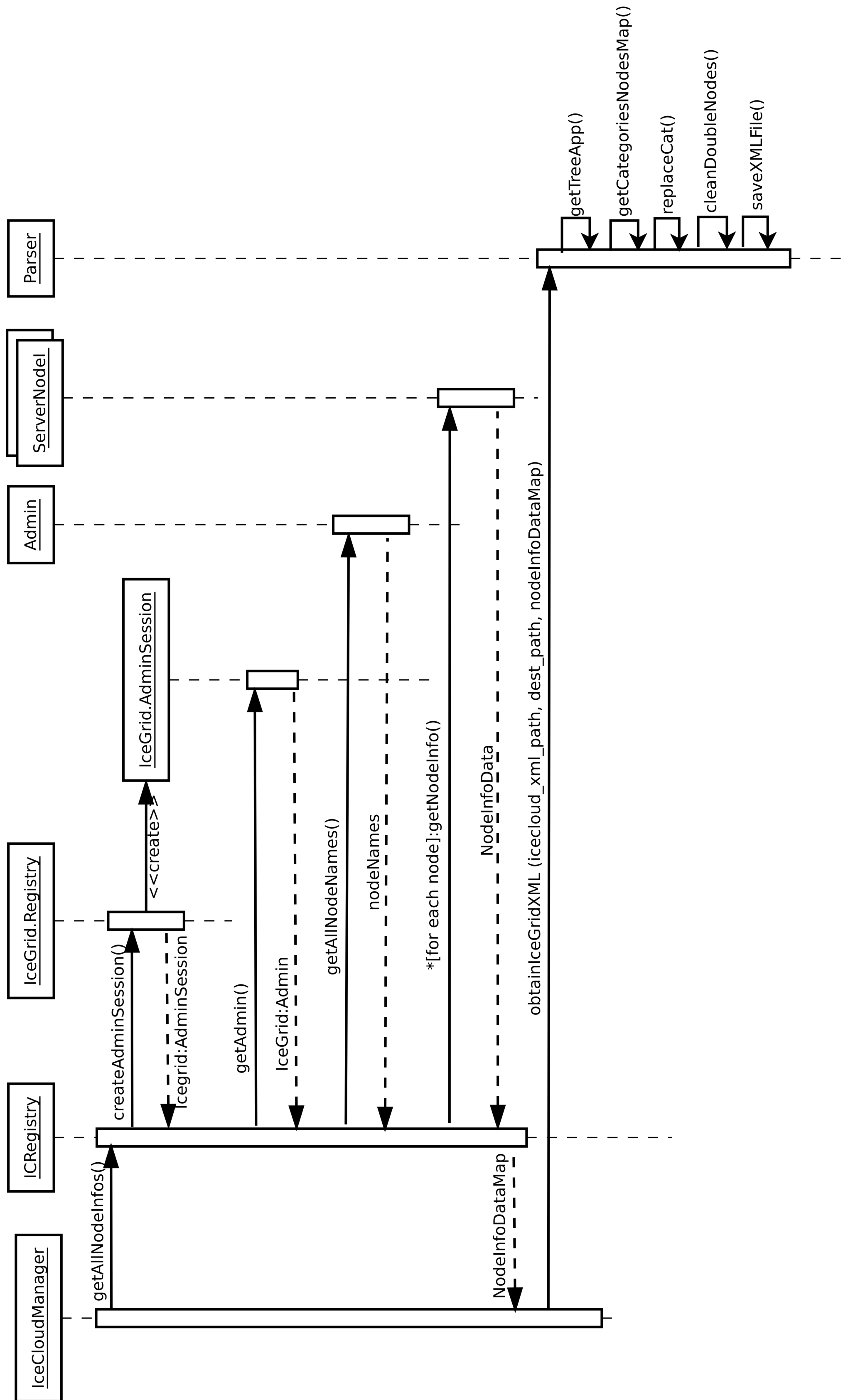

Figura 5.1: Diagrama de secuencia para la obtención del fichero IceGrid XML

Pruebas

Para verificar el correcto funcionamiento de los métodos del módulo `Parser` se implementaron las siguientes pruebas. Debido a refactorizaciones posteriores propias de la naturaleza de un desarrollo evolutivo, el código de estas pruebas no se encuentra en la versión final.

- A los nodos definidos por categorías se les asigna un nodo real correcto:

Dados:	Una aplicación con nodos definidos por categorías. Un diccionario con categorías asignadas a cada `icegridnode`.
Cuando:	Se obtiene el `icegridnode` correspondiente a cada nodo con categoría.
Entonces:	El `icegridnode` asignado es el esperado.

- El fichero XML generado tiene la estructura correcta:

Dados:	Una serie de elementos `XML` de tipo `node`. Un diccionario con un `icegridnode` asignado a cada elemento `node` con categoría.
Cuando:	Se obtiene el árbol que representa al archivo `XML` con categorías reemplazadas por `icegridnode`.
Entonces:	La estructura del árbol es correcta.

- El fichero XML generado tiene la estructura correcta:

Dada:	Una estructura de árbol que representa a un `XML` con nodos repetidos.
Cuando:	Se solicita al `Parser` que fusione los nodos repetidos.
Entonces:	La estructura del árbol es correcta y no hay nodos repetidos.

- Para probar el funcionamiento del módulo `IceCloudManager` se implementó la siguiente prueba:

Dados:	Un diccionario de índices de `icegridnode` con sus NodeInfoData asociados. La ruta a un fichero de categorías.
Cuando:	Se obtienen las categorías posibles para cada nodo.
Entonces:	Las posibles categorías son correctas.

5.3.3 Iteración 3: Despliegue de aplicaciones

Tras evaluar los resultados de la iteración anterior, se decide añadir la funcionalidad que permita el despliegue de la aplicación. Es decir, se incorporan los mecanismos necesarios para enviar la aplicación al Registry de IceGrid.

Diseño e implementación

Como comentamos en la sección 2.3, el despliegue de la aplicación se puede llevar a cabo de diversas formas:

- Mediante la interfaz gráfica de `IceGrid-GUI`.
- Mediante la herramienta *icegridadmin* por línea de comandos.
- Mediante una sesión administrativa IceGrid.

Las dos primeras opciones permiten añadir una aplicación definida en un archivo XML. Elegir cualquiera de estas dos alternativas facilitaría la tarea de añadir la aplicación, puesto que permitiría utilizar el archivo XML generado en la iteración 2; pero la primera presenta el inconveniente de no poder ser integrada en el código y ambas presentan el inconveniente de depender de herramientas externas al API de IceGrid. Por lo tanto, el despliegue de la aplicación se realizará mediante la sesión administrativa del API de IceGrid.

Para añadir una aplicación al Registry mediante una sesión administrativa de IceGrid, es necesario suministrar al Registry una instancia de la estructura `ApplicationDescriptor` ofrecida por el API de IceGrid.

Puesto que IceGrid no implementa un constructor para este tipo de estructura a partir del archivo XML para el lenguaje Python, ha sido necesario crear una clase `DescriptorBuilder` que transforme el archivo XML generado en la iteración anterior en un `ApplicationDescriptor`.

Para ello, se ha analizado la organización de dicha estructura, y sus componentes se explican a continuación. Dados los múltiples niveles de la estructura, se muestran diagramas de clases fragmentados para facilitar su legibilidad. Para describir cada característica referente a una aplicación, IceGrid requiere de una serie de descriptores, que son los siguientes (por orden alfabético):

- **AdapterDescriptor**: Describe un adaptador de objetos. Su estructura se puede consultar en la figura 5.2. Tiene los siguientes atributos:
  - **name**: El nombre del adaptador de objetos.
  - **description**: Una descripción del adaptador de objetos.
  - **id**: El identificador del adaptador de objetos.
  - **replicaGroupId**: El grupo de réplica al que pertenece el adaptador.
  - **priority**: La prioridad del adaptador. Se usa por los grupos de réplica para establecer el orden en que sirven los endpoints.
  - **registerProcess**: Especifica si el adaptador registra algún objeto de tipo *Process*.
  - **serverLifetime**: Indica si el tiempo de vida del adaptador es el mismo que el del servidor.

- **objects**: Una lista de elementos ObjectDescriptor que representan los objetos bien conocidos de este adaptador.
- **allocatables**: Una lista de elementos ObjectDescriptor que describe los objetos «asignables» registrados en el adaptador de objetos.

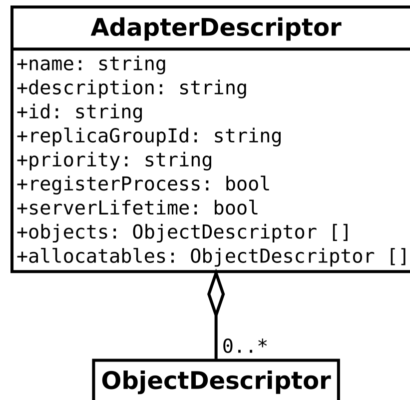

Figura 5.2: Diagrama de clases de AdapterDescriptor

- **ApplicationDescriptor**: Define una aplicación. Su diagrama puede consultarse en la figura 5.3. Sus atributos son los siguientes:
  - **name**: El nombre de la aplicación. De tipo cadena.
  - **variables**: Un diccionario con las variables definidas para la aplicación y sus valores.
  - **replicaGroups**: Una lista de ReplicaGroupDescriptor para especificar los grupos de replica de la aplicación.
  - **serverTemplates**: Las plantillas de servidores, especificadas con un diccionario cuyas claves son los nombres de las plantillas y los valores son de tipo TemplateDescriptor.
  - **serviceTemplates**: Las plantillas para servicios, especificadas con un diccionario cuyas claves son los nombres de las plantillas y los valores son de tipo TemplateDescriptor.
  - **nodes**: Los nodos de la aplicación, especificados mediante un diccionario cuyas claves son los nombres de los nodos y los valores son de tipo NodeDescriptor.
  - **distrib**: Un elemento de tipo DistributionDescriptor para especificar los detalles sobre la distribución de la aplicación.
  - **description**: Una cadena para describir la aplicación textualmente.
  - **propertySets**: Un diccionario de PropertySetDescriptor, con su nombre como clave.
- **CommunicatorDescriptor**: Usado para describir características de un servidor o servicio. Su diagrama se puede consultar en la figura 5.4, y tiene los siguientes miembros:

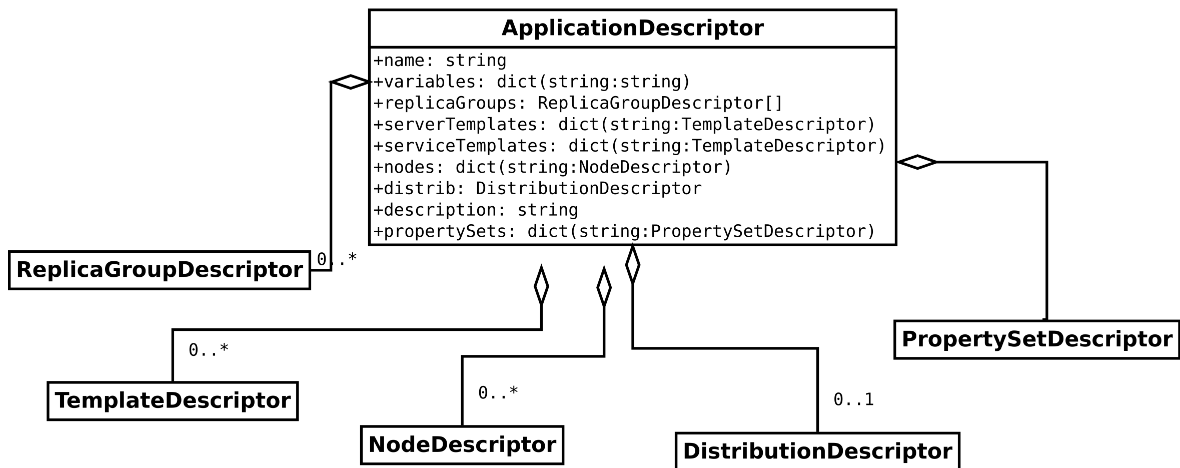

Figura 5.3: Diagrama de clases de ApplicationDescriptor

- **adapters**: Una lista de objetos AdapterDescriptor que describen los adaptadores del servidor/servicio.
- **propertySet**: Un conjunto de propiedades. De tipo PropertySetDescriptor.
- **dbEnvs**: Lista de características de la base de datos. De tipo DbEnvDescriptor.
- **logs**: Lista de cadenas con la ruta a los archivos de log.
- **description**: Una cadena con una descripción textual.

- **DbEnvDescriptor**: Describe un entorno de base de datos Freeze. Su diagrama se puede consultar en la figura 5.5, y sus miembros son los siguientes:
  - **name**: Una cadena con el nombre del entorno.
  - **description**: Una cadena con la descripción del entorno.
  - **dbHome**: Una cadena con la ruta del directorio donde se almacenarán los archivos de la base de datos. Si se especifica, el directorio debe existir; si no, se creará uno por defecto.
  - **properties**: Las propiedades de configuración del entorno. Es una lista con elementos de tipo PropertyDescriptor.

- **DistributionDescriptor**: Define un servidor IcePatch2 y los directorios en los que se encuentran los archivos a distribuir. Puede verse su diagrama en la figura 5.6, y sus atributos son los siguientes:
  - **icepatch**: Una cadena con el proxy del servidor IcePatch2.
  - **directories**: Una lista de cadenas con los directorios.

- **IceBoxDescriptor**: Define un servidor IceBox. Hereda de ServerDescriptor, y su diagrama puede verse en la figura 5.4. Añade el siguiente atributo:
  - **services**: Una lista de elementos de tipo ServiceInstanceDescriptor que especifican las instancias de servicios que ofrece el servidor.

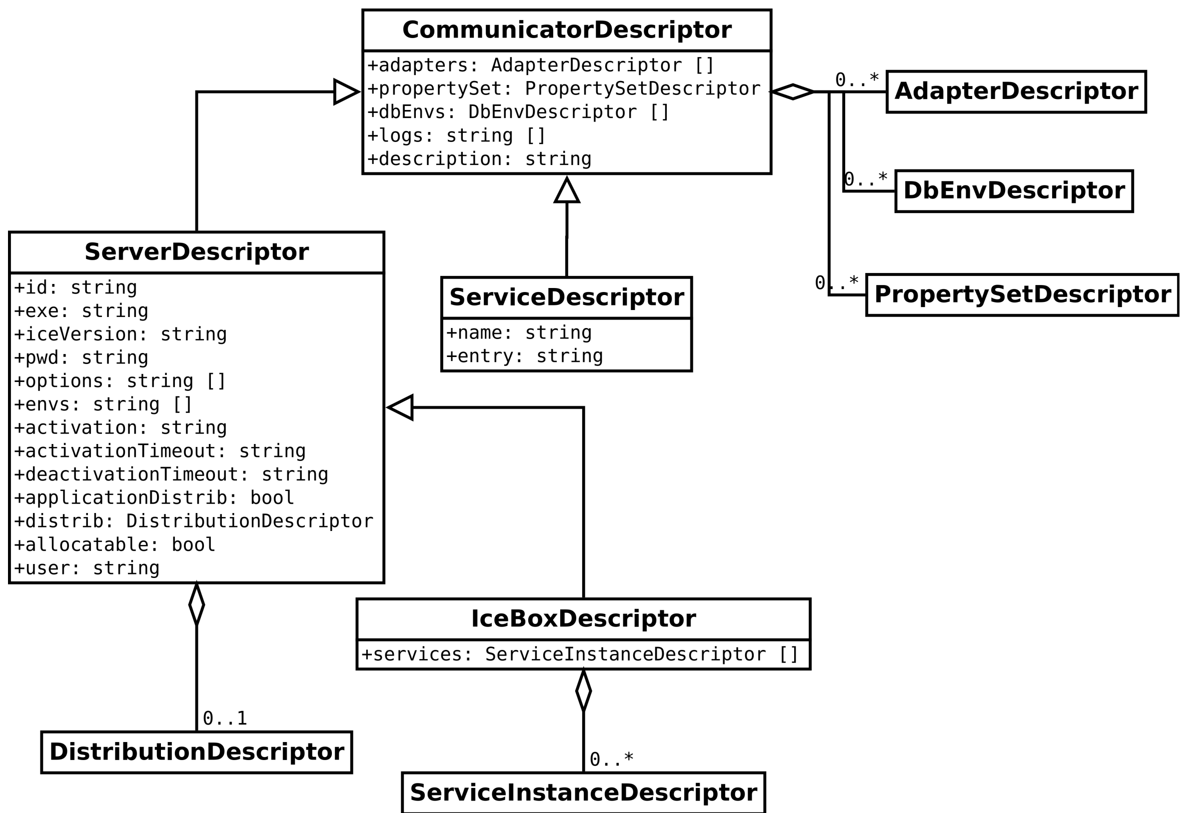

Figura 5.4: Diagrama de clases de CommunicatorDescriptor

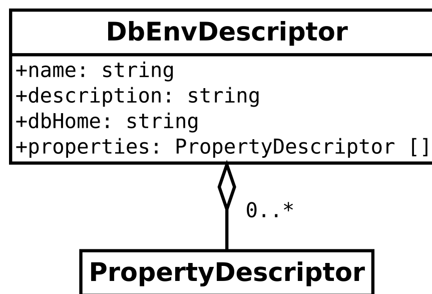

Figura 5.5: Diagrama de clases de DbEnvDescriptor

Figura 5.6: Diagrama de clase de DistributionDescriptor

- **LoadBalancingPolicy**: Clase base para las políticas de balanceo de carga. Permite especificar el número de réplicas que se usarán para definir los endpoints de un grupo de réplica mediante el atributo `nReplicas`, de tipo cadena. Siempre forma parte de un ReplicaGroupDescriptor, cuyo diagrama puede consultarse en la figura 5.11.

De esta clase heredan:

- **AdaptiveLoadBalancingPolicy**: Balanceo de carga según la ocupación de CPU. Permite especificar si se desea tomar la media de los últimos 1, 5 ó 15 minutos con el atributo `loadSample`.
- **OrderedLoadBalancingPolicy**: Balanceo de carga en determinado orden.
- **RandomLoadBalancingPolicy**: Balanceo de carga aleatorio.
- **RoundRobinLoadBalancingPolicy**: Balanceo de carga rotatorio.

- **NodeDescriptor**: Descripción de un nodo. Su diagrama se muestra en la figura 5.7. Sus atributos son los siguientes:
  - **variables**: Un diccionario para definir las variables del nodo.
  - **serverInstances**: Una lista de elementos ServerInstanceDescriptor para indicar las plantillas de servidor que deben instanciarse en el nodo.
  - **servers**: Una lista de elementos ServerDescriptor para definir los servidores (que no son instancia de una plantilla).
  - **loadFactor**: Una cadena para especificar el factor de carga del nodo.
  - **description**: Una cadena con una descripción del nodo.
  - **propertySets**: Un diccionario de elementos PropertySetDescriptor indexados por su nombre.

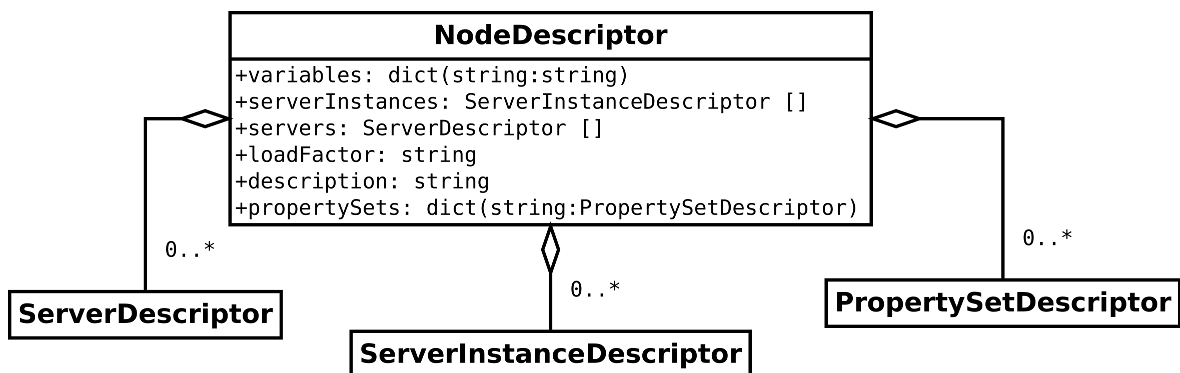

Figura 5.7: Diagrama de clase de NodeDescriptor

- **ObjectDescriptor**: Describe un objeto Ice. Su diagrama se muestra en la figura 5.8 y consta de los siguientes atributos:
  - **id**: Un elemento de tipo `Ice.Identity`, que determina la identidad del objeto.
  - **type**: Una cadena para definir el tipo del objeto.
  - **proxyOptions**: Una cadena especificando el proxy creado para el objeto. Si se omite, se creará a partir de las opciones de proxy especificadas en el adaptador donde está registrado el objeto o su grupo de réplica.

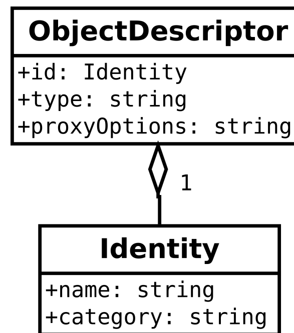

Figura 5.8: Diagrama de clase de ObjectDescriptor

- **PropertyDescriptor**: Define una propiedad. Su diagrama se encuentra en la figura 5.9 y tiene los siguientes atributos:
  - **name**: El nombre de la propiedad.
  - **value**: El valor de la propiedad.

Figura 5.9: Diagrama de clase de PropertyDescriptor

- **PropertySetDescriptor**: Describe un conjunto de propiedades. Su diagrama se presenta en la figura 5.10 y tiene los siguientes miembros:
  - **references**: Una lista de cadenas con referencias a conjuntos de propiedades con nombre.
  - **properties**: Una lista de elementos PropertyDescriptor con las propiedades.

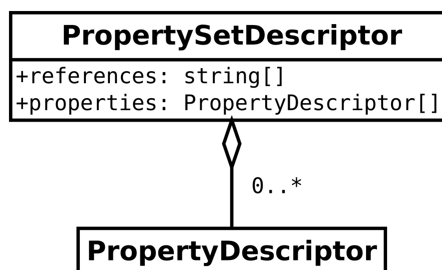

Figura 5.10: Diagrama de clase de PropertySetDescriptor

- **ReplicaGroupDescriptor**: Describe un grupo de réplica. Su diagrama se muestra en la figura 5.11, y tiene los siguientes atributos:
  - **id**: El identificador del grupo de réplica. De tipo cadena.

- **loadBalancing**: De tipo LoadBalancingPolicy, especifica la política de balanceo de carga.
- **proxyOptions**: De tipo cadena, determina las opciones por defecto de los proxies creados para el grupo de réplica.
- **objects**: Una lista de elementos ObjectDescriptor para indicar los objetos asociados al grupo.
- **description**: Una descripción textual del grupo.

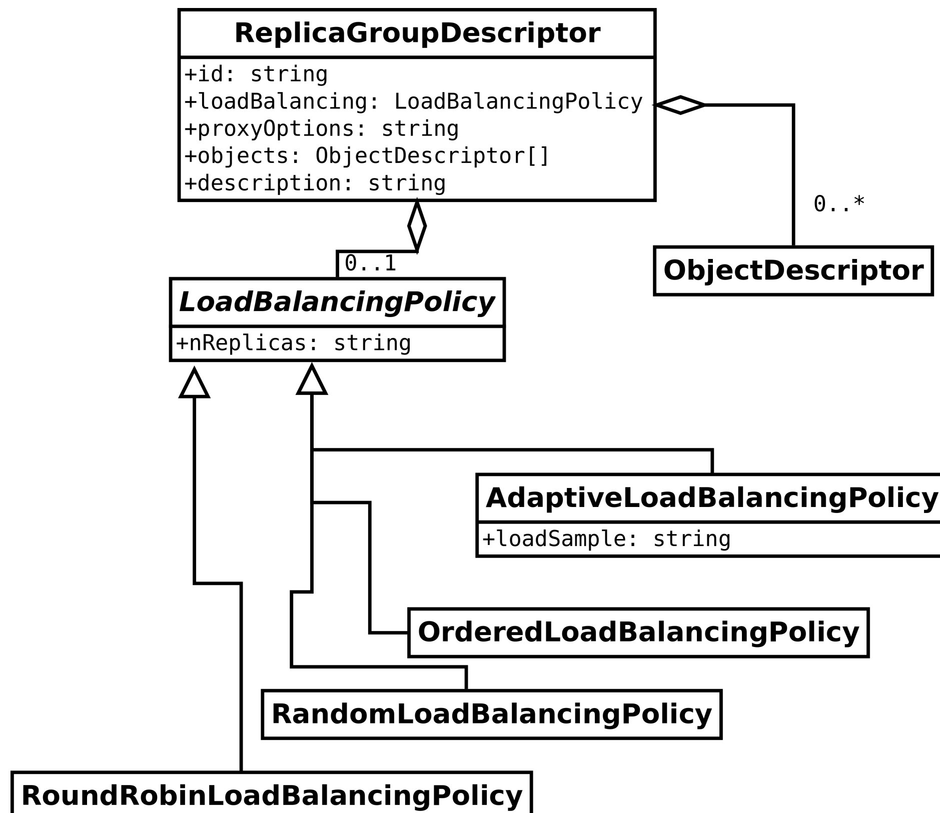

Figura 5.11: Diagrama de clase de ReplicaGroupDescriptor

- **ServerDescriptor**: Define un servidor Ice. Hereda de la clase CommunicatorDescriptor (ver diagrama 5.4). Tiene los siguientes atributos propios:
  - **id**: El identificador del servidor. De tipo cadena.
  - **exe**: La ruta al ejecutable del servidor.
  - **iceVersion**: La versión de ICE usada por el servidor.
  - **pwd**: La ruta al directorio de trabajo del servidor.
  - **options**: Una lista de opciones para pasar por linea de comandos al ejecutable del servidor.
  - **envs**: Una lista de variables de entorno.
  - **activation**: El modo de activación del servidor («on-demand» o «manual»).

- **activationTimeout**: El número de segundos a esperar para la activación del servidor.
- **deactivationTimeout**: El número de segundos a esperar para la desactivación del servidor.
- **applicationDistrib**: De tipo booleano. Indica si la distribución del servidor depende de la de la aplicación.
- **distrib**: De tipo DistributionDescriptor, describe la distribución del servidor.
- **allocatable**: De tipo booleano, indica si el servidor es «asignable».
- **user**: El usuario bajo el que se ejecuta el servidor.

- **ServerInstanceDescriptor**: Describe la instancia de una plantilla de servidor. Su diagrama se puede consultar en la figura 5.12, y tiene los siguientes atributos:
  - **template**: Una cadena con el nombre de la plantilla usada por la instancia.
  - **parameterValues**: Un diccionario de parámetros, con su nombre y su valor.
  - **propertySet**: Un conjunto de propiedades, de tipo PropertySetDescriptor.
  - **servicePropertySets**: Un diccionario con conjuntos de propiedades para servicios. Sólo es válido si la plantilla usada es de un servidor IceBox.

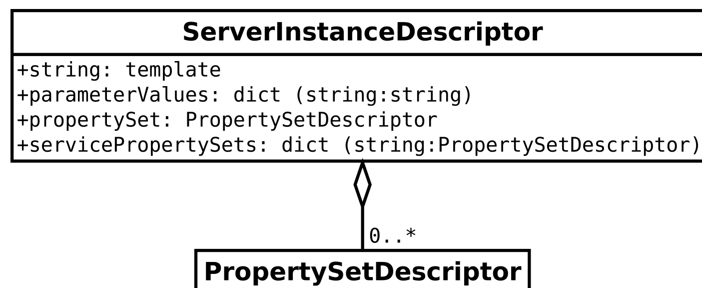

Figura 5.12: Diagrama de clase de ServerInstanceDescriptor

- **ServiceDescriptor**: Describe un servicio IceBox. Hereda de la clase CommunicatorDescriptor (ver figura 5.4). Tiene los siguientes atributos propios:
  - **name**: El nombre del servicio. De tipo cadena.
  - **entry**: El acceso al servicio. De tipo cadena.
- **ServiceInstanceDescriptor**: Describe la instancia de una plantilla de servidor. Su diagrama se puede consultar en la figura 5.13, y tiene los siguientes atributos:
  - **template**: Una cadena con el nombre de la plantilla usada por la instancia.
  - **parameterValues**: Un diccionario de parámetros, con su nombre y su valor.
  - **descriptor**: De tipo ServiceDescriptor. Si el atributo «template» está vacío, describe el servicio.

- **servicePropertySets**: Un diccionario con conjuntos de propiedades. De tipo PropertySetDescriptor.

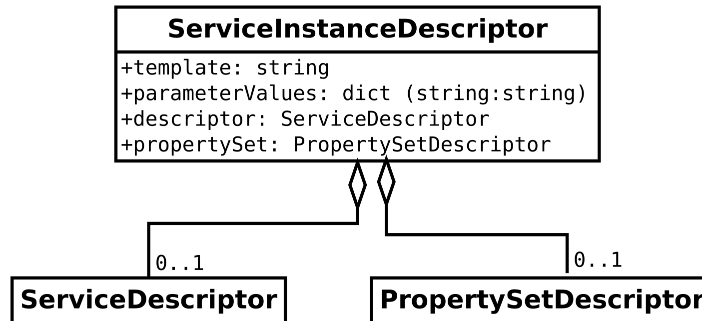

Figura 5.13: Diagrama de clase de ServiceInstanceDescriptor

- **TemplateDescriptor**: Describe una plantilla para servidores o servicios. Su diagrama se muestra en la figura 5.14. Sus atributos son los siguientes:
  - **descriptor**: La especificación del servicio o servidor. De tipo CommunicatorDescriptor.
  - **parameters**: Una lista de los parámetros necesarios para instanciar la plantilla.
  - **parameterDefaults**: Un diccionario con los parámetros por defecto y sus valores.

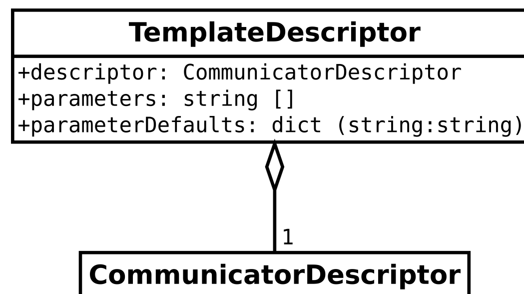

Figura 5.14: Diagrama de clase de TemplateDescriptor

Una vez estudiados los componentes del `ApplicationDescriptor`, y los elementos XML con los que se corresponden, se diseñó la clase `DescriptorBuilder`, encargada de analizar el fichero y generar el ApplicationDescriptor correspondiente.

A continuación, se añadieron al ICRegistry las operaciones:

- `addApplication`, que toma como entrada el ApplicationDescriptor de IceGrid generado a partir del XML de la aplicación del usuario, e invoca al método `addApplication` del objeto `Admin` de la sesión administrativa de IceGrid.

- `patchApplication`, que tomando como argumento el nombre de la aplicación, se encarga de llevar a cabo la distribución de la misma si ésta tiene configurado un servidor IcePatch2 correctamente.

Pruebas

Para comprobar que la generación de los diversos componentes del ApplicationDescriptor de IceGrid se realizaba correctamente, se realizó la captura de un ApplicationDescriptor de una aplicación concreta (que incluía todos los elementos posibles) generado por IceGrid Admin, se creó uno con el mismo formato y se realizaron las siguientes pruebas:

- Comprobación del atributo `variables` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getVariables` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de variables de la aplicación.
Entonces:	El diccionario del ApplicationDescriptor real y el generado coinciden.

- Comprobación del atributo `replicaGroups` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getReplicaGroups` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtienen los grupos de réplica del XML.
Entonces:	Las dos listas tienen el mismo número de elementos. Las dos listas son del mismo tipo. Para cada grupo de réplica en la lista generada, existe otro en la lista original cuyos atributos coinciden con el generado (12 comprobaciones).

- Comprobación del atributo `properties` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getPropertySetDict` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de propiedades de la aplicación del XML.
Entonces:	El diccionario de propiedades generado y el original son iguales.

- Comprobación del atributo `distrib` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getDistributionDescriptor` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el DistributionDescriptor de la aplicación.
Entonces:	El servidor especificado y la lista de directorios son los mismos en ambos descriptores.

- Comprobación del atributo `serverTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getServerTemplates_dict` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servidores especificadas en el XML.
Entonces:	Tanto el diccionario original como el generado tienen el mismo número de elementos.
	Cada clave del diccionario original está en el generado.

- Comprobación del atributo `parameterDefaults` de los elementos `TemplateDescriptor` del atributo `serverTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getServerTemplates_templates_parameterDefaults` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servidores especificadas en el XML.
Entonces:	Los parámetros por defecto de cada plantilla generada son iguales que los de su correspondiente original.

- Comprobación del atributo `parameters` de los elementos `TemplateDescriptor` del atributo `serverTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getServerTemplates_templates_parameters` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servidores especificadas en el XML.
Entonces:	La lista de parámetros de cada plantilla generada se corresponde con una idéntica en la plantilla original.

- Comprobación del atributo `descriptor` de los elementos `TemplateDescriptor` del atributo `serverTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getServerTemplates_templates_descriptor` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servidores especificadas en el XML.
Entonces:	Cada plantilla generada contiene un CommunicatorDescriptor equivalente en la plantilla original. Se comprueba el tipo de CommunicatorDescriptor y cada atributo (15 comprobaciones por plantilla).

- Comprobación del atributo `serviceTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:test_getServiceTemplates_dict` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servicios especificadas en el XML.
Entonces:	Tanto el diccionario original como el generado tienen el mismo número de elementos. Cada clave del diccionario original está en el generado.

- Comprobación del atributo `parameterDefaults` de los elementos `TemplateDescriptor` del atributo `serviceTemplates` del ApplicationDescriptor.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:`

test_getServiceTemplates_templates_parameterDefaults disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servicios especificadas en el XML.
Entonces:	Los parámetros por defecto de cada plantilla generada son iguales que los de su correspondiente original.

- Comprobación del atributo parameters de los elementos TemplateDescriptor del atributo serviceTemplates del ApplicationDescriptor.
  Esta prueba se encuentra en el método tests/test_ic_descriptor_builder.py: test_getServiceTemplates_templates_parameters disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servicios especificadas en el XML.
Entonces:	La lista de parámetros de cada plantilla generada se corresponde con una idéntica en la plantilla original.

- Comprobación del atributo descriptor de los elementos TemplateDescriptor del atributo serviceTemplates del ApplicationDescriptor.
  Esta prueba se encuentra en el método tests/test_ic_descriptor_builder.py: test_getServiceTemplates_templates_descriptor disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario con las plantillas de servicios especificadas en el XML.
Entonces:	Cada plantilla generada contiene un CommunicatorDescriptor equivalente en la plantilla original. Se comprueba el tipo de CommunicatorDescriptor y cada atributo (15 comprobaciones por plantilla).

- Comprobación del atributo nodes del ApplicationDescriptor.
  Esta prueba se encuentra en el método tests/test_ic_descriptor_builder.py: test_getNameNodeDescriptors_dict disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de nodos de la aplicación del XML.
Entonces:	Los nombres de los nodos del ApplicationDescriptor original son los mismos que el generado. También son el mismo número.

- Comprobación de los atributos de cada nodo de la aplicación (excepto servidores).
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getNameNodeDescriptors_attributes` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtienen los nodos definidos en la aplicación del XML.
Entonces:	Se comprueba que los atributos (excepto el atributo `servers`) de cada nodo en el ApplicationDescriptor generado coinciden con los de su homónimo en el original.

- Comprobación de los servidores de cada nodo (excepto servidores IceBox).
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getNameNodeDescriptors_servers` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtienen los nodos definidos en la aplicación del XML.
Entonces:	Para cada servidor en cada nodo, se comprueba que coincide con uno del nodo original en todos sus atributos.

- Comprobación de los servidores IceBox de cada nodo.
  Esta prueba se encuentra en el método `tests/test_ic_descriptor_builder.py:` `test_getNameNodeDescriptors_servers_icebox` disponible en el repositorio:

Dados:	Una aplicación IceGrid definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtienen los nodos definidos en la aplicación del XML.
Entonces:	Para cada servidor IceBox en cada nodo, se comprueba que coincide con uno del nodo original en todos sus atributos.

5.3.4 Iteración 4: Asignación dinámica de nodos

Tras la evaluación del resultado de la iteración anterior, se ha propuesto introducir un mecanismo para establecer una prioridad de elección de nodo dentro de una misma categoría en el caso de los servidores especificados mediante las mismas. Esto conlleva la introducción de una serie de cambios, tanto en la especificación de las aplicaciones como en su representación interna.

También se han introducido nuevos criterios para definir las categorías, lo que ha supuesto la modificación de la forma de definir los nodos y una reestructuración de las clases de IceCloud.

Diseño e implementación

Para poder definir las categorías teniendo en cuenta más parámetros, nos basaremos en las siguientes herramientas, que proporcionan información sobre la configuración de la máquina:

- **lshw**: *lshw* [LSH] suministra información sobre el hardware de la máquina (ver 4.2.3).
- El archivo `/proc/cpuinfo`: en sistemas GNU/Linux, este archivo proporciona información sobre las características de la CPU, tales como modelo, fabricante, tamaño de caché, etc.
- **lscpu**: este comando ofrece información sobre las características de la CPU. Incluye, por ejemplo, información sobre el número de hilos y cores.

Tras estudiar las características que nos ofrece cada herramienta, decidimos que será necesario completar la estructura *NodeInfoData* con la siguiente información:

- Ram (ram): Total de memoria ram, en MiB.
- Anchura de la arquitectura (width): 32 ó 64 bits.
- Modelo de CPU (cpuModel): El nombre del modelo de CPU, con el mismo formato que en el archivo `/proc/cpuinfo`. Por ejemplo, *Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz*.
- Número de cores (cpuNproc): El número de núcleos de procesamiento de la CPU.
- Característica personalizable (custom): Se incluye una característica para que se puedan especificar otras opciones no obtenibles mediante herramientas software, pero que podrían influir en la decisión del usuario a la hora de desplegar sus aplicaciones, como por ejemplo la región en la que se encuentran las máquinas. El usuario podría configurar esta característica al registrarse en el sistema, por ejemplo.

Por otro lado, se adapta la forma de representar las categorías a las decisiones tomadas. Las categorías se podrán anidar para definir otras categorías más extensas. Las propiedades

que representen una magnitud numérica de rango se restringirán con un valor mínimo y otro máximo, y las textuales con un sólo valor. Las características de la categoría que no se especifiquen no serán restringidas. Se permiten las siguientes características:

- `ram`: correspondiente con el campo *ram* de la estructura NodeInfoData.
- `width`: correspondiente con el campo *width* de la estructura NodeInfoData.
- `cpu-model`:correspondiente con el campo *cpuModel* de la estructura NodeInfoData.
- `n-proc`: correspondiente con el campo *cpuNproc* de la estructura NodeInfoData.
- `custom`: correspondiente con el campo *custom* de la estructura NodeInfoData.

En el listado 5.6 se puede ver un ejemplo de especificación de categorías.

```
<list>

    <category name="any-server">
    </category>

  <category name="great-server64">
    <include>great-mem</include>
    <feature id="width">
      <value>64</value>
    </feature>
  </category>


  <category name="great-mem">
    <feature id="ram">
      <min_value>7000</min_value>
      <max_value>10000000</max_value>
    </feature>
  </category>

</list>
```

Listado 5.6: Ejemplo de especificación de categorías

La descripción de NodeInfoData considerada hasta ahora, sólo aporta información estática sobra las características de los nodos. No proporciona información sobre características que varíen a lo largo de la ejecución de las aplicaciones, lo que imposibilita la capacidad de **asignar dinámicamente** nodos a los servidores o la **elasticidad** del grid. Por ello, se decide enriquecer la información aportada tomando las siguientes medidas:

- Los nodos aportarán también información sobre la memoria RAM libre en un momento determinado.
- Se desacopla el cliente de los servidores NodeInfo del ICRegistry. Se crea un objeto (local) de la clase `NodeInfoRetriever`, que será invocado por ICRegistry y se ocupará, además de consultar los objetos NodeInfo, de aportar información extra de los

nodos. Para obtener esa información extra, usará una sesión administrativa, de la que ya se habló en otras iteraciones. Para no crear múltiples sesiones, ICRegistry ejecutará un hilo que mantenga una sesión activa (las sesiones administrativas expiran después de un tiempo, y es necesario llamar a su método *keepAlive()* periódicamente para evitarlo), y ésta será compartida por todos los objetos que necesiten usarla.

El resultado de aplicar esas medidas además de la modificación de las clases nombradas es la creación de la estructura `ExtendedNodeInfo` (con la información que completa `NodeInfoRetriever`) y la modificación de `NodeInfoData` como se ve en el listado 5.7.

```
1  struct NodeInfoData {
2    long ram;                    // MiB
3    int width;                   //Architecture width 32-64
4    string cpuModel;             // As in /proc/cpuinfo
5    string cpuVendor;            // As in /proc/cpuinfo
6    int cpuNproc;                // Number of processing cores
7    string cpuArchitecture;      // CPU architecture as in lscpu
8    int freeRam;                 //MiB
9    int custom;                  // custom.info file contents
10 };

12 struct ExtendedNodeInfo{
13   NodeInfoData hostInfo;
14   IceGrid::LoadInfo loadInfo;      // CPU load in the last 1, 5, and 15
       min
15   int numberOfServers;              // number of server in the node
16 };
```

Listado 5.7: `NodeInfoData` y `ExtendedNodeInfo`

Para aprovechar estas características y permitir la *asignación dinámica* de los servidores, se introduce un cambio a la hora de describir los servidores en la aplicación en formato XML. Este cambio es la introducción del atributo `allocationRule`, que permitirá elegir un nodo con una característica dinámica concreta de los existentes en la categoría.

Los valores de dicho atributo pueden ser simples, como los mostrados en el cuadro 5.1, o compuestos. Los valores compuestos tendrán como primer término uno de los indicados en el cuadro 5.2, como segundo un operador de los indicados en el cuadro 5.3 y como tercer término un valor numérico.

Cuando un servidor contenga el atributo `allocationRule` con un valor compuesto, y no se encuentre un nodo con esas características, la aplicación no se desplegará.

Para almacenar la especificación de la aplicación en el ICRegistry, se ha creado una estructura `ApplicationDescriptor` propia de IceCloud, similar al ApplicationDescriptor de IceGrid estudiado en la iteración 3 (subsección 5.3.3). Los cambios introducidos respecto a la estructura IceGrid son los siguientes:

Valor	Descripción
min_servers	Servidor con el mínimo número de servidores alojados
max_servers	Servidor con el máximo número de servidores alojados
min_load_1	Servidor con la mínima cantidad de carga de CPU en el último minuto
max_load_1	Servidor con la máxima cantidad de carga de CPU en el último minuto
min_load_5	Servidor con la mínima cantidad de carga de CPU en los últimos 5 minutos
max_load_5	Servidor con la máxima cantidad de carga de CPU en los últimos 5 minutos
min_load_15	Servidor con la mínima cantidad de carga de CPU en los últimos 15 minutos
max_load_15	Servidor con la máxima cantidad de carga de CPU en los últimos 15 minutos
min_free_ram	Servidor con la menor cantidad de memoria RAM libre
max_free_ram	Servidor con la mayor cantidad de memoria RAM libre
max_custom	En caso de que el usuario haya personalizado el nodo, el mayor valor.
mim_custom	En caso de que el usuario haya personalizado el nodo, el menor valor.

Cuadro 5.1: Opciones para el atributo *allocationRule*

Valor	Descripción
num_servers	Número de servidores alojados
cpu_load_1	Carga de CPU en el último minuto
cpu_load_5	Carga de CPU en los últimos 5 minutos
cpu_load_15	Carga de CPU en los últimos 5 minutos
free_ram	Memoria libre
num_custom	Valor personalizado

Cuadro 5.2: Primer término del atributo compuesto *allocationRule*

Operador	Descripción
==	Igual
=	Igual
le	Menor o igual
<=	Mayor o igual
ge	Mayor o igual
>=	Mayor o igual
lt	Menor que
<	Menor que
gt	Mayor que
>	Mayor que

Cuadro 5.3: Operadores para atributo compuesto *allocationRule*

- Se añaden los atributos `categoryNodes`, para la descripción de nodos con categorías (ver definición Slice en el listado 5.8). Para definir estos nodos, se añade la estructura IceCloud.NodeDescriptor, cuyo Slice puede consultarse en el listado 5.9.

```
struct ApplicationDescriptor {
  string name;
  IceGrid::StringStringDict variables;
  IceGrid::ReplicaGroupDescriptorSeq replicaGroups;
  IceGrid::TemplateDescriptorDict serverTemplates;
  CategoryTemplateDescriptorDict categoryServerTemplates;
  IceGrid::TemplateDescriptorDict serviceTemplates;
  IceCloud::NodeDescriptorDict categoryNodes;
  IceGrid::NodeDescriptorDict nodes;
  ServerInstanceDescriptorSeq categoryServerInstances;
  IceGrid::DistributionDescriptor distrib;
  string description;
  IceGrid::PropertySetDescriptorDict propertySets;
};
```

Listado 5.8: Slice del descriptor de una aplicación

```
struct NodeDescriptor {
  ServerDescriptorSeq servers;
  string description;
};
```

Listado 5.9: Slice del descriptor de un nodo definido por categoría

- Se añade el atributo `allocationRule` a la clase `ServerDescriptor` como se muestra en el listado 5.10.

Se ha renombrado el `DescriptorBuilder` creado en la iteración anterior a `ICDescriptorBuilder`, y se ha adaptado para que cree un ApplicationDescriptor de IceCloud a partir del XML. Se ha creado otra clase `IGDescriptorBuilder` que genera un ApplicationDescriptor de IceGrid a partir del ApplicationDescriptor de IceCloud. La finalidad de esto es poder usar ICRegistry como un servidor sin necesidad de tener que mandar el XML por la red, sino una estructura bien definida en Slice. Además, esto podrá permitir guardar las aplicaciones en ICRegistry como objetos en lugar de cadenas. Ahora, el método *addApplication* de ICRegistry toma como entrada un ApplicationDescriptor de IceCloud, y llama al método *addApplication* del objeto `Admin` de la sesión administrativa de IceGrid con el `ApplicationDescriptor` de IceGrid generado con `IGDescriptorBuilder`. Se elimina por tanto la clase `Parser` que se encargaba de pasar de un XML IceCloud a otro IceGrid.

Pruebas

Para verificar el funcionamiento de los cambios implementados en la forma de describir la aplicación, se añadieron las siguientes pruebas:

```
sequence<ServerDescriptor> ServerDescriptorSeq;

class ServerDescriptor extends IceGrid::CommunicatorDescriptor {
  string id;
  string allocationRule;
  string exe;
  string iceVersion;
  string pwd;
  Ice::StringSeq options;
  Ice::StringSeq envs;
  string activation;
  string activationTimeout;
  string deactivationTimeout;
  bool applicationDistrib;
  IceGrid::DistributionDescriptor distrib;
  bool allocatable;
  string user;
};
```

Listado 5.10: Slice del descriptor de un servidor

- Comprobación del atributo `categoryNodes` del ApplicationDescriptor.
  Esta prueba corresponde al método `tests/test_ic_descriptor_builder.py:test_-getCategoryNodeDescriptors_dict` disponible en el repositorio:

Dados:	Una aplicación IceCloud definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de nodos definidos por categorías de la aplicación del XML.
Entonces:	Los nombres de los nodos del ApplicationDescriptor original son los mismos que el generado.

- Comprobación de los servidores de cada nodo (excepto servidores IceBox).
  Esta prueba corresponde al método `tests/test_ic_descriptor_builder.py:test_-getCategoryNodeDescriptors_servers` disponible en el repositorio:

Dados:	Una aplicación IceCloud definida mediante un XML.
	El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de nodos definidos por categorías de la aplicación del XML.
Entonces:	Para cada servidor en cada nodo, se comprueba que coincide con uno del nodo original en todos sus atributos.

- Comprobación de los servidores IceBox de cada nodo.
  Esta prueba corresponde al método `tests/test_ic_descriptor_builder.py:test_-getCategoryNodeDescriptors_servers_icebox` disponible en el repositorio:

Dados:	Una aplicación IceCloud definida mediante un XML. El ApplicationDescriptor correspondiente a esa aplicación.
Cuando:	Se obtiene el diccionario de nodos definidos por categorías de la aplicación del XML.
Entonces:	Para cada servidor IceBox en cada nodo, se comprueba que coincide con uno del nodo original en todos sus atributos.

Para verificar que la asignación de servidores con `allocationRule` se realizaba correctamente, se realizaron las siguientes pruebas:

- No hay nodos que coincidan con una categoría indicada.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_-test_not_matching_nodes_for_category` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un sólo servidor alojado en un nodo de categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con un sólo nodo que no corresponde a la categoría «great-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	No hay nodos para la categoría (lanza la excepción *NoMatchingNodesForCategory*).

- No hay nodos que cumplan la regla especificada en `allocationRule`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_-test_not_matching_node_in_category_for_rule` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un sólo servidor alojado en un nodo de categoría «great-mem», con `allocationRule = çpu_load_1 gt 5"`. - Un *NodeInfoRetriever* que devuelve una diccionario con un sólo nodo que no corresponde a la categoría «great-mem», con `cpu_load_1 = 5`.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	No hay nodos en la categoría para la regla indicada (lanza la excepción *NoMatchingNodesInCategoryForRule*).

- Se asigna correctamente un servidor definido en un nodo por categoría.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_-node_assignment_is_right_1_server` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un sólo servidor alojado en un nodo de categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con un nodo que pertenece a la categoría «great-mem», y otro que no pertenece a dicha categoría.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo correcto.

- Se asigna correctamente un servidor definido en un nodo sin categoría, y otro definido en un nodo con categoría.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_-node_assignment_is_right_1_server_1_static` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem», y otro alojado en un nodo no definido por categoría. - Un *NodeInfoRetriever* que devuelve una diccionario con un nodo que pertenece a la categoría «great-mem», y otro que no pertenece a dicha categoría.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Ambos servidores han sido añadidos al nodo correcto.

- Se asigna correctamente un servidor de una categoría, existiendo varios nodos de esa categoría.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_-node_assignment_is_right_1server_2nodes_same_category` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», y otro que pertenece a la categoría «low-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado a un nodo «great-mem».

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `min_servers`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_min_servers` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "min_servers"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinto número de servidores.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con menor número de servidores.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `max_servers`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_max_servers` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "max_servers"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinto número de servidores.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con mayor número de servidores.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `min_load1`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_min_load1` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "min_load1"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en el último minuto.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con menor carga de CPU en el último minuto.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `max_load1`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_max_load1` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "max_load1"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en el último minuto.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con mayor carga de CPU en el último minuto.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `min_load5`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_min_load5` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "min_load5"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en los últimos 5 minutos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con menor carga de CPU en los últimos 5 minutos.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `max_load5`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_max_load5` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "max_load5"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en los últimos 5 minutos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con mayor carga de CPU en los últimos 5 minutos.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `max_load15`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_max_load15` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "max_load15"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en los últimos 15 minutos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con mayor carga de CPU en los últimos 15 minutos.

- Se asigna correctamente un servidor de una categoría, con criterio de asignación `min_load15`.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_right_allocation_min_load15` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con un servidor alojado en un nodo de categoría «great-mem» y `allocationRule = "min_load15"` - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con distinta carga de CPU en los últimos 15 minutos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	El servidor ha sido asignado al nodo con menor carga de CPU en los últimos 15 minutos.

- Se asignan correctamente dos servidores de la misma categoría, al nodo con menos servidores.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_node_assignment_is_right_2_servers_same_category_with_rule_min_server` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con dos servidores alojados en un nodo de categoría «great-mem», y ambos con `allocationRule = min_servers`. - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con ningún servidor en ellos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Cada servidor se ha añadido a un nodo diferente.

- Se asignan correctamente cuatro servidores de la misma categoría, al nodo con menos servidores.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_node_assignment_is_right_4_servers_same_category_with_rule_min_server` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con cuatro servidores alojados en un nodo de categoría «great-mem», y ambos con `allocationRule = min_servers`. - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con ningún servidor en ellos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	A cada nodo se le han asignado 2 servidores.

- Se asignan correctamente dos servidores de la misma categoría con diferente criterio de asignación.
  Esta prueba corresponde al método `tests/test_server_assignment.py:test_node_assignment_is_right_2_servers_same_category_different_rule_different_node` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con dos servidores alojados en un nodo de categoría «great-mem», uno de ellos con `allocationRule = min_servers` y el otro con `allocationRule = min_load_5`. - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con diferentes números de servidores y diferente `cpu_load_5`
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Ambos servidores se asignaron al nodo correcto.

5.3.5 Iteración 5: Elasticidad

Después de evaluar los resultados de la iteración 4, se propone agregar la funcionalidad necesaria para lograr la *elasticidad* del grid, es decir, que los servidores puedan crecer o decrecer dinámicamente dependiendo de las capacidades del nodo en que se encuentren. Para ello, duplicaremos los servidores que lo requieran.

Diseño e implementación

Para cumplir esta funcionalidad, necesitaremos una estructura para guardar el estado del grid, que permita ser accedida y modificada cuando los nodos no satisfagan los requisitos de las aplicaciones. Esta clase se ha llamado *GridState*, e incluye los siguientes atributos:

- Un diccionario con las aplicaciones añadidas al ICRegistry, representadas por su ApplicationDescriptor de IceCloud.
- Una lista con la asignación original de los servidores al añadir la aplicación.
- Una lista para cada servidor original con sus servidores replicados.

Esta clase posee una variable *lock* con la que restringir el acceso y evitar problemas de concurrencia al modificar/consultar el estado del grid.
El objeto que define el estado del grid es conocido por ICRegistry, y modificado cada vez que se añade una aplicación.

Para definir los requisitos de los servidores, se han incluido los atributos `growingRule` y `decreasingRule` (regla de crecimiento y decrecimiento, respectivamente) a la estructura ServerDescriptor usada para describir los servidores de una aplicación. Para especificar estas reglas en el archivo XML, se siguen las mismas reglas que para especificar la regla `allocationRule` como atributo compuesto (ver tablas 5.2 y 5.3). En el listado 5.11 se puede ver un ejemplo de uso de este tipo de reglas. También se ha añadido un atributo `maxDupes` para limitar el número de duplicados que puede tener un servidor. Los servidores se duplicarán creando un servidor con las mismas características en el nodo preciso, cuyo nombre será el nombre del servidor original más un índice concatenado.

```
1  <icecloud>
2      <application name="MinimalApplication">
3          <node category="great-mem">
4              <server id="Server0" allocationRule= "min_servers"
5                  growingRule="num_servers gt 4"
6                  decreasingRule="num_servers le  2"
7                  activation="manual" exe="./server.py"
8                  maxDupes = "4">
9              </server>
10         </node>
11     </application>
12 </icecloud>
```

Listado 5.11: Ejemplo de especificación de categorías

Por otra parte, necesitamos una herramienta que consulte periódicamente el estado de los nodos y lo contraste con el estado del grid y los requisitos de las aplicaciones, para que los servidores que lo necesiten puedan crecer o decrecer. Con este fin, el servidor ICRegistry ejecutará en paralelo un hilo de la clase `MonitorTimer`, que a su vez que se encargará de llamar al objeto responsable de la comprobación del estado de los nodos y el grid.

Este último objeto, responsable de la monitorización del estado del grid, ha sido definido mediante la clase `Monitor`. Cuando su método *run()* es invocado, el objeto solicita a `NodeInfoRetriever` la información de los nodos, y para cada servidor activo de cada aplicación comprueba si se cumplen o no las reglas de decrecimiento. Si dichas reglas no se cumplen, ordenará que los servidores se adapten dinámicamente a esta la situación.

La clase responsable de ejecutar las órdenes del `Monitor`, es la clase `Scheduler`. Mediante una instancia de esta clase, el `Monitor` solicitará el crecimiento o decrecimiento de los servidores. El objeto `Scheduler` comprueba los cambios necesarios son posibles, atendiendo a la descripción del servidor original (la categoría del nodo, el criterio de asignación y el máximo de copias), del estado del grid y de los nodos. Este objeto planifica la ubicación de los nuevos servidores, y decide qué servidores deben ser eliminados.

Para actualizar el estado de la aplicación, además de modificar el objeto *GridState*, el *Scheduler* debe informar de la nueva situación al Registry de IceGrid, para que el despliegue de la aplicación cambie. Esta actualización se lleva a cabo mediante una sesión administrativa de IceGrid y el método *updateApplication* de su objeto *Admin*. Este método toma como parámetro un objeto de tipo `ApplicationUpdateDescriptor` de IceGrid, que el *Scheduler* construye según la planificación decidida. Cuando el *Monitor* ha comprobado todos los servidores de una aplicación, ejecuta el método *commit()* del objeto *Scheduler*, que se encargará de enviar la información al Registry. El objeto `Scheduler` también almacena los cambios en un fichero de registro (*log*) e informa de los mismos mediante la publicación de eventos en un canal de eventos IceStorm.

Pruebas

Para comprobar que la elasticidad del grid funciona correctamente, se ejecutaron las siguientes pruebas:

- Comprobación de que un servidor crece según su regla de crecimiento.
  Esta prueba corresponde al método `tests/test_scheduler.py:test_grow` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con un servidor con criterios de asignación (min_servers) y decrecimiento (num_servers gt 3). - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones del servidor (con distinto número de servidores), y en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo con menor número de servidores cumple la regla de crecimiento del servidor. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	El servidor se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler duplica el servidor en el nodo con menos servidores.

- El servidor no puede crecer porque ha alcanzado el máximo de duplicados.
  Esta prueba corresponde al método `tests/test_scheduler.py:test_not_grow_max_dupes` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con un servidor con el máximo de duplicados establecido a 0, con condiciones de crecimiento y condición de asignación de mínimo de servidores. - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones del servidor con diferente número de servidores, y en su segunda consulta un nodo (el que antes tenía el mínimo de servidores) que cumple la condición de crecimiento del servidor. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación y posteriormente se ejecuta el método *run()* del Monitor.
Entonces:	El servidor debería crecer por su condición de crecimiento, pero no se duplica por alcanzar el máximo de copias.

- Comprobación de que un servidor decrece según su regla de decrecimiento.
  Esta prueba corresponde al método `tests/test_scheduler.py:test_decrease` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con un servidor con criterios de asignación (min_servers) y decrecimiento (num_servers gt 3). - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones del servidor (con distinto número de servidores), en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo con menor número de servidores cumple la regla de crecimiento del servidor, y en su tercera consulta suministra información de esos nodos con características que cumplen la regla de decrecimiento del servidor. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	El servidor se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler duplica el servidor en el nodo con menos servidores.
Cuando:	Se ejecuta el método run() del Monitor por segunda vez, y el servidor duplicado cumple la regla de decrecimiento.
Entonces:	El Scheduler ha eliminado el duplicado del servidor.

- Un servidor original (especificado en la definición de la aplicación, no una copia) cumple la regla de decrecimiento, pero no se elimina este servidor sino una copia.
  Esta prueba corresponde al método `tests/test_scheduler.py:test_decrease_original_node` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un `XML`, con un servidor con criterios de asignación (min_servers) y decrecimiento (num_servers gt 3). - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones del servidor (con distinto número de servidores), en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo con menor número de servidores cumple la regla de crecimiento del servidor, y en su tercera consulta suministra información de esos nodos con características que cumplen la regla de decrecimiento del servidor original. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	El servidor se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler duplica el servidor en el nodo con menos servidores.
Cuando:	Se ejecuta el método run() del Monitor por segunda vez y el servidor original cumple la regla de decrecimiento.
Entonces:	El Scheduler ha eliminado el duplicado del servidor.

5.3.6 Iteración 6: Inclusión de plantillas de servidores

Dado que IceGrid permite describir servidores mediante plantillas e instanciar luego varias copias de una misma plantilla, se decidió agregar esta funcionalidad a las aplicaciones IceCloud tras evaluar la iteración anterior.

Diseño e implementación

El objetivo es permitir especificar plantillas con requisitos de categoría, criterios de asignación, crecimiento, decrecimiento y máximo de instancias para que se se asignen y escalen según el estado de los nodos.

Para describir las plantillas, se ha creado una estructura

`CategoryTemplateDescriptor` con los siguientes campos:

- **`descriptor`**: El descriptor del servidor.
- **`parameters`**: Los parámetros necesarios para instanciar la plantilla.
- **`parameterDefaults`**: El valor por defecto de los parámetros.
- **`category`**: La categoría de los nodos donde debería instanciarse esta plantilla.

- **allocationRule**: El criterio de asignación entre nodos de la categoría que deberían cumplir las instancias de la plantilla.
- **growingRule**: La regla de crecimiento de las instancias.
- **decreasingRule**: La regla de decrecimiento de las instancias.
- **maxInstances**: El máximo de instancias permitido de esta plantilla.

Para incluir este tipo de plantillas en el formato XML que describe una aplicación, la etiqueta requerida es `category-server-template`. Estos elementos deben ser hijos del elemento `application`. La plantilla se describe del mismo modo que los elementos `server-template`, con la diferencia de que un elemento `category-server-template` puede contener los atributos `category` (obligatorio), `allocationRule`, `growingRule`, `decreasingRule` y `maxInstances`.

Para instanciar un elemento de estas plantillas, se hará del mismo modo que para instanciar una plantilla cualquiera, usando la etiqueta `server-instance`. A diferencia de las instancias normales, las instancias de plantillas definidas con categoría, deben realizarse fuera de los nodos, es decir, deben ser hijas de el elemento `application`. Si se instancia una de estas plantillas dentro de un nodo concreto (que puede coincidir o no con la categoría indicada en la plantilla), perderá las propiedades de elasticidad y asignación dinámica.

En el listado 5.12 se muestra la definición de una aplicación que contiene una plantilla definida por categoría y una instancia de dicha plantilla.

```
1  <icecloud>
2    <application name="MinimalApplication">
3      <category-server-template id="ServerTemplate"
4        category="great-mem"
5        allocationRule= "min_servers"
6        growingRule="num_servers gt 3"
7        maxInstances = "4">
8        <server id="Server" activation="manual" exe=".">
9        </server>
10     </category-server-template>
11     <server-instance template="ServerTemplate"/>
12   </application>
13 </icecloud>
```

Listado 5.12: Aplicación con `category-server-template`

Las instancias que se definan en la aplicación, se crearán independientemente del atributo `maxInstances`, y no se eliminarán aunque se cumpla la regla de decrecimiento de la plantilla, sólo se eliminaran copias.

Para realizar las copias, IceCloud usará el parámetro reservado `servercount`, de tal modo que el nombre de las réplicas de las instancias será el indicado en la plantilla con dicho parámetro concatenado. El usuario no es el responsable de asignar el valor a este parámetro,

pero debe tener en cuenta que si usa un parámetro con el mismo nombre para otro propósito, será inutilizado.

Para agregar este nuevo elemento al sistema, ha sido necesario añadir nuevos atributos a la clase *GridState* que representen las instancias originales y secundarias, y los mecanismos para eliminar y añadir esas instancias.

También ha sido necesario modificar la clase `ICRegistry` para que gestione el índice de las instancias al añadir la aplicación.

En la clase `Monitor` se han introducido modificaciones para que evalúe el crecimiento o decrecimiento de las plantillas, y en la clase `Scheduler` también se ha establecido el procedimiento para la asignación y eliminación de forma correcta de las instancias.

Pruebas

En esta iteración se han realizado pruebas para comprobar la correcta asignación de las instancias, y para comprobar que escalan correctamente.

Para verificar que la asignación de las instancias originales es correcta, las pruebas realizadas fueron las siguientes:

- No hay nodos que coincidan con una categoría indicada.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:test_not_matching_nodes_for_category` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un `XML`, con una instancia de una plantilla con categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con un sólo nodo que no corresponde a la categoría «great-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	No hay nodos para la categoría (lanza la excepción *NoMatchingNodesForCategory*).

- No hay nodos que cumplan la regla especificada en `allocationRule`.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:test_not_matching_node_in_category_for_rule` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con una instancia de una plantilla con categoría «great-mem», y `allocationRule = "cpu_load_1 gt 5"`. - Un *NodeInfoRetriever* que devuelve una diccionario con un sólo nodo que no corresponde a la categoría «great-mem», con `cpu_load_1 = 5`.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	No hay nodos en la categoría para la regla indicada (lanza la excepción *NoMatchingNodesInCategoryForRule*).

- Se asigna una instancia de una plantilla con categoría al nodo correcto.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py: test_node_assignment_is_right_1_instance` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con una instancia de una plantilla de categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con un nodo que pertenece a la categoría «great-mem», y otro que no pertenece a dicha categoría.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	La instancia ha sido asignado al nodo correcto.

- Se asignan correctamente 2 instancias de una plantilla con categoría.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py: test_node_assignment_is_right_2_instances` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con dos instancias de una plantilla de categoría «great-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con un nodo que pertenece a la categoría «great-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Ambas instancias han sido añadidas al nodo, y tienen diferente valor del parámetro `servercount`.

- Se asignan correctamente 2 instancias, estando una de ellas definida en un nodo concreto.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:` `test_node_assignment_is_right_1_server_1_static` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con una instancia de una plantilla con categoría «great-mem» en un nodo concreto, y otra a nivel de aplicación. - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», y otro que pertenece a la categoría «low-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Ambas instancias están en el nodo que les corresponde y tienen diferente `servercount`.

- Se asignan correctamente 2 instancias de dos plantillas con categorías diferentes.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:` `test_node_assignment_is_right_2_instances_different_node` disponible en el repositorio.

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con una instancia de una plantilla con categoría «great-mem» y otra de una plantilla con categoría «low-mem». - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», y otro que pertenece a la categoría «low-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Ambas instancias están en el nodo que les corresponde.

- Se asignan correctamente dos servidores de la misma categoría, al nodo con menos servidores.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:` `test_node_assignment_is_right_2_servers_same_category_with_rule_min_server` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con dos instancias de plantillas con categoría «great-mem», y ambos con `allocationRule = min_servers`. - Un *NodeInfoRetriever* que devuelve una diccionario con dos nodos que pertenecen a la categoría «great-mem», con ningún servidor en ellos.
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Cada instancia se ha añadido a un nodo diferente.

- Es compatible el uso de instancias y servidores.
  Esta prueba se corresponde con el método `tests/test_instance_assignment.py:` `test_node_assignment_is_right_2_instances_1_server` disponible en el repositorio:

Dados:	- Un ICRegistry con un fichero de definición de categorías. - Una aplicación IceCloud definida mediante un XML, con una instancia de una plantilla con categoría «great-mem» en un nodo concreto, y otra a nivel de aplicación. También con un servidor definido por categoría en un nodo «great-mem» - Un *NodeInfoRetriever* que devuelve una diccionario con un nodo que pertenece a la categoría «great-mem», y otro a la categoría «low-mem».
Cuando:	Se obtiene el descriptor IceCloud de la aplicación, y se añade al ICRegistry.
Entonces:	Tanto las instancias como el servidor se asignan correctamente.

Para comprobar que las instancias crecen y decrecen de manera correcta, se han realizado las siguientes pruebas:

- Comprobación de que una instancia crece según la regla de crecimiento de la plantilla que implementa.
  Esta prueba se corresponde con el método `tests/test_scheduler.py:test_grow_template` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con una plantilla de categoría «great-mem», con criterios de asignación (min_servers) y crecimiento (num_servers gt 3) y una instancia de dicha plantilla - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones de la plantilla (con distinto número de servidores), en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo con menor número de servidores cumple la regla de crecimiento de la plantilla. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	La plantilla se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler ha creado una nueva instancia de la plantilla en el nodo correcto.

- Comprobación de se elimina una instancia cuando se cumple la regla de decrecimiento
  Esta prueba se corresponde con el método `tests/test_scheduler.py:test_decrease_-instance` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con una plantilla con categoría «great-mem», criterios de asignación (min_servers), crecimiento (num_servers gt 4) y decrecimiento (num_servers le 2). - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones de la plantilla (con distinto número de servidores), en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo con menor número de servidores cumple la regla de crecimiento de la plantilla, y en su tercera consulta suministra información de esos nodos con características que cumplen la regla de decrecimiento de la plantilla. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	La instancia se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler crea una nueva instancia de la plantilla en el nodo con menos servidores.
Cuando:	La segunda instancia creada cumple la regla de decrecimiento y se ejecuta el método run() del Monitor por segunda vez.
Entonces:	El Scheduler ha eliminado la segunda instancia.

- Comprobación de que no se eliminan las instancias originales.
  Esta prueba se corresponde con el método `tests/test_scheduler.py:test_decrease_-original_instance` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con una plantilla con categoría «great-mem», criterios de asignación (min_servers), crecimiento (num_servers gt 4) y decrecimiento (num_servers le 2). - Un `NodeInfoRetriever` que suministra en su primera consulta dos nodos que satisfacen las condiciones de la plantilla (con distinto número de servidores), en su segunda consulta suministra la información de esos mismos nodos modificada de modo que el nodo que antes tenía menor número de servidores ahora tiene más servidores que el otro y además cumple la regla de crecimiento de la plantilla, y en su tercera consulta suministra información de esos nodos con características que cumplen la regla de decrecimiento de la plantilla. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación.
Entonces:	La instancia se asigna al nodo con menor número de servidores.
Cuando:	Se ejecuta el método run() del Monitor.
Entonces:	El Scheduler crea una nueva instancia de la plantilla en el nodo con menos servidores.
Cuando:	La instancia original cumple la regla de decrecimiento y se ejecuta el método run() del Monitor por segunda vez.
Entonces:	El Scheduler ha eliminado la segunda instancia.

5.3.7 Iteración 7: Creación de una interfaz de usuario

Tras revisar los resultados de la iteración anterior, con la que quedaron satisfechos todos los requisitos referentes a la asignación dinámica de servidores y elasticidad del grid, se decidió añadir una interfaz de usuario para añadir las aplicaciones y gestionar el crecimiento/decrecimiento de servidores.

Diseño e implementación

La aplicación a desarrollar, llamada `icecloudadmin` se ejecuta por línea de comandos y permite ejecutar las siguientes operaciones:

- Añadir aplicaciones.
- Eliminar aplicaciones.
- Arrancar servidores.

- Parar servidores.
- Duplicar servidores.
- Eliminar réplicas de servidores.
- Instanciar plantillas.
- Eliminar instancias de plantillas.

La herramienta `icecloudadmin` actúa como un cliente del objeto remoto ICRegistry, por que se han añadido al ICRegistry los métodos necesarios para llevar a cabo esas operaciones. En el listado 5.13 se puede ver la especificación Slice de la interfaz que ICRegistry implementa.

```
1  interface ICRegistry {
2    void addApplication(IceCloud::ApplicationDescriptor
3            applicationDescriptor) throws
4            IceGrid::AccessDeniedException,
5            IceGrid::DeploymentException,
6            NoMatchingNodesInCategoryForRule,
7            NoMatchingNodesForCategory;

9    void removeApplication(string appName) throws
10           IceGrid::AccessDeniedException,
11           IceGrid::DeploymentException,
12           IceGrid::ApplicationNotExistException;

14   void startServer(string serverName) throws
15           IceGrid::ServerNotExistException,
16           IceGrid::ServerStartException,
17           IceGrid::NodeUnreachableException,
18           IceGrid::DeploymentException;

20   void stopServer(string serverName) throws
21           IceGrid::ServerNotExistException,
22           IceGrid::ServerStopException,
23           IceGrid::NodeUnreachableException,
24           IceGrid::DeploymentException;

26   void growServer(string serverName) throws InvalidServerException;

28   void decreaseServer(string serverName) throws InvalidServerException
         ;

30   void growTemplate(string templateName, string application)
31           throws InvalidTemplateException;

33   void decreaseTemplate(string templateName, string application)
34           throws InvalidTemplateException;
35 };
```

Listado 5.13: SLICE de ICRegistry.

Pruebas

Para verificar el funcionamiento de los requisitos añadidos, se han realizado las siguientes pruebas unitarias.

- Un servidor crece correctamente por orden el usuario.
  Esta prueba se corresponde con el método `tests/test_scale_by_user.py:test_-grow_server` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con un servidor con categoría. - Un `NodeInfoRetriever` que suminista un nodo perteneciente a la categoría del servidor. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación y se duplica el servidor.
Entonces:	El servidor se ha duplicado correctamente

- Un servidor decrece correctamente por orden el usuario.
  Esta prueba se corresponde con el método `tests/test_scale_by_user.py:test_-decrease_server` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con un servidor con categoría. - Un `NodeInfoRetriever` que suminista un nodo perteneciente a la categoría del servidor. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación, se duplica el servidor y se ordena que decrezca.
Entonces:	Sólo se encuentra el servidor orignal.

- Una plantilla se instancia correctamente por orden el usuario.
  Esta prueba se corresponde con el método `tests/test_scale_by_user.py:test_-grow_template` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con una plantilla con categoría y una instancia de la misma. - Un `NodeInfoRetriever` que suminista un nodo perteneciente a la categoría de la plantilla. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación y se solicita otra instancia de la plantilla.
Entonces:	En el sistema hay 2 instancias de la plantilla.

- Se elimina una instancia correctamente por orden el usuario.
  Esta prueba se corresponde con el método `tests/test_scale_by_user.py:test_-decrease_instance` disponible en el repositorio:

Dados:	- Una aplicación IceCloud definida mediante un XML, con una plantilla con categoría y una instancia de la misma. - Un `NodeInfoRetriever` que suminista un nodo perteneciente a la categoría de la plantilla. -Un ICRegistry, un Monitor y un Scheduler.
Cuando:	Se añade la aplicación, se solicita otra instancia de la plantilla y luego se solicita la eliminación de una instancia.
Entonces:	En el sistema sólo queda la instancia original.

Prueba de integración

Llegados a este punto, también se ha efectuado una prueba de integración en la que aparecen involucrados todos los elementos del sistema.

Esta prueba se encuentra en el archivo `tests/test_integration.py` disponible en el repositorio:

La prueba se ha realizado sobre un grid simulado con máquinas virtuales en VirtualBox. El grid estaba compuesto por 3 nodos, en los cuales se desplegaba la aplicación IceCloud.

Todas las operaciones de la prueba se han realizado sobre una aplicación con una plantilla definida por categoría, una instancia de la misma, un servidor definido también por categoría y un servidor IcePatch alojado en un nodo conocido (donde se encuentran los archivos de la aplicación).

Mediante esta prueba se ha comprobado:

- El correcto funcionamiento de las operaciones de *icecloudadmin*.
- Que las aplicaciones se añaden correctamente a los nodos reales.
- Que se realiza correctamente la distribución de las aplicaciones.
- Que los servidores crecen, solicitan su distribución, y se ejecutan correctamente en nodos reales.
- Que las instancias crecen, solicitan su distribución, y se ejecutan correctamente en nodos reales.
- Que se lanzan las excepciones correctas.
- Que las aplicaciones se eliminan correctamente del grid.

5.3.8 Iteración 8: Creación de una aplicación de uso real

En esta última iteración, se ha procedido a crear una aplicación distribuida para el renderizado en Blender, que pudiera servir como ejemplo de ejecución de una aplicación con utilidad real en una plataforma de Cloud Computing.

Diseño e implementación

La aplicación consta de dos elementos principales que se ejecutan como servidores Ice (recuerde que los servidores Ice pueden tomar el rol de cliente o servidor independientemente de su nomenclatura):

- **Deliverer**: Este servidor implementa una interfaz que respalda a un objeto Deliverer (ver Slice en el listado 5.14). Este servidor se encarga de llevar una cuenta de los frames renderizados.
- **Renderer**: Este servidor se encarga de solicitar frames y renderizarlos.

```
1  module BlenderApp{
2    sequence<byte> ByteSeq;
3    interface Deliverer
4    {
5      void write(string name, int offset, ByteSeq bytes, int
         numberOfFrame);
6      void endedFrame(int numberOfFrame);
7      int getFrameToRender();
8    };
9  };
```

Listado 5.14: SLICE de la aplicación de renderizado.

En la figura 5.15 podemos ver el funcionamiento de estos servidores. Los *Renderer* solicitan frames al `Deliverer`. El *Deliverer* envía el número de frame a renderizar, y el *Renderer* lo renderiza mediante Blender. Cuando ha terminado el proceso de renderizado, se envía el frame renderizado al *Deliverer*, que guardará el archivo y marcará ese frame como renderizado.

Esta aplicación puede beneficiarse de las ventajas de elasticidad de IceCloud, ya que se podrían ejecutar varias copias de los servidores *Renderer* y renderizar en paralelo varios frames, adaptando la cantidad de copias a las fluctuaciones de recursos del sistema.

Pruebas

Para probar esta aplicación, se ha buscado una animación adecuada para el problema en BlendSwap[1] [BLS].

[1] BlendSwap es una página para el intercambio, colaboración y aprendizaje entre la comunidad de diseñadores 3D. [BLS]

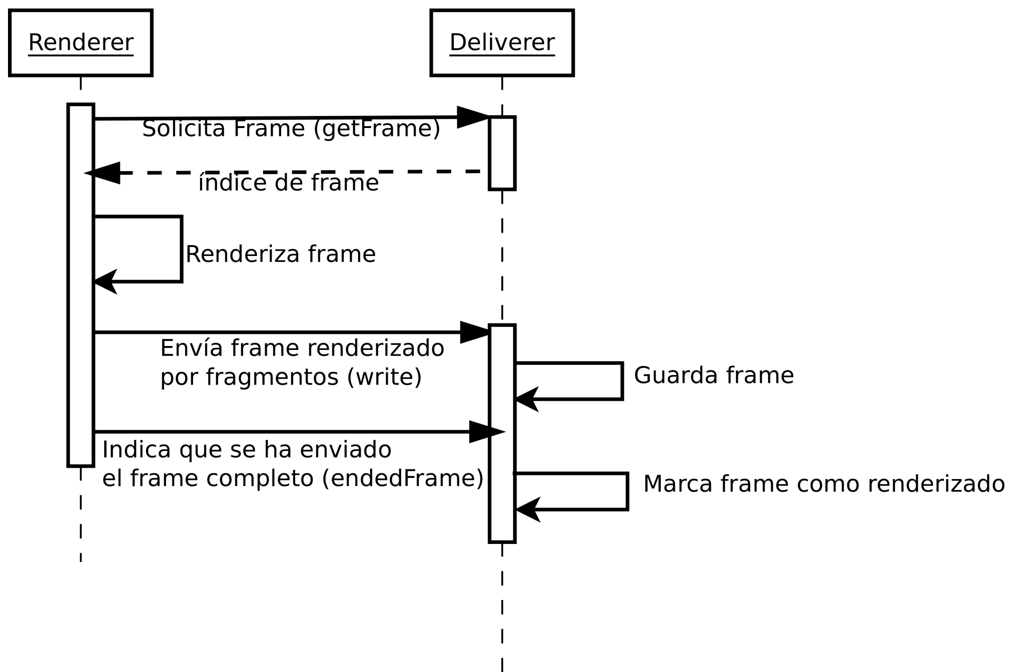

Figura 5.15: Diagrama de secuencia del renderizado.

La animación elegida ha sido «Sintel Walkcycle» [2], realizada por el usuario «Shader» bajo licencia Creative Commons Zero Mark.

Se ha ejecutado la aplicación en IceCloud con resultado satisfactorio en un grid con tres nodos. Para consultar una descripción más detallada de la configuración de la aplicación y de los resultados de ejecutar esta aplicación en el grid con IceCloud, véase el capítulo 6.

[2]http://www.blendswap.com/blends/view/14269

Resultados

Em esta sección se muestra un ejemplo de aplicación de la herramienta desarrollada. Además, se presenta un análisis del esfuerzo de desarrollo del proyecto y una estimación del coste que ha supuesto la realización del mismo.

6.1 Aplicación de la herramienta

Para comprobar el funcionamiento de IceCloud con una aplicación real, se ha implementado una aplicación destinada a renderizar frames de una animación con Blender, creada en la iteración 8 (ver sección 5.3.8) del proceso de desarrollo. El funcionamiento de esta aplicación se basa en la existencia de una serie de servidores *Renderer* que solicitan frames a un servidor *Deliverer* y le envían los resultados tras renderizarlos (ver figura 6.1).

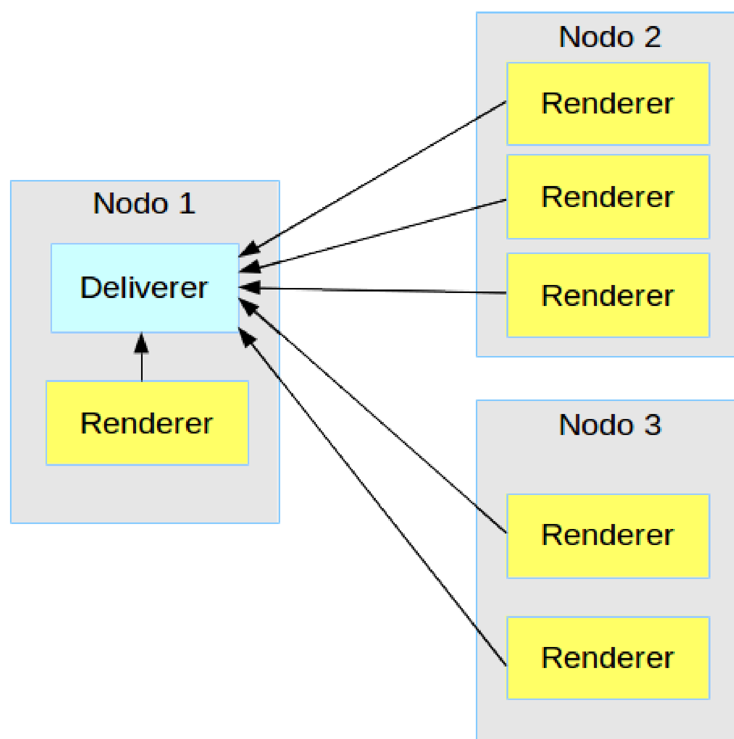

Figura 6.1: Estructura de la aplicación de renderizado

La prueba ha sido realizada en un grid compuesto por tres nodos (*grid-node0*, *grid-node1* y *grid-node2*) con las siguientes características de memoria RAM:

- **grid-node0**: 7762 MB
- **grid-node1**: 4000 MB
- **grid-node2**: 2000 MB

El nodo *grid-node0* ha sido alojado en el portátil descrito en la sección 4.2.2, y el resto han sido alojados en máquinas virtuales ejecutadas en ese mismo portátil. Dadas las limitaciones que esto conlleva en lo referente a la fluctuación del uso de los recursos de las máquinas (como, por ejemplo, la dependencia del uso CPU entre máquinas virtuales), los servidores de la aplicación se han definidos en términos de la memoria RAM de los nodos, por lo que el resto de características son irrelevantes.

Según el fichero de categorías suministrado, todos los nodos pertenecen a la categoría *any-server*. La aplicación, que se utilizará en este ejemplo para renderizar una animación con 320 frames, cuenta con los siguientes elementos:

- Un servidor IcePatch alojado en *grid-node0*, ya que este nodo se encuentra en misma máquina donde está el código fuente de la aplicación.
- Un servidor *Deliverer* alojado también en *grid-node0*.
- Un servidor *Renderer* alojado en un servidor de categoría `any-server`, con `allocationRule = "max_free_ram"`, `growingRule = "free_ram lt 4000"`, `decreasingRule = "free_ram gt 4000"` y `maxDupes = "8"`. Es decir, un servidor que se alojará en el nodo con la máxima memoria RAM libre, crecerá cuando la memoria libre sea menor de 4000 MB hasta un máximo de 8 copias y decrecerá cuando sea mayor de 4000 MB.

En el cuadro 6.1 se muestra la evolución de los servidores al ejecutarse la aplicación en función del tiempo respecto a la creación del primer servidor. Y en la figura 6.2 se muestran varias capturas del estado del grid en diferentes instantes de la ejecución de la aplicación.

Tras finalizar la ejecución, los servidores habían renderizado el número de frames que se muestra en el cuadro 6.2. Se aprecia una diferencia considerable entre los servidores alojados en el nodo *grid-node0* y el resto. Esto se debe a que el uso de CPU en el proceso de renderizado es crucial, y el nodo *grid-node0*, a diferencia de los ejecutados en máquinas virtuales, dispone de varios hilos de ejecución. El total de frames renderizados es superior al número de frames de la aplicación, debido a que el *Deliverer* puede haber suministrado frames a un servidor que ya estaban en proceso de renderizado por parte de otros servidores, sin que hubiesen terminado aún.

Instante de creación (mm:ss)	Servidor	Nodo
00:00	Renderer0	grid-node1
00:29,143	Renderer0.0	grid-node1
01:22,639	Renderer0.1	grid-node1
02:15,639	Renderer0.2	grid-node0
03:12,675	Renderer0.3	grid-node0
04:13,069	Renderer0.4	grid-node1
05:17,307	Renderer0.5	grid-node1
06:29,211	Renderer0.6	grid-node1
07:44,785	Renderer0.7	grid-node1

Cuadro 6.1: Evolución de los servidores

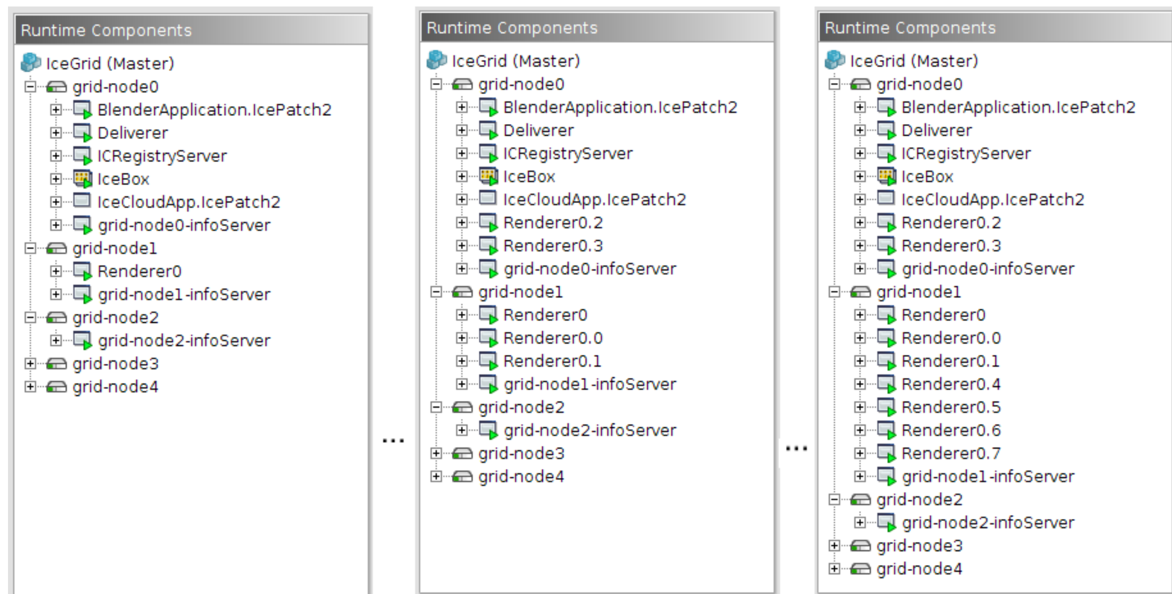

Figura 6.2: Evolución de los servidores durante la ejecución

Servidor	Total frames renderizados
Renderer0	14
Renderer0.0	13
Renderer0.1	12
Renderer0.2	128
Renderer0.3	124
Renderer0.4	10
Renderer0.5	10
Renderer0.6	9
Renderer0.7	9

Cuadro 6.2: Número total de frames renderizados por servidor

6.2 Costes y recursos

A continuación se muestra un resumen de las líneas de código desarrolladas para los lenguajes utilizados (ver cuadro 6.3).

Lenguaje	Nº líneas
Python	5020
XML	380
Slice	506
Script Shell	9
Total	5915

Cuadro 6.3: Total líneas de código por lenguaje

En el listado 6.1 se muestra una estimación del esfuerzo realizado en el desarrollo de este proyecto. Para obtener estos datos, se ha utilizado la herramienta `sloccount`, que analiza los ficheros de un directorio dado, calcula el tamaño del proyecto en líneas de código y emplea el modelo COnstructive COst MOdel (COCOMO) para obtener la estimación del tiempo y los costes necesarios para desarrollar el sistema. Dicha herramienta no tiene en cuenta los ficheros Slice. Tampoco se ha valorado la elaboración de esta memoria.

```
Total Physical Source Lines of Code (SLOC)                = 5,409
Development Effort Estimate, Person-Years (Person-Months) = 1.18
   (14.12)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.57
   (6.84)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 2.07
Total Estimated Cost to Develop                           = \$ 159,006
 (average salary = \$56,286/year, overhead = 2.40).
```

Listado 6.1: Estimación de costes del proyecto

6.3 Repositorio

El código correspondiente al desarrollo del proyecto IceCloud y la documentación de esta memoria están disponibles en un repositorio alojado en *BitBucket*[1], que puede ser descargado con la siguiente orden:

```
hg clone https://bitbucket.org/arco_group/pfc.icecloud
```

[1] https://bitbucket.org/

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones extraídas tras la elaboración del proyecto y los objetivos que se han alcanzado con él. También se describen una serie de propuestas como trabajo futuro para completar los resultados obtenidos en este Proyecto Fin de Carrera (PFC).

7.1 Conclusiones y objetivos alcanzados

Tras finalizar el desarrollo del trabajo propuesto, se puede concluir que se han alcanzado satisfactoriamente los objetivos planteados en un principio.

La aplicación desarrollada en el proyecto proporciona una serie de herramientas para el seguimiento y categorización de los nodos de un grid heterogéneo, y permite especificar aplicaciones en función de esa categorización, abstrayendo el desarrollo de la aplicación (si se desea) de la arquitectura real del grid.

También se ha conseguido dotar al grid de **asignación dinámica**, mediante la cual ejecutar la aplicación en determinados nodos según los recursos disponibles en un momento determinado, sin necesidad de interacción con el desarrollador o administrador de la aplicación. Además, se ha proporcionado al proyecto desarrollado la capacidad de **elasticidad**, esencial en los sistemas de Cloud Computing. Con esta elasticidad, las aplicaciones son automáticamente escalables según las necesidades de sus servidores.

Una de las partes más complejas ha sido el estudio a fondo de **IceGrid** y de sus estructuras internas de comunicación y especificación de aplicaciones. Tras comprender este servicio del framework ZeroC ICE, se han adquirido los conocimientos necesarios para gestionar el despliegue y actualización de las aplicaciones mediante los recursos ofrecidos por su API. También se han comprendido las estructuras necesarias para especificar todos los elementos que pueden formar parte de un grid, lo que ha permitido especificar aplicaciones sin perder ninguna funcionalidad de las ofrecidas por el framework.

En cuanto a las pruebas realizadas, han supuesto un peso importante dentro de la elaboración del proyecto, y han permitido depurar los resultados y otorgar robustez a la implementación. Ha sido muy interesante y útil el uso de *dobles de prueba* para llevar a cabo las mismas.

Por último, se ha creado una aplicación sencilla de renderizado que ha permitido demostrar la utilidad real de un servicio que permite abstraer las aplicaciones de los nodos donde se ejecutan sin perder la posibilidad de elegir las características de los mismos, y los beneficios de la elasticidad del sistema.

7.2 Trabajo futuro

A continuación se detallan algunas propuestas que podrían mejorar la eficiencia de IceCloud y completar sus funcionalidades:

- **Reasignación de servidores y valoración de consecuencias:** Durante el desarrollo del proyecto, al plantear la opción de elasticidad surgió la siguiente cuestión: «¿al asignar un nuevo servidor, deberían tenerse en cuenta las posibles consecuencias de esa asignación en el estado del nodo?». Esta propuesta presenta un problema complejo de tipo NP-completo, que implicaría emplear técnicas de programación dinámica o Inteligencia Artificial y estimaciones para que la asignación de nuevos nodos sea cercana a la óptima y pudiera permitir reasignar todos los nodos del grid para el mejor aprovechamiento posible de los recursos desde el punto de vista de las exigencias de la aplicación. Esta mejora sería muy interesante, pero su desarrollo no es trivial, ya que los recursos referidos miden distintas magnitudes, y la migración o asignación de cualquier servidor influye en el estado del grid.
- **Interfaz gráfica**: Para facilitar el uso de IceCloud, sería interesante diseñar una interfaz gráfica que permitiera añadir y gestionar las aplicaciones. También podría incluir una interfaz para el seguimiento de las mismas y la creación de las aplicaciones, similar a las que ofrece IceGrid Admin.
- **Persistencia**: En la aplicación actual, toda la información sobre las aplicaciones en el sistema se pierde cuando se desactiva el servidor ICRegistry. Una mejora significante sería establecer mecanismos de persistencia para almacenar el estado de las aplicaciones añadidas.

ANEXOS

Referencia IceGrid XML

En este anexo se muestra la guía de referencias XML para la definición de aplicaciones según el manual de ZeroC ICE [Zerb].

A.1 Adapter Descriptor Element

Un elemento `adapter` define un adaptador de objetos.

Este elemento debe ser hijo de un elemento `server` o un elemento `service`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`endpoints`	Especifica uno o más endpoints para el adaptador. Estos endpoints normalmente no especifican un puerto.	No
`id`	Especifica el identificador del adapador. Debe ser único entre todos los adaptadores y grupos de réplica en el registry. Si no se especifica se construirá uno en base al nombre del adaptador y el nombre del servidor.	Sí
`name`	El nombre del adaptador de objetos dado por el servidor que lo crea.	Sí
`priority`	Un valor entero que otorga una prioridad al adaptador. Esta prioridad se usa por parte de los grupos de réplica con balanceo de carga para determinar qué adaptador servirá las peticiones. Si no se define, su valor será 0.	No
`proxy-options`	Las opciones de proxy se utilizan al generar los proxies de los objetos bien conocidos.	No
`register-process`	Este atributo sólo es válido para servidores que usan versiones de ICE inferiores a la 3.3. Si es `true`, registrará un objeto que facilita el apagado del servidor.	No
`replica-group`	Especifica un identificador de un grupo de réplica.	No

Atributo	Descripción	Requerido
`server-lifetime`	Si es `true` indica que el adaptador estará activo siempre que lo esté su servidor. Un nodo IceGrid considerará un servidor activado si todos sus adaptadores con este atributo verdadero están registrados en el registry. Cuando uno de los adaptadores con este atributo verdadero no está registrado en el registry, el servidor se desactiva.	No

Es posible añadir una descripción en forma de texto con el elemento anidado texttdescription.

El listado A.1 muestra un ejemplo de uso de este elemento:

```
1 <adapter name="MyAdapter"
2     endpoints="default"
3     id="MyAdapterId"
4     proxy-options="-t -e 1.0"
5     replica-group="MyReplicaGroup">
6     <description>A description of the adapter.</description>
7     ...
8 </adapter>
```

Listado A.1: Ejemplo de uso del elemento `adapter`

A.2 Allocatable Descriptor Element

Define un objeto «asignable» (los clientes pueden solicitarlos para su uso exclusivo).

Solamente puede aparecer como hijo de un elemento `adapter`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`identity`	Especifica la identida por la que es conocido el objeto	Sí
`property`	Especifica el nombre de una propiedad que contendrá la identidad textual del objeto	No
`type`	Una cadena arbitraria para agrupar los objetos asignables	No
`proxy-options`	Las opciones del proxy.	No

A.3 Application Descriptor Element

Un elemento `aplication` define una aplicación IceGrid. Normamente, incluye al menos un elemento `node`, pero puede usarse para definir plantillas, grupos de réplica y conjuntos de propiedades.

Este elemento debe ser hijo de un elemento `icegrid` y sólo se permite uno por archivo.

Soporta los siguientes atributos:

Atributo	**Descripción**	**Requerido**
`import-default-templates`	Si es `true`, se importarán y podrán usarse las plantillas por defecto configuradas para el registry. Por defecto, su valor es `false`.	No
`name`	El nombre de la aplicación. Debe ser único en el registry. Los elementos hilos pueden referirse a este nombre mediante la variable `${application}`.	Sí

Se le puede anidar un elemento `description` para especificar una descripción textual de la aplicación.

En el listado A.2 se ejemplifica el uso de este elemento.

```
1 <icegrid>
2     <application name="MyApplication" import-default-templates="true">
3         <description>A description of the application.</description>
4         ...
5     </application>
6 </icegrid>
```

Listado A.2: Ejemplo de uso del elemento `application`

A.4 DbEnv Descriptor Element

Un elemento `dbenv` hace que el nodo genere propiedades de configuración propias de Freeze para el servidor o servicio en el que está definido, lo que puede llevar a crear un directorio para la base de datos. Este elemento debe contener elementos de tipo `dbproperty`.

Este elemento sólo puede aparecer como hijo de un elemento `server` o `service`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`home`	El directorio donde ubicar el entorno de la base de datos. Si no se especifica, se usará el directorio en el que se encuentran los datos del nodo.	No
`name`	El nombre del entorno de la base de datos .	Sí

Los atributos se usan para definir la propiedad Freeze.DbEnv.*env-name*.DbHome.

Es posible anidar un elemento de tipo `description` para describir este elemento en forma de texto.

En el listado A.3 se ejemplifica el uso de este elemento.

```
1 <dbenv name="MyEnv" home="/opt/data/db">
2     <description>A description of the database environment.</
          description>
3     ...
4 </dbenv>
```

Listado A.3: Ejemplo de uso del elemento `dbenv`

A.5 DbProperty Descriptor Element

Un elemento `dbproperty` define una propiedad de configuración para un entorno de base de datos Freeze.

Sólo debe aparecer como hijo de un elemento `dbenv`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`name`	Nombre de la propiedad	Sí
`value`	El valor de la propiedad. Si no se define, será una cadena vacía.	Sí

En el listado A.4 se muestra un ejemplo de uso:

```
1 <dbenv name="MyEnv" home="/opt/data/db">
2     <description>A description of the database environment.</
          description>
3     <dbproperty name="set_cachesize" value="0 52428800 1"/>
4 </dbenv>
```

Listado A.4: Ejemplo de uso del elemento `dbproperty`

A.6 Description Descriptor Element

Un elemento description ofrece una descripción textual de su padre.

Sólo debe aparecer como hijo de los elementos application, replica-group, node, server, service, icebox adapter y dbenv.

El listado A.5 se muestra un ejemplo de uso:

```
1 <node name="localnode">
2     <description>Free form descriptive text for localnode</description
        >
3 </node>
```

Listado A.5: Ejemplo de uso del elemento description

A.7 Directory Descriptor Element

Un elemento directory especifica un directorio de una distribución.

Este elemento sólo aparece como hijo de un elemento distrib.

No soporta atributos.

A.8 Distrib Descriptor Element

Un elemento distrib especifica los archivos de distribución que los clientes de un servidor IcePatch2 deben descargarse, así como el proxy de ese servidor.

Este elemento sólo debe aparecer como hijo de los elementos application y server.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
icepatch	Proxy del servidor IcePatch2. Si no se define, será ${application}.IcePatch2/server.	No

En el listado A.6 se muestra el uso de este elemento.

```
1 <distrib icepatch="DemoIcePatch2/server:tcp -p 12345">
2     <directory>dir1</directory>
3     <directory>dir2/subdir</directory>
4 </distrib>
```

Listado A.6: Ejemplo de uso del elemento distrib

A.9 IceBox Descriptor Element

Un elemento `icebox` define un servidor IceBox. Por lo general, suele contener al menos un elemento de tipo `service`, y puede especificar información como opciones de la linea de comandos, variables de entorno, propiedades y distribución.
El adaptador de objetos que gestiona los servicios, `IceBox.ServiceManager`, sólo puede configurarse mediante propiedades.

Este elemento sólo aparece como hijo de un elemento `node` o `server-template`.

Soporta los mismos atributos que un elemento `server`.

Puede anidarse un elemento de tipo `description` para añadir una descripción del mismo.

El listado A.7 muestra un ejemplo de su uso.

```
1  <icebox id="MyIceBox"
2          activation="on-demand"
3          activation-timeout="60"
4          application-distrib="false"
5          deactivation-timeout="60"
6          exe="/opt/Ice/bin/icebox"
7          pwd="/">
8      <option>--Ice.Trace.Network=1</option>
9      <env>PATH=/opt/Ice/bin:\$PATH</env>
10     <property name="IceBox.UseSharedCommunicator.Service1" value="1"/>
11     <service name="Service1" .../>
12     <service-instance template="ServiceTemplate" name="Service2"/>
13 </icebox>
```

Listado A.7: Ejemplo de uso del elemento `icebox`

A.10 IceGrid Descriptor Element

El elemento `icegrid` es el elemento raíz de los descriptores en formato XML. No soporta atributos.

A.11 Load-Balancing Descriptor Element

El elemento `load-balancing` especifica la política de balanceo de carga de un grupo de réplica.

Sólo puede aparecer como hijo de un elemento `replica-group`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
load-sample	Sus valores pueden ser 1, 5 y 15. En una política adaptativa especifica si se toma como referencia la carga en los últimos 1, 5 o 15 minutos, respectivamente. Por defecto, su valor es 1.	No
n-replicas	Especifica el número máximo de réplicas para calcular el número de endpoints. Por defecto, es 1.	No
type	El tipo de política de balanceo de carga. Sus valores pueden ser adaptive, ordered, round-robin y random.	No

En el listado A.8 se muestra el uso de este elemento.

```
1 <application name="MyApp">
2     <replica-group id="ReplicatedAdapter">
3         <load-balancing type="adaptive" load-sample="15" n-replicas="3
              "/>
4         <description>A description of this replica group.</description
              >
5         <object identity="WellKnownObject" .../>
6     </replica-group>
7     ...
8 </application>
```

Listado A.8: Ejemplo de uso del elemento load-balancing

A.12 Log Descriptor Element

Un elemento log especifica el nombre de un fichero de log de un servidor o servicio. Se debe especificar un elemento log para cada archivo de log que pueda ser accesible remotamente con una herramienta administrativa, excepto para los de las propiedades Ice.StdErr y Ice.StdOut.

Este elemento sólo puede aparecer como hijo de un elemento server o service.

En el listado A.9 se muestra un ejemplo de uso de este elemento:

```
1 <server id="MyServer" ...>
2     <log path="\${server}.log" property="LogFile"/>
3 </server>
```

Listado A.9: Ejemplo de uso del elemento log

A.13 Node Descriptor Element

Un elemento node define un nodo IceGrid. Los servidores alojados en el nodo se describen con hijos.

Este elemento sólo aparece como hijo de un elemento `application`. Pueden aparecer varios nodos con el mismo nombre, y su contenido se mezclará, prevaleciendo la última definición de balanceo de carga.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`load-factor`	Un valor en punto flotante que se multiplica por la carga de trabajo usada por la políta de balanceo de carga. Por defecto, es 1 en plataformas Unix y 1 o el número de procesadores en Windows.	No
`name`	El nombre del nodo. Sus elementos hijos pueden referirse a él con la variable `{$node}`. Debe haber un icegridnode con el mismo nombre ejecutándose en el grid.	Sí

En el listado A.10 se muestra un ejemplo de uso de este elemento:

```
<node name="Node1" load-factor="2.0">
    <description>A description of this node.</description>
    <server id="Server1" ...>
        <property name="NodeName" value="\${node}"/>
        ...
    </server>
</node>
```

Listado A.10: Ejemplo de uso del elemento `node`

A.14 Object Descriptor Element

Un elemento `object` define un objeto bien conocido.

Este elemento sólo aparece como hijo de un elemento `adapter` o `replica-group`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`identity`	Especifica la identidad por la que es conocido el objeto.	Sí
`property`	Especifica el nombre de una propiedad a generar que contendrá la identidad textual del objeto. Sólo se permite si es hijo de un elemento `adapter`.	No
`type`	Una cadena usada para agrupar objetos.	No
`proxy-options`	Las opciones de uso del proxy del objeto devueltas por IceGrid.	No

El listado A.11 muestra el uso de este elemento.

```
1 <adapter name="MyAdapter" id="WellKnownAdapter" ...>
2     <object identity="WellKnownObject" type="
          ::Module::WellKnownInterface" proxy-options="-o"/>
3 </adapter>
```

Listado A.11: Ejemplo de uso del elemento object

A.15 Parameter Descriptor Element

Un elemento parameter define un parámetro de una plantilla.

Este elemento sólo aparece como hijo de los elementos server-template y service-template.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
name	El nombre del parámetro. Si se usa, por ejemplo index como el nombre, se podrá hacer referencia al parámetro usando ${index}.	Sí
default	Un valor por defecto para el parámetro, que será usado si el servidor o servicio no instancian el parámetro.	No

El listado A.12 muestra el uso de este elemento.

```
1 <server-template id="MyServerTemplate">
2     <parameter name="index"/>
3     <parameter name="exepath" default="/opt/myapp/bin/server"/>
4     ...
5 </server-template>
```

Listado A.12: Ejemplo de uso del elemento parameter

A.16 Properties Descriptor Element

Un elemento properties puede usarse en tres casos diferentes:

- Como una un conjunto de propiedades con nombre, si se concreta su atributo id. En este caso, el elemento debe ser hijo de un elemento application o node.
- Como referencia a un conjunto de propiedades, si se concreta su atributo refid. En este caso, el elemento debe ser hijo de otro elemento properties.

- Como un conjunto sin nombre de propiedades si no se incluyen los atributos anteriores. En este caso, el elemento será hijo de un elemento `server-instance`.

  Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`id`	El nombre del conjunto de propiedades. Debe ser único entre los conjuntos de propiedades con el mismo alcance.	No
`refid`	El nombre del conjunto de propiedades al que este elemento hace referencia.	No
`service`	Especifica el nombre de un servicio IceBox. El elemento ampliará las propiedades del servicio	No

El listado A.13 muestra el uso de este elemento.

```
1  <application name="Simple">
2     <properties id="Debug">
3          <property name="Ice.Trace.Network" value="1"/>
4     </properties>

6      <server id="MyServer" exe="./server">
7          <properties>
8              <properties refid="Debug"/>
9              <property name="AppProperty" value="1"/>
10         </properties>
11     </server>
12 </application>
```

Listado A.13: Ejemplo de uso del elemento `properties`

A.17 Property Descriptor Element

Un elemento `property` define propiedades adicionales a las que un nodo genera para sus servidores y servicios.
Este elemento sólo puede aparecer como hijo de un elemento `server`, `service`, `icebox` o `properties`
Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`name`	El nombre de la propiedad.	Sí
`value`	El valor de la propiedad.	No

El listado A.14 muestra el uso de este elemento.

```
<server id="MyServer" ...>
    <property name="Ice.ThreadPool.Server.SizeMax" value="10"/>
    ...
</server>
```

Listado A.14: Ejemplo de uso del elemento property

A.18 Replica-Group Descriptor Element

.

Un elemento replica-group crea un adaptador de objetos virtual para dotar a un conjunto de a adaptadores de objetos de capacidades de replicación y balanceo de carga. Debe declarar todos los objetos bien conocidos registrados en los adaptadores que agrupa y puede contener un hijo de tipo load-balancing para especificar la política de balanceo de carga. Si no se especifica, la política será de tipo random.
Este elemento debe aparecer como hijo de un elemento application.
Soporta los siguientes atributos:

Atributo	Descripción	Requerido
id	El identificador del grupo de réplica, que debe ser único en todo el registry.	Sí
proxy-optons	Opciones de proxy a usar para los proxies de los objetos bien conocidos del grupo de réplica.	No

El listado A.15 muestra el uso de este elemento.

```
<application name="MyApp">
    <replica-group id="ReplicatedAdapter" proxy-options="-e 1.0">
        <load-balancing type="adaptive" load-sample="15" n-replicas="3
            "/>
        <description>A description of this replica group.</description
            >
        <object identity="WellKnownObject" .../>
    </replica-group>
    ...
</application>
```

Listado A.15: Ejemplo de uso del elemento replica-group

A.19 Server Descriptor Element

Un elemento de tipo server define a un servidor que se desplegará en un nodo.
Sólo puede aparecer como hijo de un elemento nodo o server-template.
Soporta los siguientes atributos:

Atributo	Descripción	Requerido
activation	Especifica el modo de activación del servidor. Puede ser manual, on-demandalways o session.	No
activation-timeout	Especifica los segundos que un nodo esperará para que el servidor arranque. Si sobrepasa el tiempo, no será posible acceder a él. Por defecto, es el contenido de la propiedad IceGrid.Node.WaitTime del nodo.	No
allocatable	Especifica si el servidor es «asignable».	No
application-distrib	Especifica si la distribución del servidor es dependiente de la distribución de la aplciación. Por defecto su valor es true.	No
deactivation-timeout	Los segundos que un nodo esperará a que el servidor se desactive. Si se excede este tiempo, el nodo matará todos los procesos relacionados con el servidor. Por defecto, su valor es el de la propiedad IceGrid.Node.WaitTime del nodo.	No
exe	La ruta al ejecutable del servidor.	Sí
ice-version	La versión de Ice usada por el servidor. Si no se especifica, será la misma que use IceGrid.	No
id	Identificador del servidor. Debe ser único en todo el registry. Sus elementos hijos pueden referirse a ella con la variable ${server}.	Sí
pwd	El directorio de trabajo del servidor.	No
user	El nombre del usuario bajo el que debe ejecutarse el servidor.	No

En el listado A.16 se puede observar un ejemplo de uso de este elemento.

A.20 Server-Instance Descriptor Element

Un elemento server-instance despliega una instancia de una plantilla de servidor en un nodo.

Sólo puede aparecer como hijo de un elemento node.

Soporta un atributo:

Atributo	Descripción	Requerido
template	La plantilla que instancia.	Sí

```
<server id="MyServer"
        activation="on-demand"
        activation-timeout="60"
        application-distrib="false"
        deactivation-timeout="60"
        exe="/opt/app/bin/myserver"
        pwd="/">
    <option>--Ice.Trace.Network=1</option>
    <env>PATH=/opt/Ice/bin:$PATH</env>
    <property name="ServerId" value="${server}"/>
    <adapter name="Adapter1" .../>
</server>
```

Listado A.16: Ejemplo de uso del elemento `server`

En el listado A.17 se puede observar un ejemplo de uso de este elemento.

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate" id="TheServer"/
                >
        </node>
    </application>
</icegrid>
```

Listado A.17: Ejemplo de uso del elemento `server-instance`

A.21 Server-Template Descriptor Element

Un elemento `server-template` define una plantilla para un servidor, de modo que simplifica la tarea de desplegar varias instancias de un mismo servidor.

Este elemento debe aparecer como hijo de un elemento `application`.

Soporta un atributo:

Atributo	Descripción	Requerido
`id`	El identificador de la plantilla. Debe ser único entre las plantillas de servidores de la aplicación.	Sí

Se pueden declarar parámetros para instanciar los servidores.

En el listado A.18 se puede observar un ejemplo de uso de este elemento.

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate" id="TheServer"/
               >
        </node>
    </application>
</icegrid>
```

Listado A.18: Ejemplo de uso del elemento `server-template`

A.22 Service Descriptor Element

Un elemento `service` define un servicio IceBox.

Solamente puede aparecer como hijo de un elemento `icebox` o `service-template`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`entry`	Especifica el punto de acceso del servidor. Tiene la siguiente forma: `path[,version]:function`.	Sí
`name`	Especifica el nombre del servicio. Los elementos hijos pueden referirse a él mediante la variable `${service}`.	Sí

Se puede anidar un elemento de tipo `descriptor` para añadir una descripción textual del servicio.

El listado A.19 muestra un ejemplo de uso de este elemento.

```
<icebox id="MyIceBox" ...>
    <service name="Service1" entry="service1:Create">
        <description>A description of this service.</description>
        <property name="ServiceName" value="${service}"/>
        <adapter name="MyAdapter" id="${service}Adapter" .../>
    </service>
    <service name="Service2" entry="service2:Create"/>
</icebox>
```

Listado A.19: Ejemplo de uso del elemento `service`

A.23 Service-Instance Descriptor Element

Un elemento `service-instance` crea una instancia de una plantilla de servicio en un servidor IceBox.

Este elemento sólo aparece como hijo de un elemento `icebox`.

Requiere el siguiente atributo:

Atributo	Descripción	Requerido
`template`	La plantilla que instancia.	Sí

En el listado A.20 se muestra un ejemplo de uso de este elemento.

```
<icebox id="IceBoxServer" ...>
    <service-instance template="ServiceTemplate" name="Service1"/>
</icebox>
```

Listado A.20: Ejemplo de uso del elemento `service-instance`

A.24 Service-Template Descriptor Element

Un elemento `service-template` sirve para definir una plantilla para un servicio, simplificando la tarea de desplegar varias instancias de un mismo servicio. Las plantillas deberían contener un elemento `service` parametrizado, que se instanciará mediante un elemento `service-instance`.

Este elemento solamente puede aparecer como hijo de un elemento `application`.

Requiere un atributo:

Atributo	Descripción	Requerido
`id`	Identifica la plantilla. Debe ser único entre las plantillas de servicios de la aplicación.	Sí

En el listado A.21 podemos ver un ejemplo de uso de este elemento.

```
<icegrid>
  <application name="IceBoxApp">
    <service-template id="ServiceTemplate">
      <parameter name="name"/>
      <service name="${name}" entry="DemoService:create">
        <adapter name="${service}" .../>
      </service>
    </service-template>
    <node name="Node1">
      <icebox id="IceBoxServer" ...>
        <service-instance template="ServiceTemplate" name="Service1"/>
      </icebox>
    </node>
  </application>
</icegrid>
```

Listado A.21: Ejemplo de uso del elemento `service-template`

A.25 Variable Descriptor Element

Un elemento `variable` define una variable.

Sólo puede aparecer como hijo de un elemento `application` o un elemento `node`.

Soporta los siguientes atributos:

Atributo	Descripción	Requerido
`name`	El nombre de la variable. Se puede acceder a la variable con la sintaxis siguiente: `${name}`.	Sí
`name`	El valor de la variable. Por defecto, es una cadena vacía.	No

En el listado A.22 podemos ver un ejemplo de uso de este elemento.

```
1 <icegrid>
2     <application name="SampleApp">
3         <variable name="Var1" value="foo"/>
4         <variable name="Var2" value="${Var1}bar"/>
5         ...
6     </application>
7 </icegrid>
```

Listado A.22: Ejemplo de uso del elemento `variable`

Bibliografía

[AS3] Amazon Simple Storage Service (Amazon S3). url: `http://aws.amazon.com/s3`.

[AZU] Microsoft Azure. url: `http://azure.microsoft.com`.

[BLE] Blender. Home of the Blender Project. url: `http://www.blender.org`.

[BLS] BlendSwap. url: `http://mercurial.selenic.com/`.

[BYV+09] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, y I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616, Junio 2009.

[CJKB12] G. Coulouris, J.Dollimore, T. Kindberg, y G. Blair. *Distributed Systems. Concepts and design. 5th Ed.* Addison Wesley, 2012.

[CLM+03] B. Cascales, P. Lucas, J.M. Mira, A.J. Pallarés, y S. Sánchez-Pedreño. *El libro de LaTeX*. Prentice Hall, edición 1ª, 2003.

[COS09] Cloud Computing como una red de servicios. Technical report, Instituto Tecnológico de Costa Rica, 2009.

[EC2] Amazon Elastic Compute Cloud (Amazon EC2). url: `http://aws.amazon.com/ec2/`.

[Fou] The Eclipse Foundation. url: `http://www.eclipse.org/`.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, y S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. En *Grid Computing Environments Workshop*. IEEE, 2008.

[GCP] Google Cloud Platform (GAE). url: `https://cloud.google.com/`.

[Kle69] L. Kleinrock. UCLA to be First Station in Nationwide Computer Network. Press Release, July 1969.

[LSH] HardwareLister. url: `http://ezix.org/project/wiki/HardwareLiSter`.

[LXM] XML and HTML with Python. url: `http://lxml.de/`.

[MER14] Mercurial: Work easier, Work faster, 2014. url: http://mercurial.selenic.com/.

[MFC00] Tim Mackinnon, Steve Freeman, y Philip Craig. Endo-Testing: Unit Testing with Mock Objects. *XP eXamined*, 2000.

[nis11] The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Julio 2011.

[O'S] B. O'Sullivan. *Mercurial: The Definitive Guide*. url: http://hgbook.red-bean.com/read/.

[OST] OpenStack. url: http://www.openstack.org/.

[Pre10] R.S. Pressman. *Ingeniería del Software. Un enfoque práctico. 7ª ed.* Mc Graw Hill, 2010.

[PRP05] A. Puder, K. Römer, y F. Pilhofer. *Distributed Systems Architecture: A Middleware Approach*. Morgan Kaufman, 2005.

[pyd] PyDev. Python IDE for Eclipse. url: http://pydev.org/.

[PZH$^+$] J. Pechta, F. Zenith, Danielsson H., Braun T., Ludwig M., y Mauch F. *The Kile Handbook*. url: http://docs.kde.org/stable/en/extragear-office/kile/kile.pdf.

[Som05] I. Sommerville. *Ingeniería del Software. 7ª ed.* Addison Wesley, 2005.

[Vil] D. Villa. Python-doublex. url: https://bitbucket.org/DavidVilla/python-doublex/wiki/Home.

[Zera] ZeroC. The Internet Communications Engine. url: http://www.zeroc.com.

[Zerb] Inc. ZeroC. *Manual de ZeroC Ice*. url: http://doc.zeroc.com/display/Ice/Ice+Manual.

Este documento fue editado y tipografiado con $\LaTeX$
empleando la clase **arco-pfc** que se puede encontrar en:
`https://bitbucket.org/arco_group/arco-pfc`

[Respeta esta atribución al autor]