

Computación Distribuida con ZeroC Ice



Universidad de Castilla-La Mancha

Escuela Superior de Informática
de Ciudad Real

David Villa, Francisco Moya y Óscar Aceña

2016/09/20

Escuela Superior de Informática

e-mail esi@uclm.es

Teléfono 926 29 53 00

Web <http://www.esi.uclm.es>

© Los autores del documento. Se permite la copia, distribución y/o modificación de este documento bajo los términos de la licencia de documentación libre GNU, versión 1.1 o cualquier versión posterior publicada por la *Free Software Foundation*, sin secciones invariantes. Puede consultar esta licencia en <http://www.gnu.org>.

Índice general

- Índice general III
- Prólogo XI
- 1. **Introducción** 1
 - 1.1. ZeroC Ice 3
 - 1.2. Especificación de interfaces 3
 - 1.2.1. Correspondencia con los lenguajes de implementación . . . 4
 - 1.3. Terminología 5
 - 1.3.1. Clientes y servidores. 5
 - 1.3.2. Objetos 5
 - 1.3.3. Proxies 6
 - 1.3.4. Sirvientes (*servants*). 8
 - 1.4. Semántica *at-most-once* 10
 - 1.5. Métodos de entrega 10
 - 1.5.1. Invocaciones en una sola dirección. 10
 - 1.5.2. Invocaciones por lotes en una sola dirección 11
 - 1.5.3. Invocaciones en modo datagrama 12
 - 1.5.4. Invocaciones en modo datagrama por lotes 12
 - 1.6. Excepciones 12
 - 1.7. Propiedades 13
 - 1.8. El protocolo IceP 13
- 2c. **Hola mundo distribuido (versión C++)** 15
 - 2c.1. Sirviente 16

2c.2.	Servidor	17
2c.3.	Cliente	18
2c.4.	Compilación	19
2c.5.	Ejecutando el servidor	20
2c.6.	Ejecutando el cliente	22
2c.7.	Ejercicios	22
2j.	Hola mundo distribuido (versión Java)	25
2j.1.	Sirviente	26
2j.2.	Servidor	26
2j.3.	Cliente	28
2j.4.	Compilación	29
2j.5.	Ejecutando el servidor	29
2j.6.	Ejecutando el cliente	31
2j.7.	Ejercicios	31
2p.	Hola mundo distribuido (versión Python)	33
2p.1.	Sirviente	34
2p.2.	Servidor	34
2p.3.	Cliente	35
2p.4.	Compilación	36
2p.5.	Ejecutando el servidor	36
2p.6.	Ejecutando el cliente	38
2p.7.	Ejercicios	38
3c.	Gestión de aplicaciones distribuidas (versión C++)	41
3c.1.	Introducción	41
3c.2.	IceGrid	41
3c.2.1.	Componentes de IceGrid	42
3c.2.2.	Configuración de IceGrid	43
3c.2.3.	Arrancando IceGrid	45
3c.3.	Creando una aplicación distribuida	46
3c.4.	Despliegue de aplicaciones con IcePatch2	51
3c.5.	Instanciación del servidor	51
3c.5.1.	Instanciando un segundo servidor	53

3c.6.	Ejecutando la aplicación	54
3c.6.1.	Despliegue de la aplicación	54
3c.6.2.	Ejecutando los servidores	54
3c.6.3.	Ejecutando el cliente	57
3c.7.	Objetos bien conocidos	58
3c.8.	Activación y desactivación implícita.	59
3c.9.	Depuración	59
3c.9.1.	Evitando problemas con IcePatch2	59
3c.9.2.	Descripción de la aplicación	60
3c.10.	Receta	60
3c.11.	Ejercicios	61
3j.	Gestión de aplicaciones distribuidas (versión Java)	63
3j.1.	Introducción	63
3j.2.	IceGrid	63
3j.2.1.	Componentes de IceGrid	64
3j.2.2.	Configuración de IceGrid	65
3j.2.3.	Arrancando IceGrid	67
3j.3.	Creando una aplicación distribuida	68
3j.4.	Despliegue de aplicaciones con IcePatch2.	73
3j.5.	Instanciación del servidor	73
3j.5.1.	Instanciando un segundo servidor	75
3j.6.	Ejecutando la aplicación	76
3j.6.1.	Despliegue de la aplicación	76
3j.6.2.	Ejecutando los servidores	76
3j.6.3.	Ejecutando el cliente	79
3j.7.	Objetos bien conocidos	80
3j.8.	Activación y desactivación implícita.	81
3j.9.	Depuración	81
3j.9.1.	Evitando problemas con IcePatch2	81
3j.9.2.	Descripción de la aplicación	82
3j.10.	Receta	82
3j.11.	Ejercicios	83

3p. Gestión de aplicaciones distribuidas (versión Python).	85
3p.1. Introducción	85
3p.2. IceGrid	85
3p.2.1. Componentes de IceGrid	86
3p.2.2. Configuración de IceGrid	87
3p.2.3. Arrancando IceGrid	89
3p.3. Creando una aplicación distribuida	90
3p.4. Despliegue de aplicaciones con IcePatch2	95
3p.5. Instanciación del servidor	95
3p.5.1. Instanciando un segundo servidor	97
3p.6. Ejecutando la aplicación	98
3p.6.1. Despliegue de la aplicación	98
3p.6.2. Ejecutando los servidores	98
3p.6.3. Ejecutando el cliente	101
3p.7. Objetos bien conocidos	102
3p.8. Activación y desactivación implícita.	103
3p.9. Depuración	103
3p.9.1. Evitando problemas con IcePatch2	103
3p.9.2. Descripción de la aplicación	104
3p.10. Receta	104
3p.11. Ejercicios	105
4c. Propagación de eventos en C++.	107
4c.1. Arranque del servicio	110
4c.2. Federación de canales	111
4c.2.1. Propagación entre canales de eventos federados	112
4j. Propagación de eventos en Java.	115
4j.1. Arranque del servicio	118
4j.2. Federación de canales	119
4j.2.1. Propagación entre canales de eventos federados	120

4p. Propagación de eventos en Python	123
4p.1. Arranque del servicio	125
4p.2. Federación de canales	127
4p.2.1. Propagación entre canales de eventos federados	128
5c. Invocación y despacho asíncrono en C++	131
5c.1. Invocación síncrona de métodos	131
5c.2. Invocación asíncrona de métodos	132
5c.2.1. Proxies asíncronos	132
5c.2.2. Utilizando un objeto <i>callback</i>	134
5c.3. Despachado asíncrono de métodos	136
5c.4. Mecanismos desacoplados	138
5c.5. Ejercicios	139
5j. Invocación y despacho asíncrono en Java	141
5j.1. Invocación síncrona de métodos	141
5j.2. Invocación asíncrona de métodos	142
5j.2.1. Proxies asíncronos	142
5j.2.2. Utilizando un objeto <i>callback</i>	144
5j.3. Despachado asíncrono de métodos	145
5j.4. Mecanismos desacoplados	148
5j.5. Ejercicios	148
5p. Invocación y despacho asíncrono en Python	151
5p.1. Invocación síncrona de métodos	151
5p.2. Invocación asíncrona de métodos	152
5p.2.1. Proxies asíncronos	152
5p.2.2. Utilizando un objeto <i>callback</i>	154
5p.3. Despachado asíncrono de métodos	155
5p.4. Mecanismos desacoplados	158
5p.5. Ejercicios	159

6p. **Patrones (versión Python)** 161

 6p.1. Contenedor de objetos 161

 6p.2. Factoría de objetos. 164

Referencias 167

Listado de acrónimos

AMI	Asynchronous Method Invocation
AMD	Asynchronous Method Dispatching
API	Application Program Interface
ARCO	Arquitectura y Redes de Computadores
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
DNS	Domain Name System
DoS	Denial of Service
EJB	Enterprise JavaBeans
GIOP	General IOP
Ice	Internet Communications Engine
IceP	ICE Protocol
IDL	Interface Definition Language
IP	Internet Protocol
IPC	Inter-Process Communication
JMS	Java Message Service
NIC	Network Interface Controller
POSIX	Portable Operating System Interface; UNIX
RMI	Remote Method Invocation
RPC	Remote Procedure Call
Slice	Specification Language for Ice
SSL	Secure Socket Layer
STL	C++ Standard Template Library
TCP	Transmission Control Protocol
TCP/IP	Arquitectura de protocolos de Internet
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
XDR	eXternal Data Representation
XML	Extensible Markup Language

Prólogo

El presente documento es una introducción muy práctica al desarrollo de aplicaciones distribuidas mediante middlewares orientados a objeto. En concreto se utiliza el middleware ICE (Internet Communications Engine) de la empresa ZeroC, Inc. Se trata de un middleware de propósito general, que a pesar de estar diseñado por una empresa, se ciñe en gran medida a los estándares de la industria. Maneja básicamente los mismos conceptos y abstracciones del estándar CORBA (Common Object Request Broker Architecture), aunque es más sencillo, potente y flexible que la mayoría de las implementaciones disponibles para éste último.

Todas las técnicas y mecanismos explicados se complementan con ejemplos que utilizan este middleware. Son ejemplos de código intencionadamente simples, pero completamente funcionales. Por ello resultan perfectos para diseñar y desarrollar aplicaciones más complejas tomándolos como punto de partida.

Tanto el código como la documentación ha sido desarrollada por los miembros del grupo ARCO (Arquitectura y Redes de Computadores)¹. Es el resultado de la gran experiencia del grupo derivada tanto de docencia universitaria en redes de computadores y sistemas distribuidos como de la participación en grandes proyectos de investigación y desarrollo tanto públicos como privados a lo largo de más de una década.

Sobre los ejemplos

Todos los ejemplos que aparecen en este documento (y algunos otros) están disponibles para descarga a través del repositorio mercurial en:

https://bitbucket.org/arco_group/ice-hello

Aunque es posible descargar estos ficheros individualmente o como un archivo comprimido, se aconseja utilizar el sistema de control de versiones mercurial².

Si encuentra alguna errata u omisión es los programas de ejemplo, por favor, utilice la herramienta de gestión de incidencias (*issue tracker*) accesible desde:

https://bitbucket.org/arco_group/ice-hello/issues

¹<http://arco.esi.uclm.es>

²<http://mercurial.selenic.com>

Sobre este documento

Los fuentes de este documentos (en \LaTeX) también se encuentran en un repositorio mercurial https://bitbucket.org/arco_group/ice-book aunque no es accesible públicamente. Si quiere colaborar activamente en el desarrollo o mejora de este documento póngase en contacto con los autores.

Al igual que los ejemplos, también existe una herramienta de gestión de incidencias (pública) en la que puede notificar problemas o errores de cualquier tipo que haya detectado en el documento.

Capítulo 1

Introducción

Un middleware de comunicaciones es un sofisticado sistema de IPC (Inter-Process Communication) orientado a mensajes. A diferencia de los otros IPC como los sockets, los middlewares suelen ofrecer soporte para interfaces concretas entre las entidades que se comunican, es decir, permiten definir la estructura y semántica para los mensajes.

Se puede entender un middleware como un software de conectividad que hace posible que aplicaciones distribuidas puedan ejecutarse sobre distintas plataformas heterogéneas, es decir, sobre plataformas con distintos sistemas operativos, que usan distintos protocolos de red, y que incluso involucran distintos lenguajes de programación en la aplicación distribuida.

Desde otro punto de vista, se puede ver el middleware como una abstracción de la complejidad y de la heterogeneidad que las redes de comunicaciones imponen. De hecho, uno de los objetivos de un middleware es ofrecer un acuerdo en las interfaces y en los mecanismos de interoperabilidad, como solución a los distintos desacuerdos en hardware, sistemas operativos, protocolos de red, y lenguajes de programación.

Existen muchísimos middlewares de comunicaciones: RPC (Remote Procedure Call), CORBA, EJB (Enterprise JavaBeans), Java RMI (Remote Method Invocation), DDS (Data Distribution Service), .Net Remoting, etc. En todos ellos, el programador puede especificar un API. En el caso de RPC se indican un conjunto de funciones que podrán ser invocadas remotamente por un cliente. Los demás, y la mayoría de los actuales, son RMI, es decir, son middlewares orientados a objetos. La figura 1.1 muestra el esquema de invocación remota a través de un núcleo de comunicaciones típico de este tipo de middlewares.

Gracias al middleware, en un diseño orientado a objetos, el ingeniero puede decidir qué entidades del dominio serán accesibles remotamente. Puede «particionar» su diseño, eligiendo qué objetos irán en cada nodo, cómo se comunicarán con los demás, cuáles serán los flujos de información y sus tipos. En definitiva está diseñando una aplicación distribuida. Obviamente todo eso tiene un coste, debe tener muy claro que una invocación remota puede resultar hasta 100 veces más lenta que una invocación local convencional.

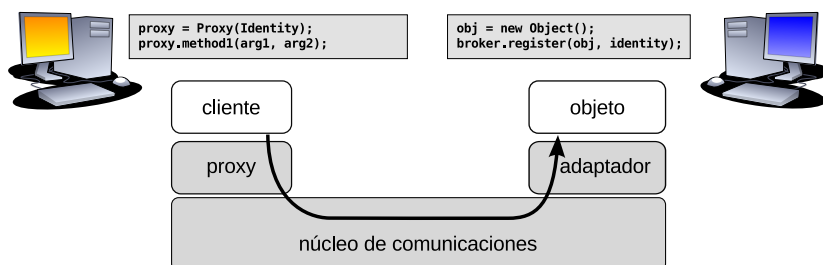


FIGURA 1.1: Invocación a método remoto

Las plataformas de objetos distribuidos tratan de acercar el modelo de programación de los sistemas distribuidos al de la programación de sistemas centralizados, es decir, ocultando los detalles de las llamadas a los procedimientos remotos. El enfoque más extendido entre las plataformas de objetos distribuidos es la generación automática de un *proxy* en la parte del cliente y clases base para los sirvientes en el servidor. Para ello, el cliente utiliza un objeto proxy con la misma interfaz definida en la parte del servidor, y que actúa como intermediario. El servidor, por otro lado, utiliza una clase base (una por cada interfaz especificada) encargada de traducir los eventos de la red a invocaciones sobre cada uno de los métodos. Como se puede apreciar, existen grandes similitudes (en cuanto al proceso al menos) entre la versión distribuida y la versión centralizada.

Obviamente, un middleware de comunicaciones se basa en las mismas primitivas del sistema operativo y el subsistema de red. El middleware no puede hacer nada que no puedan hacer los

sockets. Pero hay una gran diferencia; con el middleware lo haremos con mucho menos esfuerzo gracias a las abstracciones y servicios que proporciona, hasta el punto que habría muchísimas funcionalidades que serían prohibitivas en tiempo y esfuerzo sin el middleware. El middleware encapsula técnicas de programación de sockets y gestión de concurrencia que pueden ser realmente complejas de aprender, implementar y depurar; y que con él podemos aprovechar fácilmente. El middleware se encarga también de gestionar problema inherentes a las comunicaciones: identifica y numera los mensajes, comprueba duplicados, controla retransmisiones, conectividad, asigna puertos, gestiona el ciclo de vida de las conexiones, identifica los objetos, proporciona soporte para invocación y despacho asíncrono; y un largo etcétera. Por todo ello, la solución más extendida se basa en un núcleo de comunicaciones genérico y de un generador automático de **stubs**, que realizan la (de)serialización y que incluyen los proxies para el cliente y las clases base para los sirvientes en el servidor..

objeto distribuido

Es un objeto cuyos métodos (algunos al menos) pueden ser invocados remotamente.

1.1. ZeroC Ice

ICE (Internet Communication Engine) es un middleware de comunicaciones orientado a objetos desarrollado por la empresa ZeroC Inc¹. Implementa un modelo de objetos distribuidos similar al de CORBA, pero con un diseño mucho más sencillo.

ICE soporta múltiples lenguajes (Java, C#, C++, ObjectiveC, Python, Ruby, PHP, etc.) y multiplataforma (Windows, GNU/Linux, Solaris, Mac OS X, Android, IOS, etc.) lo que proporciona una gran flexibilidad para construir sistemas muy heterogéneos o integrar sistemas existentes.

Además ofrece *servicios comunes* muy valiosos para la propagación de eventos, persistencia, tolerancia a fallos, seguridad, etc.

Algunas ventajas importantes de Ice:

- El desarrollo multi-lenguaje no añade complejidad al proyecto, puesto que se utiliza la misma implementación para todos ellos.
- Los detalles de configuración de las comunicaciones (protocolos, puertos, etc.) son completamente ortogonales al desarrollo del software. Esto permite separar los roles del diseñador de aplicaciones distribuidas del arquitecto de sistema y retrasar decisiones de arquitectura del sistema hasta incluso después de haber completado el desarrollo inicial de la aplicación.
- El interfaz es relativamente sencillo y el significado de las operaciones se puede deducir fácilmente. Esto contrasta fuertemente con arquitecturas más veteranas, donde la terminología es suele ser más ambigua.

Tanto el cliente como el servidor se pueden ver como una mezcla de código de aplicación, código de bibliotecas, y código generado a partir de la especificación de las interfaces remotas.

El núcleo de ICE contiene el soporte de ejecución para las comunicaciones remotas en el lado del cliente y en el del servidor. De cara al desarrollador, dicho núcleo se corresponde con un determinado número de bibliotecas con las que la aplicación puede enlazar. El desarrollador utiliza el API para la gestión de tareas administrativas, como por ejemplo la inicialización y finalización del núcleo de ejecución.

1.2. Especificación de interfaces

Cuando nos planteamos una interacción con un objeto remoto, lo primero es definir el «contrato», es decir, el protocolo concreto que cliente y objeto (servidor) van a utilizar para comunicarse.

Antes de las RPC cada nueva aplicación implicaba definir un nuevo protocolo (ya sea un estándar o específico para una aplicación) y programar las rutinas de

¹<http://www.zeroc.com>

serialización y des-serIALIZACIÓN de los parámetros de los mensajes para convertirlos en secuencias de bytes, que es lo que realmente podemos enviar a través de los sockets. Esta tarea puede ser compleja porque se tiene que concretar la ordenación de bytes, el padding para datos estructurados, las diferencias entre arquitecturas, etc.

Los middlewares permiten especificar la interfaz mediante un lenguaje de programación de estilo declarativo. A partir de dicha especificación, un compilador genera código que encapsula toda la lógica necesaria para (des)serializar los mensajes específicos produciendo una representación externa canónica de los datos. A menudo el compilador también genera «esqueletos»² para el código dependiente del problema. El ingeniero únicamente tiene que rellenar ese esqueleto con la implementación concreta.

El lenguaje de especificación de interfaces de Ice se llama SLICE (Specification Language for Ice) y proporciona compiladores de interfaces (*translators*) para todos los lenguajes soportados dado que el código generado tendrá que compilar/enlazar con el código que aporte el programador de la aplicación. En cierto sentido, el compilador de interfaces es un generador de protocolos a medida para nuestra aplicación.



El lenguaje SLICE al igual que otros muchos lenguajes para definición de interfaces —como IDL (Interface Definition Language) o XDR (eXternal Data Representation)— son puramente declarativos, es decir, no permiten especificar lógica ni funcionalidad, únicamente interfaces.

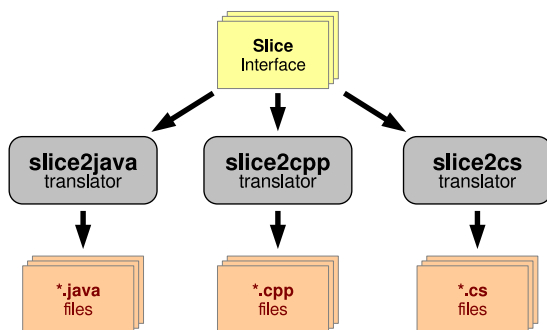


FIGURA 1.2: Ice proporciona compiladores para cada uno de los lenguajes soportados

1.2.1. Correspondencia con los lenguajes de implementación

Las reglas que definen cómo se traduce cada construcción SLICE en un lenguaje de programación específico se conocen como correspondencias con lenguajes (*mappings*). Por ejemplo, para la correspondencia con C++, una secuencia en Slice

²El *esqueleto* suele ser una definición de una clase en la que los cuerpos de los métodos están vacíos

aparece como un vector STL, mientras que para la correspondencia con Java, una secuencia en Slice aparece como un array.

Con el objetivo de determinar qué aspecto tiene la API generada para un determinado lenguaje, sólo se necesita conocer la especificación en Slice y las reglas de correspondencia hacia dicho lenguaje.

Actualmente, ICE proporciona correspondencias para los lenguajes C++, Java, C#, Visual Basic .NET, Python, y, para el lado del cliente, PHP y Ruby.

1.3. Terminología

Ice introduce una serie de conceptos técnicos que componen su propio vocabulario, como ocurre con cualquier nueva tecnología. Sin embargo, se procuró reutilizar la mayor parte de terminología existente en sistemas de este tipo, de forma que la cantidad de términos nuevos fuera mínima. De hecho, si el lector ha trabajado con tecnologías relacionadas como CORBA, la terminología aquí descrita le será muy familiar.

1.3.1. Clientes y servidores

Los términos cliente y servidor no están directamente asociados a dos partes distintas de una aplicación, sino que más bien hacen referencia a los roles que las diferentes partes de una aplicación pueden asumir durante una petición:

- Los clientes son entidades activas, es decir, solicita servicios a objetos remotos mediante invocaciones a sus métodos.
- Los servidores son entidades pasivas, es decir, proporcionan un servicio en respuesta a las solicitudes de los clientes. Se trata de programas o servicios que inicializan y activan los recursos necesarios para poner objetos a disposición de los clientes.



Nótese que *servidor* y *cliente* son roles en la comunicación, no tipos de programas. Es bastante frecuente que un mismo programa actúe como servidor (alojando objetos) a la vez que invoca métodos de otros, lo que convierte al sistema en una aplicación *peer-to-peer*.

1.3.2. Objetos

Un objeto ICE es una entidad conceptual o una abstracción que mantiene una serie de características:

- Es una entidad en el espacio de direcciones remoto o local que es capaz de responder a las peticiones de los clientes.
- Un único objeto puede instanciarse en un único servidor o, de manera redundante, en múltiples servidores. Si un objeto tienes varias instancias simultáneamente, todavía sigue siendo un único objeto.

- Cada objeto tiene una o más interfaces. Una interfaz es una colección de operaciones soportadas por un objeto (correspondientes a una especificación Slice). Los clientes emiten sus peticiones invocando dichas operaciones.
- Una operación tiene cero o más parámetros y un valor de retorno. Los parámetros y los valores de retorno tienen un tipo específico. Además, los parámetros tienen una determinada dirección: los parámetros de entrada se inicializan en la parte del cliente y se pasan al servidor, y los parámetros de salida se inicializan en el servidor y se pasan a la parte del cliente. El valor de retorno no es más que un parámetro de salida especial.
- Un objeto tiene una interfaz diferencia del resto y conocida como la *interfaz principal*. Además, un objeto puede proporcionar cero o más interfaces alternativas excluyentes, conocidas como facetas o facets. De esta forma, un cliente puede seleccionar entre las distintas facetas de un objeto para elegir la interfaz con la que quiere trabajar. Se trata de un concepto cercano al de los componentes.
- Cada objeto tiene una identidad de objeto única. La identidad de un objeto es un valor identificativo que distingue a un objeto del resto de objetos. El modelo de objetos definido por ICE asume que las identidades de los objetos son únicas de forma global, es decir, dos objetos no pueden tener la misma identidad dentro de un mismo dominio de comunicación.

1.3.3. Proxies

Para que un cliente sea capaz de comunicarse con un objeto ha de tener acceso a un proxy para el objeto. Un proxy es un componente local al espacio de direcciones del cliente, y representa al objeto (posiblemente remoto). Además, actúa como el representante local de un objeto, de forma que cuando un cliente invoca una operación en el proxy, el núcleo de comunicaciones:

1. Localiza al objeto remoto.
2. Activa el servidor del objeto si no está en ejecución.
3. Activa el objeto dentro del servidor.
4. Transmite los parámetros de entrada al objeto.
5. Espera a que la operación se complete.
6. Devuelve los parámetros de salida y el valor de retorno al cliente (o una excepción en caso de error).

Un proxy encapsula toda la información necesaria para que tenga lugar todo este proceso. En particular, un proxy contiene información asociada a diversas cuestiones:

- Información de direccionamiento que permite al núcleo de ejecución de la parte del cliente contactar con el servidor correcto.

- Información asociada a la identidad del objeto que identifica qué objeto particular es el destino de la petición en el servidor.
- Información sobre el identificador de faceta opcional que determina a qué faceta del objeto en concreto se refiere el proxy.

1.3.3.1. Proxies textuales (*stringified proxies*)

La información asociada a un proxy se puede representar como una cadena. Por ejemplo, la cadena `AlarmService:default -p 10000` es una representación legible de un proxy. El API de ICE proporciona funciones para convertir proxies en cadenas y viceversa. Sin embargo, este tipo de representación se utiliza para aplicaciones básicas y con propósitos didácticos, ya que su tratamiento requiere de operaciones adicionales que se pueden evitar utilizando otro tipo de representaciones expuestas a continuación.

1.3.3.2. Proxies directos (*direct proxies*)

Un proxy directo es un proxy que encapsula una identidad de objeto junto con la dirección asociada a su servidor. Dicha dirección está completamente especificada por los siguientes componentes:

- Un identificador de protocolo (como TCP o UDP).
- Una dirección específica de un protocolo (como el nombre y el puerto de una máquina).
- Con el objetivo de contactar con el objeto asociado a un proxy directo, el núcleo de ejecución de utiliza la información de direccionamiento vinculada al proxy para contactar con el servidor, de forma que la identidad del objeto se envía al servidor con cada petición realizada por el cliente.

1.3.3.3. Proxies indirectos (*indirect proxies*)

Un proxy indirecto tiene dos formas. Puede proporcionar sólo la identidad de un objeto, o puede especificar una identidad junto con un identificador de adaptador de objetos. Un objeto que es accesible utilizando sólo su identidad se denomina objeto bien conocido. Por ejemplo, la cadena `AlarmService` es un proxy válido para un objeto bien conocido con la identidad `AlarmService`. Un proxy indirecto que incluye un identificador de adaptador de objetos tiene la forma textual `AlarmService@AlarmAdapter`. Cualquier objeto del adaptador de objetos puede ser accedido utilizando dicho proxy sin importar si ese objeto también es un objeto bien conocido.

Se puede apreciar que un proxy indirecto no contiene información de direccionamiento. Para determinar el servidor correcto, el núcleo de ejecución de la parte del cliente pasa la información del proxy a un servicio de localización. Posteriormente, el servicio de localización utiliza la identidad del objeto o el identificador del adaptador de objetos como clave en una tabla de búsqueda que contiene la dirección del servidor y que devuelve la dirección del servidor actual al cliente. El núcleo

de ejecución de la parte del cliente es capaz ahora de contactar con el servidor y de enviar la solicitud del cliente de la manera habitual. El proceso completo es similar a la traducción de los nombres de dominio en Internet a direcciones IP, es decir, similar al DNS.

1.3.3.4. Binding directo e indirecto

El proceso de convertir la información de un proxy en una pareja protocolo-dirección se conoce como binding. De forma intuitiva, la resolución directa se usa para los proxies directos y la resolución indirecta para los proxies indirectos.

La principal ventaja de la resolución indirecta es que permita movilidad de servidores (es decir, cambiar su dirección) sin tener que invalidar los proxies existentes asociados a los clientes. De hecho, los proxies indirectos siguen trabajando aunque haya migración del servidor a otro lugar.

1.3.3.5. Proxies fijos (*fixed proxies*)

Un proxy fijo es un proxy que está asociado a una conexión en particular: en lugar de contener información de direccionamiento o de un nombre de adaptador, el proxy contiene un manejador de conexión. Dicho manejador de conexión permanece en un estado válido siempre que la conexión permanezca abierta. Si la conexión se cierra, el proxy no funciona (y no volverá a funcionar más). Los proxies fijos no pueden pasarse como parámetros a la hora de invocar operaciones, y son utilizados para permitir comunicaciones bidireccionales, de forma que un servidor puede efectuar retrollamadas a un cliente sin tener que abrir una nueva conexión.

1.3.4. Sirvientes (*servants*)

Como se comentó anteriormente, un objeto ICE es una entidad conceptual que tiene un tipo, una identidad, e información de direccionamiento. Sin embargo, las peticiones de los clientes deben terminar en una entidad de procesamiento en el lado del servidor que proporcione el comportamiento para la invocación de una operación. En otras palabras, una petición de un cliente ha de terminar en la ejecución de un determinado código en el servidor, el cual estará escrito en un determinado lenguaje de programación y ejecutado en un determinado procesador.

El componente en la parte del servidor que proporciona el comportamiento asociado a la invocación de operaciones se denomina sirviente. Un sirviente encarna a uno o más objetos distribuidos. En la práctica, un sirviente es simplemente una instancia de una clase escrita por un el desarrollador de la aplicación y que está registrada en el núcleo de ejecución de la parte del servidor como el sirviente para uno o más objetos. Los métodos de esa clase se corresponderían con las operaciones de la interfaz del objeto ICE y proporcionarían el comportamiento para dichas operaciones.

Un único sirviente puede encarnar a un único objeto en un determinado momento o a varios objetos de manera simultánea. En el primer caso, la identidad del objeto encarnado por el sirviente está implícita en el sirviente. En el segundo

caso, el sirviente mantiene la identidad del objeto con cada solicitud, de forma que pueda decidir qué objeto encarnar mientras dure dicha solicitud.

En cambio, un único objeto distribuido puede tener múltiples sirvientes. Por ejemplo, podríamos tomar la decisión de crear un proxy para un objeto con dos direcciones distintas para distintas máquinas. En ese caso, tendríamos dos servidores, en los que cada servidor contendría un sirviente para el mismo objeto. Cuando un cliente invoca una operación en dicho objeto, el núcleo de ejecución de la parte del cliente envía la petición a un servidor.

En otras palabras, tener varios sirvientes para un único objeto permite la construcción de sistemas redundantes: el núcleo de ejecución de la parte del cliente trata de enviar la petición a un servidor y, si dicho servidor falla, envía la petición al segundo servidor. Sólo en el caso de que el segundo servidor falle, el error se envía para que sea tratado por el código de la aplicación de la parte del cliente.

La figura 1.3 muestra los componentes principales del middleware y su relación en una aplicación típica que involucra a un cliente y a un servidor. Los componentes de color azul son proporcionados en forma de librerías o servicios. Los componentes marcados en naranja son generados por el compilador de interfaces.

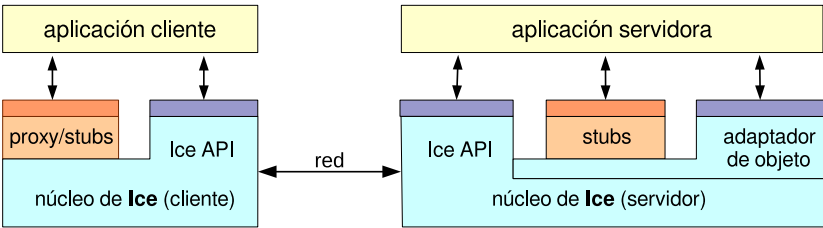


FIGURA 1.3: Componentes básicos del middleware

El diagrama de secuencia de la figura 1.4 describe una interacción completa correspondiente a una invocación remota síncrona, es decir, el cliente queda bloqueado hasta que la respuesta llega de vuelta. En el diagrama, el cliente efectúa una invocación local convencional sobre un método sobre el proxy. El proxy funciona como una referencia remota al objeto distribuido alojado en el servidor y por ello implementa la misma interfaz. El proxy serializa la invocación y construye un mensaje que será enviado al host servidor mediante un socket. Al tratarse de una llamada síncrona, el proxy queda a la espera de la respuesta lo que bloquea por tanto a la aplicación cliente.

Ya en el nodo servidor, el stub recibe el mensaje, lo des-serializa, identifica el objeto destino y sintetiza una llamada equivalente a la que realizó al cliente. A continuación realiza una invocación local convencional a un método del objeto destino y recoge el valor de retorno. Lo serializa en un mensaje de respuesta y lo envía de vuelta al nodo cliente. El proxy recoge ese mensaje y devuelve el valor de retorno al cliente, completando la ilusión de una invocación convencional.

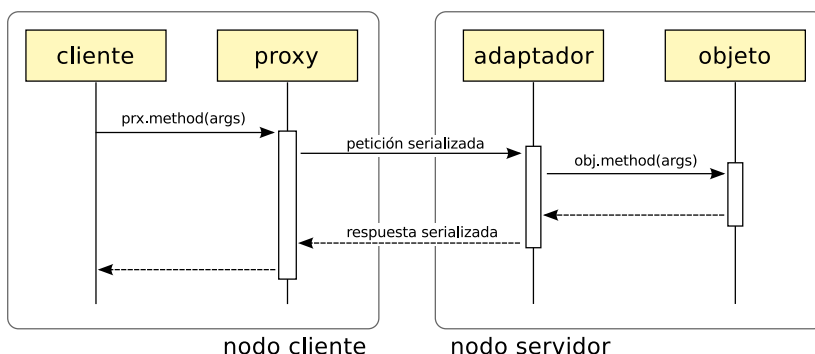


FIGURA 1.4: Diagrama de secuencia de una invocación a un objeto remoto

1.4. Semántica *at-most-once*

Las solicitudes ICE tienen una semántica *at-most-once*: el núcleo de comunicaciones hace todo lo posible para entregar una solicitud al destino correcto y, dependiendo de las circunstancias, puede volver a intentar una solicitud en caso de fallo. ICE Garantiza que entregará la solicitud o, en caso de que no pueda hacerlo, informará al cliente con una determinada excepción. Bajo ninguna circunstancia una solicitud se entregará dos veces, es decir, los reintentos se llevarán a cabo sólo si se conoce con certeza que un intento previo falló.

Esta semántica es importante porque asegura que las operaciones que no son idempotentes puedan usarse con seguridad. Una operación idempotente es aquella que, si se ejecuta dos veces, provoca el mismo efecto que si se ejecutó una vez. Por ejemplo, $x = 1$; es una operación idempotente, mientras que $x++$; no es una operación idempotente.

Sin este tipo de semánticas se pueden construir sistemas distribuidos robustos ante la presencia de fallos en la red. Sin embargo, los sistemas reales requieren operaciones no idempotentes, por lo que la semántica *at-most-once* es una necesidad, incluso aunque implique tener un sistema menos robusto ante la presencia de fallos. ICE permite marcar a una determinada operación como idempotente. Para tales operaciones, el núcleo de comunicaciones utiliza un mecanismo de recuperación de error más agresivo que para las operaciones no idempotentes.

1.5. Métodos de entrega

1.5.1. Invocaciones en una sola dirección (*oneway*)

Los clientes pueden invocar ciertas operaciones como una operación en una sola dirección (*oneway*). Dicha invocación mantiene una semántica *best-effort*. Para este tipo de invocaciones, el núcleo de ejecución de la parte del cliente entrega la invocación al transporte local, y la invocación se completa en la parte del cliente tan pronto como el transporte local almacene la invocación.

La invocación se envía de forma transparente por el sistema operativo. El servidor no responde a invocaciones de este tipo, es decir, el flujo de tráfico es sólo del cliente al servidor, pero no al revés.

Este tipo de invocaciones no son exactamente invocaciones a métodos. Por ejemplo, el objeto destino puede no existir, por lo que la invocación simplemente se perdería. De forma similar, la operación podría ser tratada por un sirviente en el servidor, pero dicha operación podría fallar (por ejemplo, porque los valores de los parámetros no fuesen correctos). En este caso, el cliente no recibiría ningún tipo de notificación al respecto.

Las invocaciones oneway son sólo posibles en operaciones que no tienen ningún tipo de valor de retorno, no tienen parámetros de salida, y no arrojan ningún tipo de excepción de usuario.

En lo que se refiere al código de aplicación de la parte del servidor estas invocaciones son transparentes, es decir, no existe ninguna forma de distinguir una invocación twoway de una oneway.

Las invocaciones oneway están disponibles sólo si el objeto destino ofrece un transporte orientado a flujo, como TCP o SSL.

1.5.2. Invocaciones por lotes en una sola dirección (*batched oneway*)

Cada invocación oneway envía un mensaje al servidor. En una serie de mensajes cortos, la sobrecarga es considerable: los núcleos de ejecución del cliente y del servidor deben cambiar entre el modo usuario y el modo núcleo para cada mensaje y, a nivel de red, se incrementa la sobrecarga debido a la afluencia de paquetes de control y de confirmación.

Las invocaciones batched oneway permiten enviar una serie de invocaciones oneway en un único mensaje: cada vez que se invoca una operación batched oneway, dicha invocación se almacena en un buffer en la parte del cliente. Una vez que se han acumulado todas las invocaciones a enviar, se lleva a cabo una llamada al API para enviar todas las invocaciones a la vez. A continuación, el núcleo de ejecución de la parte del cliente envía todas las invocaciones almacenadas en un único mensaje, y el servidor recibe todas esas invocaciones en un único mensaje. Este enfoque evita los inconvenientes descritos anteriormente.

Las invocaciones individuales en este tipo de mensajes se procesan por un único hilo en el orden en el que fueron colocadas en el buffer. De este modo se garantiza que las operaciones individuales sean procesadas en orden en el servidor.

Las invocaciones batched oneway son particularmente útiles para servicios de mensajes como IceStorm, y para las interfaces de grano fino que ofrecen operaciones set para atributos pequeños.

1.5.3. Invocaciones en modo datagrama

Este tipo de invocaciones mantienen una semántica best-effort similar a las invocaciones oneway. Sin embargo, requieren que el mecanismo de transporte empleado sea orientado a datagramas (UDP).

Estas invocaciones tienen las mismas características que las invocaciones oneway, pero abarcan un mayor número de escenarios de error:

- Las invocaciones pueden perderse en la red debido a la naturaleza del protocolo.
- Las invocaciones pueden llegar fuera de orden debido a la naturaleza del protocolo.

Este tipo de invocaciones cobran más sentido en el ámbito de las redes locales, en las que la posibilidad de pérdida es pequeña, y en situaciones en las que la baja latencia es más importante que la fiabilidad, como por ejemplo en aplicaciones interactivas en Internet.

Los protocolos orientados a datagramas tienen la característica adicional de que pueden tener direcciones multicast. Este tipo de invocaciones resulta muy conveniente para difundir mensajes a múltiples destinos o en procesos de anuncio o búsqueda de servicios. Al no existir mensajes de respuesta se consigue un importante ahorro de ancho de banda.

1.5.4. Invocaciones en modo datagrama por lotes (*batched datagram*)

Este tipo de invocaciones son análogas a las batched oneway, pero en el ámbito de los protocolos orientados a datagrama, como UDP.

1.6. Excepciones

Excepciones en tiempo de ejecución

Cualquier invocación a una operación puede lanzar una excepción en tiempo de ejecución. Las excepciones en tiempo de ejecución están predefinidas por el núcleo de ejecución de ICE y cubren condiciones de error comunes, como fallos de conexión, timeouts, o fallo en la asignación de recursos. Dichas excepciones se presentan a la aplicación como excepciones propias a C++, Java, o C#, por ejemplo, y se integran en la forma de tratar las excepciones de estos lenguajes de programación.

Excepciones definidas por el usuario

Las excepciones definidas por el usuario se utilizan para indicar condiciones de error específicas a la aplicación a los clientes. Dichas excepciones pueden llevar asociadas una determinada cantidad de datos complejos y pueden integrarse en jerarquías de herencia, lo cual permite que la aplicación cliente maneje diversas categorías de errores generales.

1.7. Propiedades

La mayor parte de los servicios y utilidades ICE se pueden configurar a través de las propiedades. Estos elementos son parejas «clave=valor», como por ejemplo:

```
Ice.Default.Protocol=tcp
```

Dichas propiedades están normalmente almacenadas en ficheros de texto generados de forma automática y son trasladadas al núcleo de comunicaciones para configurar diversas opciones, como el tamaño del pool de hilos, el nivel de trazado, y muchos otros parámetros de configuración.

Los nombres de las propiedades están jerarquizados y separados mediante puntos sencillos. Así por ejemplo, todas las propiedades relacionadas con el servicio IceGrid empiezan con la secuencia 'IceGrid.', mientras que todas las que comienzan con 'Ice.' se aplican a la configuración del núcleo de comunicaciones.

1.8. El protocolo IceP

De forma similar al protocolo GIOP (General IOP) de CORBA, ICE proporciona un protocolo de aplicación para la codificación de las invocaciones a métodos remotos que puede usarse sobre diversos protocolos de transporte. El protocolo ICE define:

- Un conjunto de tipos de mensajes (solicitud, respuesta, etc.).
- Una máquina de estados que determina qué secuencia siguen los distintos tipos de mensajes que se intercambian el cliente y el servidor, junto con el establecimiento de conexión asociado.
- Reglas de codificación que determinan cómo se representa cada tipo de datos «en el cable».
- Una cabecera para cada tipo de mensaje que contiene detalles como el tipo del mensaje, su tamaño, protocolo y la versión de codificación empleada.

ICE también soporta compresión en la red. Ajustando un parámetro de configuración se pueden comprimir todos los datos asociados al tráfico de red para reducir el ancho de banda utilizado. Esta propiedad puede resultar útil si la aplicación intercambia grandes cantidades de datos entre el cliente y el servidor.

El protocolo ICEP (ICE Protocol) también soporta operaciones bidireccionales: si un servidor quiere enviar un mensaje a un objeto de retrollamada proporcionado por el cliente, la retrollamada puede efectuarse a partir de la conexión establecida inicialmente por el cliente. Esta característica es especialmente importante cuando el cliente está detrás de un cortafuegos que permite conexiones de salida pero no de entrada.

Capítulo 2c

«Hola mundo» distribuido

[C++]

En esta primera aplicación, el servidor proporciona un objeto que dispone de un único método remoto llamado `write()`. Este método imprime en la salida estándar del servidor la cadena que el cliente le pase como parámetro.

Tal como hemos visto, lo primero que necesitamos es escribir la especificación de la interfaz remota para estos objetos. El siguiente listado corresponde al fichero `Printer.ice` y contiene la interfaz *Printer* en lenguaje SLICE.

LISTADO 2C.1: Especificación SLICE para un «Hola mundo» distribuido
`Printer.ice`

```
1 module Example {  
2     interface Printer {  
3         void write(string message);  
4     };  
5 };
```

Lo más importante de este fichero es la declaración del método `write()`. El compilador de interfaces generará los stubs que incluyen una versión básica de la interfaz *Printer* en el lenguaje de programación que el programador decida. Cualquier clase que herede de esa interfaz *Printer* debería redefinir (especializar) un método `write()`, que podrá ser invocado remotamente, y que debe tener la misma signatura. De hecho, en la misma aplicación distribuida puede haber varias implementaciones del mismo interfaz en una o varias computadoras y escritos en diferentes lenguajes.

El compilador también genera los «cabos» para el cliente. Igual que antes, los clientes escritos en distintos lenguajes o sobre distintas arquitecturas podrán usar los objetos remotos que cumplan la misma interfaz.

Para generar los stubs de cliente y servidor en C++ (para este ejemplo) usamos el *translator slice2cpp*.

```
$ slice2cpp Printer.ice
```

Esto genera dos ficheros llamados `Printer.cpp` y `Printer.h` que deben ser compilados para obtener las aplicaciones cliente y servidor.

2c.1. Sirviente

El compilador de interfaces genera la clase *Example::Printer*. La implementación del *sirviente* debe heredar de esa interfaz, proporcionando una implementación (por sobrecarga) de los métodos especificados en la interfaz SLICE.

El propio compilador de interfaces puede generar una clase «hueca» que sirva al programador como punto de partida para implementar el sirviente:

```
$ slice2cpp --impl Printer.ice
```

De este modo genera además los ficheros `PrinterI.cpp` y `PrinterI.h`. La letra ‘I’ hace referencia a «Implementación de la interfaz». El fichero de cabecera generado (`PrinterI.h`) tiene el siguiente aspecto:

LISTADO 2C.2: Sirviente de la aplicación Printer
`cpp/PrinterI.h`

```

1  #ifndef __PrinterI_h__
2  #define __PrinterI_h__
3
4  #include <Printer.h>
5
6  namespace Example {
7
8      class PrinterI : virtual public Printer {
9      public:
10         virtual void write(const ::std::string&,
11                             const Ice::Current&);
12     };
13 }
14
15 #endif
```

Y el fichero de implementación `PrinterI.cpp`:

LISTADO 2C.3: Sirviente de la aplicación Printer
`cpp/PrinterI.cpp`

```

1  #include <iostream>
2  #include "PrinterI.h"
3
4  void
5  Example::PrinterI::write(const ::std::string& message,
6                          const Ice::Current& current) {
7      std::cout << message << std::endl;
8  }
```

La única modificación respecto al fichero generado está en la **línea 7**.

2c.2. Servidor

Nuestro servidor consiste principalmente en la implementación de una clase que hereda de *Ice::Application*. De ese modo se ahorra parte del trabajo de inicialización del comunicador¹. En esta clase debemos definir el método `run()`. Veamos el código en detalle:

LISTADO 2C.4: Servidor de la aplicación Printer
`cpp/Server.cpp`

```

1  #include <Ice/Ice.h>
2  #include "PrinterI.h"
3
4  using namespace std;
5  using namespace Ice;
6
7  class Server: public Application {
8  int run(int argc, char* argv[]) {
9      Example::PrinterPtr servant = new Example::PrinterI();
10
11      ObjectAdapterPtr adapter =
12          communicator()->createObjectAdapter("PrinterAdapter");
13      ObjectPrx proxy = adapter->add(
14          servant, communicator()->stringToIdentity("printer1"));
15
16      cout << communicator()->proxyToString(proxy) << endl;
17
18      adapter->activate();
19      shutdownOnInterrupt();
20      communicator()->waitForShutdown();
21
22      return 0;
23  }
24 };
25
26 int main(int argc, char* argv[]) {
27     Server app;
28     return app.main(argc, argv);
29 }
```

En la **línea 9** se crea el sirviente (una instancia de la clase `PrinterI`). En la **línea 20** se crea un adaptador de objetos, que es el componente encargado de multiplexar entre los objetos alojados en el servidor. El adaptador requiere un endpoint —un punto de conexión a la red materializado por un protocolo (TCP o UDP), un host y un puerto. En este caso esa información se extrae de un fichero de configuración a partir del nombre del adaptador (`PrinterAdapter` en este caso).

En la **línea 13** se registra el sirviente en el adaptador mediante el método `add()`, indicando para ello la identidad que tendrá el objeto (`printer1`). En este ejemplo la identidad es un identificador bastante simple, pero lo recomendable es utilizar una

¹El comunicador representa el *broker* de objetos del núcleo de comunicaciones: http://en.wikipedia.org/wiki/Object_request_broker

secuencia globalmente única (UUID). El método `add()` devuelve una referencia al objeto distribuido recién creado, que se denomina *proxy*.

La **línea 18** corresponde con la activación del adaptador, que se ejecuta en otro hilo. A partir de ese momento el servidor puede escuchar y procesar peticiones para sus objetos. El método `waitForShutown()` (**línea 20**) bloquea el hilo principal hasta que el comunicador sea terminado. El método `shutdownOnInterrupt()` (**línea 19**) indica a la aplicación que termine el comunicador al recibir la señal `SIGQUIT` (Control-C).

Por último, las **líneas 26–29** contienen la función `main()` en la que se crea una instancia de la clase `Server` y se invoca su método `main()`.

Con esto tenemos el servidor completo. Pero existe un problema, el cliente necesita un modo de referenciar el objeto alojado en el servidor. ICE proporciona mecanismos de localización (que veremos en capítulos posteriores) pero en este primer ejemplo haremos algo más sencillo. Existe una forma de conseguir una representación textual² del proxy del objeto —el valor devuelto por el método `add()` del adaptador. A su vez, el cliente puede generar un objeto a partir de esa representación textual que puede utilizar para contactar con el servidor.

En su forma programática, el proxy funciona como una especie de referencia o puntero para acceder al objeto remoto.

2c.3. Cliente

La aplicación cliente únicamente debe conseguir una referencia al objeto remoto e invocar el método `write()`. El cliente también se puede implementar como una especialización de la clase `Ice::Application`. El código completo del cliente aparece a continuación:

LISTADO 2C.5: Cliente de la aplicación Printer
`cpp/Client.cpp`

```

1  #include <Ice/Ice.h>
2  #include "Printer.h"
3
4  using namespace Ice;
5  using namespace Example;
6
7  class Client: public Ice::Application {
8  int run(int argc, char* argv[]) {
9      ObjectPrx proxy = communicator()->stringToProxy(argv[1]);
10     PrinterPrx printer = PrinterPrx::checkedCast(proxy);
11
12     printer->write("Hello, World!");
13
14     return 0;
15 }
16 };
17

```

² *stringified proxy*

```

18  int main(int argc, char* argv[]) {
19      Client app;
20      return app.main(argc, argv);
21  }

```

El programa acepta por línea de comandos la representación textual del proxy del objeto remoto. A partir de ella se obtiene un objeto proxy (**línea 8**). Sin embargo, esa referencia es para un proxy genérico. Para poder invocar los métodos de la interfaz *Printer* se requiere una referencia de tipo *PrinterPrx*, es decir, un proxy a un objeto remoto *Printer*.

Para lograrlo se ha de efectuar *downcasting*³ mediante el método `PrinterPrx::checkedCast()` (**línea 9**). Gracias al soporte de introspección de los objetos remotos, ICE puede comprobar que efectivamente el objeto remoto es del tipo al que tratamos de convertirlo. Esta comprobación no se realizaría si empleáramos la modalidad `uncheckedCast()`.

Una vez conseguido el proxy del tipo correcto (objeto *printer*) se puede invocar el método remoto `write()` (**línea 12**) pasando por parámetro la cadena "Hello, world!".

2c.4. Compilación

La figura 2c.1 muestra el esquema de compilación del cliente y servidor a partir del fichero de especificación de la interfaz. Los ficheros marcados en amarillo corresponden a aquellos que el programador debe escribir o completar, mientras que los ficheros generados aparecen en verde.

Para automatizar el proceso de compilación se puede utilizar el fichero *Makefile*, que se muestra en el listado 2c.6.

LISTADO 2c.6: Aplicación Printer
cpp/Makefile

```

1  #!/usr/bin/make -f
2  # -*- mode:makefile -*-
3
4  include ../cpp-common.mk
5
6  CC=g++ -std=c++11
7
8  APP=Printer
9  STUBS=$(addprefix $(APP), .h .cpp)
10 TARGET=Server Server-UUID Client
11
12 all: $(TARGET)
13
14 Server: Server.o $(APP)I.o $(APP).o
15 Server-UUID: Server-UUID.o $(APP)I.o $(APP).o

```

³Consiste en moldear (*cast*) un puntero o referencia de una clase a una de sus subclases, asumiendo que realmente es una instancia de ese tipo: <http://en.wikipedia.org/wiki/Downcasting>.

```

16 Client: Client.o $(APP).o
17
18 Server.cpp Client.cpp: $(STUBS)
19
20 %.cpp %.h: %.ice
21     slice2cpp $<
22
23 dist:
24     mkdir dist
25
26 gen-dist: all dist
27     cp Client Server dist/
28     icepatch2calc dist/
29
30 clean:
31     $(RM) $(TARGET) $(STUBS) *.o *~
32     $(RM) *.bz2 IcePatch2.sum
33     $(RM) -r dist
34
35 calc:
36     icepatch2calc .
37
38 run-server: Server
39     ./Server --Ice.Config=Server.config | tee proxy.out
40
41 run-client: Client
42     ./Client '$(shell head -1 proxy.out)'

```

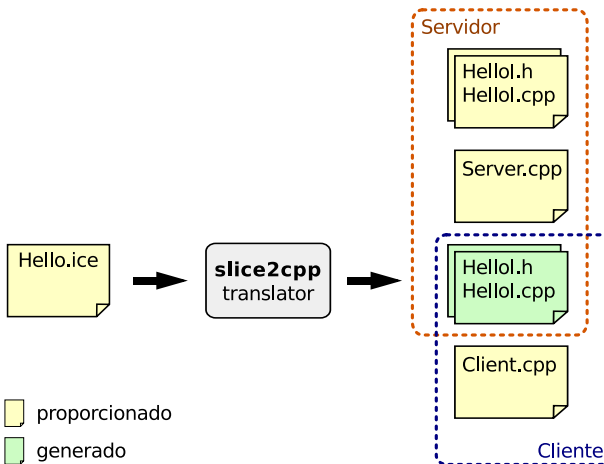


FIGURA 2C.1: Esquema de compilación de cliente y servidor en C++

2c.5. Ejecutando el servidor

Si se intenta ejecutar el servidor sin argumentos se obtiene un error:


```
$ ./Server
!! 03/10/12 19:52:05.733 ./Server: error: ObjectAdapterI.cpp:915: Ice::
    InitializationException:
    initialization exception:
    object adapter 'PrinterAdapter' requires configuration
```

Como vimos en la sección 2c.2, el servidor necesita información específica que le indique los *endpoints* en los que debe escuchar el adaptador `PrinterAdapter`. Para ello, debemos proporcionar un fichero adicional (`Server.config`) cuyo contenido aparece en el siguiente listado:

LISTADO 2C.7: Servidor de la aplicación Printer
`Server.config`

```
1 PrinterAdapter.Endpoints=tcp -p 9090
```

Este tipo de ficheros contiene definiciones de *propiedades*, que son parejas clave=valor. La mayoría de los servicios de ICE pueden configurarse por medio de propiedades, lo que le otorga gran flexibilidad sin necesidad de recompilar el código. Esta propiedad en concreto indica que el adaptador debe utilizar un socket TCP en el puerto 9090. Al no especificar una IP, el adaptador escuchará en todas las interfaces de red del computador. El puerto tampoco es obligatorio. De no especificarlo, hubiera elegido automáticamente uno libre. Lo que sí es obligatorio es el protocolo que debe usar (en este caso TCP).

Para que el servidor cargue las propiedades del fichero de configuración, ejecute:

```
$ ./Server --Ice.Config=Server.config
printer1 -t:tcp -h 192.168.0.12 -p 9090:tcp -h 10.1.1.10 -p 9090
```

En esta ocasión el programa arranca correctamente y queda ocupando la shell como corresponde a cualquier servidor. Lo que aparece en consola es, como se indicó anteriormente, la representación textual del proxy para el objeto distribuido⁴. La línea contiene varios datos:

- La identidad del objeto: `'printer1'`.
- El tipo de proxy (`'-t'`), que corresponde a *twoway*. Existen otros tipos que implican semánticas de llamada diferentes: `'-o'` para *oneway*, `'-d'` para *datagram*, etc.
- Una lista de endpoints separados con el carácter `':'`, que corresponden con sockets, es decir, un protocolo, una dirección IP (parámetro `'-h'`) y un puerto (parámetro `'-p'`).

Concretamente, el adaptador de nuestro servidor escucha en el puerto TCP de dos interfaces de red puesto que el fichero de configuración no lo limita a una interfaz concreta.

⁴<http://doc.zeroc.com/display/Ice/Proxy+and+Endpoint+Syntax>

2c.6. Ejecutando el cliente

Para ejecutar el cliente debemos indicar el proxy del objeto remoto en línea de comandos, precisamente el dato que imprime el servidor. Nótese que se debe escribir entre comillas para que la shell lo interprete como un único parámetro:

```
$ ./Client "printer1 -t:tcp -h 192.168.0.12 -p 9090"
```

El programa se ejecuta y retorna inmediatamente. El servidor mostrará por consola la cadena "Hello, world!" como corresponde a la implementación del método `write()` del sirviente.

2c.7. Ejercicios

- E 2c.01** Modifica el cliente para que acepte una cadena por línea de comandos en lugar de enviar siempre "Hello World". Comprueba que funciona.
- E 2c.02** Ejecuta varios clientes que envían cadenas distintas.
- E 2c.03** Con la herramienta más simple que conozcas, comprueba que el servidor efectivamente está escuchando en el puerto especificado.
- E 2c.04** Ejecuta servidor y cliente en computadores diferentes.
- E 2c.05** Modifica el sirviente para que acepte una cadena de texto en el constructor y la imprima junto con cada mensaje entrante.
- E 2c.06** Modifica el servidor creando varios sirvientes `PrinterI` y añadiéndolos al adaptador `PrinterAdapter`. Imprime los proxies de cada uno de ellos. Invócalos desde distintos clientes. Ahora escribe un cliente que crea varios objetos, pero que utiliza la misma instancia del sirviente para todos ellos.
- E 2c.07** Intenta ejecutar el servidor en dos terminales distintos en el mismo computador ¿qué ocurre?
- E 2c.08** Edita el fichero `Server.config` y modifica el endpoint del adaptador para que escuche en un puerto diferente. Comprueba que el cliente funciona correctamente con el nuevo puerto.
- E 2c.09** Elimina el puerto de la especificación del endpoint en `Server.config` y comprueba que servidor y cliente siguen funcionando. Ahora puedes ejecutar dos servidores en el mismo computador ¿por qué?
- E 2c.10** Añade un endpoint UDP a `Server.config`, arranca el servidor y ejecuta dos clientes que invocan el objeto utilizando cada uno un endpoint diferente. Escribe el comando `tshark`⁵ que demuestra que efectivamente un cliente utiliza transporte TCP y el otro UDP.

⁵<http://www.wireshark.org/docs/man-pages/tshark.html>

- E 2c.11** Añade una pausa de 3 segundos al comienzo del método `write()` del sirviente. Ejecuta dos clientes en terminales diferentes tratando de que arranquen con la mínima diferencia de tiempo posible. A la vista del resultado ¿crees que el objeto está atendiendo a los clientes de forma concurrente? ¿Por qué?
- E 2c.12** Averigua qué significa exactamente cada uno de los parámetros que aparecen en el proxy textual que imprime el servidor. Averigua cómo alterar estos parámetros mediante el API de ICE y escribe un programa que lo demuestre.
- E 2c.13** Crea una versión diferente del servidor. Éste, en lugar de imprimir el mensaje recibido en pantalla, lo enviará (añadiendo el prefijo «*redirect:*» al mensaje original) a otro objeto *Printer*. Para ello necesitarás el proxy de ese otro objeto, que puedes crear con la versión original del servidor. Utilizando el cliente original invoca a los objetos alojados en cada uno de los servidores.

Capítulo 2j

«Hola mundo» distribuido

[Java]

En esta primera aplicación, el servidor proporciona un objeto que dispone de un único método remoto llamado `write()`. Este método imprime en la salida estándar del servidor la cadena que el cliente le pase como parámetro.

Tal como hemos visto, lo primero que necesitamos es escribir la especificación de la interfaz remota para estos objetos. El siguiente listado corresponde al fichero `Printer.ice` y contiene la interfaz *Printer* en lenguaje SLICE.

LISTADO 2J.1: Especificación SLICE para un «Hola mundo» distribuido
`Printer.ice`

```
1 module Example {  
2   interface Printer {  
3     void write(string message);  
4   };  
5 };
```

Lo más importante de este fichero es la declaración del método `write()`. El compilador de interfaces generará los stubs que incluyen una versión básica de la interfaz *Printer* en el lenguaje de programación que el programador decida. Cualquier clase que herede de esa interfaz *Printer* debería redefinir (especializar) un método `write()`, que podrá ser invocado remotamente, y que debe tener la misma signatura. De hecho, en la misma aplicación distribuida puede haber varias implementaciones del mismo interfaz en una o varias computadoras y escritos en diferentes lenguajes.

El compilador también genera los «cabos» para el cliente. Igual que antes, los clientes escritos en distintos lenguajes o sobre distintas arquitecturas podrán usar los objetos remotos que cumplan la misma interfaz.

Para generar los stubs de cliente y servidor en Java (para este ejemplo) usamos el *translator slice2java*.

```
$ slice2java Printer.ice
```

Esto genera un paquete cuyo nombre se corresponde con el del módulo que hay en el fichero `Printer.ice` (en este caso `Example`). Dentro se encuentran las clases que se usarán tanto en el cliente como en el servidor, pero su contenido exacto no nos interesa ahora mismo.

2j.1. Sirviente

El compilador de interfaces genera la clase `Example._PrinterDisp`. La implementación del *sirviente* debe heredar de esa interfaz, proporcionando una implementación (por sobrecarga) de los métodos especificados en la interfaz `SLICE`.

El propio compilador de interfaces puede generar una clase «hueca» que sirva al programador como punto de partida para implementar el sirviente:

```
$ slice2java --impl Printer.ice
```

De este modo genera además el fichero `Example/PrinterI.java`. La letra ‘I’ hace referencia a «Implementación de la interfaz». El fichero generado tiene el siguiente aspecto:

LISTADO 2J.2: Sirviente de la aplicación Printer
`java/PrinterI.java`

```
1 public final class PrinterI extends Example._PrinterDisp {
2     public PrinterI() {}
3
4     public void write(String message, Ice.Current current) {
5         System.out.println(message);
6     }
7 }
```

La única modificación relevante respecto al fichero generado está en la **línea 5**.

2j.2. Servidor

Nuestro servidor consiste principalmente en la implementación de una clase que hereda de `Ice.Application`. De ese modo se ahorra parte del trabajo de inicialización del *communicator*¹. En esta clase debemos definir el método `run()`. Veamos el código en detalle:

LISTADO 2J.3: Servidor de la aplicación Printer
`java/Server.java`

```
1 import Ice.*;
2
3 public class Server extends Ice.Application {
```

¹El *communicator* representa el *broker* de objetos del núcleo de comunicaciones: http://en.wikipedia.org/wiki/Object_request_broker

```

4      public int run(String[] args) {
5          Ice.Object servant = new PrinterI();
6
7          ObjectAdapter adapter =
8              communicator().createObjectAdapter("PrinterAdapter");
9          ObjectPrx proxy =
10             adapter.add(servant, Util.stringToIdentity("printer1"));
11
12             System.out.println(communicator().proxyToString(proxy));
13
14             adapter.activate();
15             shutdownOnInterrupt();
16             communicator().waitForShutdown();
17
18             return 0;
19         }
20
21         static public void main(String[] args) {
22             Server app = new Server();
23             app.main("Server", args);
24         }
25     }

```

En la **línea 5** se crea el sirviente (una instancia de la clase `PrinterI`). En la **línea 16** se crea un adaptador de objetos, que es el componente encargado de multiplexar entre los objetos alojados en el servidor. El adaptador requiere un endpoint —un punto de conexión a la red materializado por un protocolo (TCP o UDP), un host y un puerto. En este caso esa información se extrae de un fichero de configuración a partir del nombre del adaptador (`PrinterAdapter` en este caso).

En la **línea 10** se registra el sirviente en el adaptador mediante el método `add()`, indicando para ello la identidad que tendrá el objeto (`printer1`). En este ejemplo la identidad es un identificador bastante simple, pero lo recomendable es utilizar una secuencia globalmente única (UUID). El método `add()` devuelve una referencia al objeto distribuido recién creado, que se denomina *proxy*.

La **línea 14** corresponde con la activación del adaptador, que se ejecuta en otro hilo. A partir de ese momento el servidor puede escuchar y procesar peticiones para sus objetos. El método `waitForShutown()` (**línea 16**) bloquea el hilo principal hasta que el comunicador sea terminado. El método `shutdownOnInterrupt()` (**línea 15**) indica a la aplicación que termine el comunicador al recibir la señal `SIGQUIT` (Control-C).

Por último, las **líneas 21–24** contienen la función `main()` en la que se crea una instancia de la clase `Server` y se invoca su método `main()`.

Con esto tenemos el servidor completo. Pero existe un problema, el cliente necesita un modo de referenciar el objeto alojado en el servidor. ICE proporciona mecanismos de localización (que veremos en capítulos posteriores) pero en este primer ejemplo haremos algo más sencillo. Existe una forma de conseguir una representación textual² del proxy del objeto —el valor devuelto por el método

²*stringified proxy*

`add()` del adaptador. A su vez, el cliente puede generar un objeto a partir de esa representación textual que puede utilizar para contactar con el servidor.

En su forma programática, el proxy funciona como una especie de referencia o puntero para acceder al objeto remoto.

2j.3. Cliente

La aplicación cliente únicamente debe conseguir una referencia al objeto remoto e invocar el método `write()`. El cliente también se puede implementar como una especialización de la clase *Ice.Application*. El código completo del cliente aparece a continuación:

LISTADO 2J.4: Cliente de la aplicación Printer
java/Client.java

```

1  public class Client extends Ice.Application {
2      public int run(String[] args) {
3          Ice.ObjectPrx obj = communicator().stringToProxy(args[0]);
4          Example.PrinterPrx printer = Example.PrinterPrxHelper.checkedCast(obj);
5
6          printer.write("Hello, World!");
7
8          return 0;
9      }
10
11     static public void main(String[] args) {
12         Client app = new Client();
13         app.main("Client", args);
14     }
15 }
```

El programa acepta por línea de comandos la representación textual del proxy del objeto remoto. A partir de ella se obtiene un objeto proxy (**línea 3**). Sin embargo, esa referencia es para un proxy genérico. Para poder invocar los métodos de la interfaz *Printer* se requiere una referencia de tipo `PrinterPrx`, es decir, un proxy a un objeto remoto *Printer*.

Para lograrlo se ha de efectuar *downcasting*³ mediante el método `PrinterPrxHelper.checkedCast()` (**línea 4**). Gracias al soporte de introspección de los objetos remotos, ICE puede comprobar que efectivamente el objeto remoto es del tipo al que tratamos de convertirlo. Esta comprobación no se realizaría si empleáramos la modalidad `uncheckedCast()`.

Una vez conseguido el proxy del tipo correcto (objeto `printer`) se puede invocar el método remoto `write()` (**línea 6**) pasando por parámetro la cadena "Hello, world!".

³Consiste en moldear (*cast*) un puntero o referencia de una clase a una de sus subclases, asumiendo que realmente es una instancia de ese tipo: <http://en.wikipedia.org/wiki/Downcasting>.

2j.4. Compilación

La figura 2j.1 muestra el esquema de compilación del cliente y servidor a partir del fichero de especificación de la interfaz. Los ficheros marcados en amarillo corresponden a aquellos que el programador debe escribir o completar, mientras que los ficheros generados aparecen en verde.

Para automatizar el proceso de compilación se puede utilizar el fichero `Makefile`, que se muestra en el listado 2j.5.

LISTADO 2j.5: Aplicación Printer
java/Makefile

```

1  #!/usr/bin/make -f
2  # -*- mode:makefile -*-
3
4  CLASSPATH=-classpath ../usr/share/java/Ice.jar
5
6  all: Example Server.class Client.class
7
8  %.class: %.java
9      javac $(CLASSPATH) $<
10
11 Example: Printer.ice
12     slice2java $<
13
14 dist:
15     mkdir dist
16
17 gen-dist: all dist
18     cp -r *.class Example dist/
19     icepatch2calc dist/
20
21 clean:
22     $(RM) *.class proxy.out *~
23     $(RM) -r Example
24     $(RM) -r dist
25
26 run-server: Server.class
27     java $(CLASSPATH) \
28         Server --Ice.Config=Server.config | tee proxy.out
29
30 run-client: Client.class
31     java $(CLASSPATH) \
32         Client '$(shell head -1 proxy.out)'
```

2j.5. Ejecutando el servidor

Si se intenta ejecutar el servidor sin argumentos se obtiene un error:

```

$ java -classpath /usr/share/java/Ice.jar:. Server
!! 11/04/12 09:49:24:284 Server: error: main: Ice.InitializationException
    reason = "object adapter `PrinterAdapter' requires configuration"
```

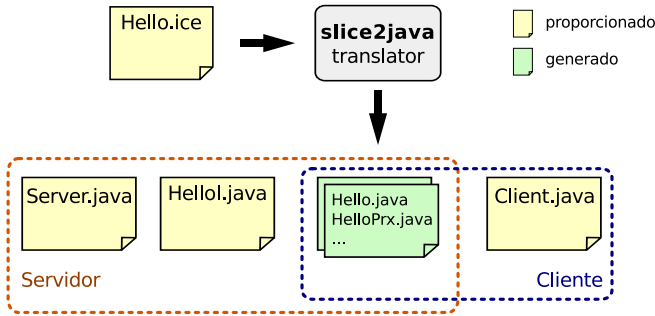


FIGURA 2J.1: Esquema de compilación de cliente y servidor en Java

```

at Ice.ObjectAdapterI.<init>(ObjectAdapterI.java:884)
at IceInternal.ObjectAdapterFactory.createObjectAdapter(ObjectAdapterFactory.java
:130)
at Ice.CommunicatorI.createObjectAdapter(CommunicatorI.java:77)
at Server.run(Server.java:7)
at Ice.Application.doMain(Application.java:202)
at Ice.Application.main(Application.java:182)
at Ice.Application.main(Application.java:71)
at Server.main(Server.java:23)

```

Como vimos en la sección 2j.2, el servidor necesita información específica que le indique los *endpoints* en los que debe escuchar el adaptador `PrinterAdapter`. Para ello, debemos proporcionar un fichero adicional (`Server.config`) cuyo contenido aparece en el siguiente listado:

LISTADO 2J.6: Servidor de la aplicación Printer
`Server.config`

```

1 PrinterAdapter.Endpoints=tcp -p 9090

```

Este tipo de ficheros contiene definiciones de *propiedades*, que son parejas clave=valor. La mayoría de los servicios de ICE pueden configurarse por medio de propiedades, lo que le otorga gran flexibilidad sin necesidad de recompilar el código. Esta propiedad en concreto indica que el adaptador debe utilizar un socket TCP en el puerto 9090. Al no especificar una IP, el adaptador escuchará en todas las interfaces de red del computador. El puerto tampoco es obligatorio. De no especificarlo, hubiera elegido automáticamente uno libre. Lo que sí es obligatorio es el protocolo que debe usar (en este caso TCP).

Para que el servidor cargue las propiedades del fichero de configuración, ejecute:

```

$ java -classpath /usr/share/java/Ice.jar:. Server --Ice.Config=Server.config
printer1 -t:tcp -h 192.168.0.12 -p 9090:tcp -h 10.1.1.10 -p 9090

```

En esta ocasión el programa arranca correctamente y queda ocupando la shell como corresponde a cualquier servidor. Lo que aparece en consola es, como se indicó

anteriormente, la representación textual del proxy para el objeto distribuido⁴. La línea contiene varios datos:

- La identidad del objeto: 'printer1'.
- El tipo de proxy ('-t'), que corresponde a *twoway*. Existen otros tipos que implican semánticas de llamada diferentes: '-o' para *oneway*, '-d' para *datagram*, etc.
- Una lista de endpoints separados con el carácter ':', que corresponden con sockets, es decir, un protocolo, una dirección IP (parámetro '-h') y un puerto (parámetro '-p').

Concretamente, el adaptador de nuestro servidor escucha en el puerto TCP de dos interfaces de red puesto que el fichero de configuración no lo limita a una interfaz concreta.

2j.6. Ejecutando el cliente

Para ejecutar el cliente debemos indicar el proxy del objeto remoto en línea de comandos, precisamente el dato que imprime el servidor. Nótese que se debe escribir entre comillas para que la shell lo interprete como un único parámetro:

```
$ java -classpath /usr/share/java/Ice.jar:. Client "printer1 -t:tcp -h 192.168.0.12 -p 9090"
```

El programa se ejecuta y retorna inmediatamente. El servidor mostrará por consola la cadena "Hello, World!" como corresponde a la implementación del método `write()` del sirviente.

2j.7. Ejercicios

- E 2j.01** Modifica el cliente para que acepte una cadena por línea de comandos en lugar de enviar siempre "Hello World". Comprueba que funciona.
- E 2j.02** Ejecuta varios clientes que envían cadenas distintas.
- E 2j.03** Con la herramienta más simple que conozcas, comprueba que el servidor efectivamente está escuchando en el puerto especificado.
- E 2j.04** Ejecuta servidor y cliente en computadores diferentes.
- E 2j.05** Modifica el sirviente para que acepte una cadena de texto en el constructor y la imprima junto con cada mensaje entrante.
- E 2j.06** Modifica el servidor creando varios sirvientes `PrinterI` y añadiéndolos al adaptador `PrinterAdapter`. Imprime los proxies de cada uno de ellos. Invócalos desde distintos clientes. Ahora escribe un cliente que crea varios objetos, pero que utiliza la misma instancia del sirviente para todos ellos.

⁴<http://doc.zeroc.com/display/Ice/Proxy+and+Endpoint+Syntax>

- E 2j.07** Intenta ejecutar el servidor en dos terminales distintos en el mismo computador ¿qué ocurre?
- E 2j.08** Edita el fichero `Server.config` y modifica el endpoint del adaptador para que escuche en un puerto diferente. Comprueba que el cliente funciona correctamente con el nuevo puerto.
- E 2j.09** Elimina el puerto de la especificación del endpoint en `Server.config` y comprueba que servidor y cliente siguen funcionando. Ahora puedes ejecutar dos servidores en el mismo computador ¿por qué?
- E 2j.10** Añade un endpoint UDP a `Server.config`, arranca el servidor y ejecuta dos clientes que invocan el objeto utilizando cada uno un endpoint diferente. Escribe el comando `tshark`⁵ que demuestra que efectivamente un cliente utiliza transporte TCP y el otro UDP.
- E 2j.11** Añade una pausa de 3 segundos al comienzo del método `write()` del sirviente. Ejecuta dos clientes en terminales diferentes tratando de que arranquen con la mínima diferencia de tiempo posible. A la vista del resultado ¿crees que el objeto está atendiendo a los clientes de forma concurrente? ¿Por qué?
- E 2j.12** Averigua qué significa exactamente cada uno de los parámetros que aparecen en el proxy textual que imprime el servidor. Averigua cómo alterar estos parámetros mediante el API de ICE y escribe un programa que lo demuestre.
- E 2j.13** Crea una versión diferente del servidor. Éste, en lugar de imprimir el mensaje recibido en pantalla, lo enviará (añadiendo el prefijo «*redirect:*» al mensaje original) a otro objeto *Printer*. Para ello necesitarás el proxy de ese otro objeto, que puedes crear con la versión original del servidor. Utilizando el cliente original invoca a los objetos alojados en cada uno de los servidores.

⁵<http://www.wireshark.org/docs/man-pages/tshark.html>

«Hola mundo» distribuido

[Python]

En esta primera aplicación, el servidor proporciona un objeto que dispone de un único método remoto llamado `write()`. Este método imprime en la salida estándar del servidor la cadena que el cliente le pase como parámetro.

Tal como hemos visto, lo primero que necesitamos es escribir la especificación de la interfaz remota para estos objetos. El siguiente listado corresponde al fichero `Printer.ice` y contiene la interfaz *Printer* en lenguaje SLICE.

LISTADO 2P.1: Especificación SLICE para un «Hola mundo» distribuido
`Printer.ice`

```
1 module Example {  
2     interface Printer {  
3         void write(string message);  
4     };  
5 };
```

Lo más importante de este fichero es la declaración del método `write()`. El compilador de interfaces generará los stubs que incluyen una versión básica de la interfaz *Printer* en el lenguaje de programación que el programador decida. Cualquier clase que herede de esa interfaz *Printer* debería redefinir (especializar) un método `write()`, que podrá ser invocado remotamente, y que debe tener la misma signatura. De hecho, en la misma aplicación distribuida puede haber varias implementaciones del mismo interfaz en una o varias computadoras y escritos en diferentes lenguajes.

El compilador también genera los «cabos» para el cliente. Igual que antes, los clientes escritos en distintos lenguajes o sobre distintas arquitecturas podrán usar los objetos remotos que cumplan la misma interfaz.

Al igual que para los demás lenguajes soportados, es posible generar una librería de soporte con el *translator slice2py*. Sin embargo, lo habitual es Python es utilizar una función de carga dinámica llamada `loadSlice()` que acepta directamente la ruta al fichero Slice.

2p.1. Sirviente

El compilador de interfaces genera la clase *Example.Printer*. La implementación del *sirviente* debe heredar de esa interfaz, proporcionando una implementación (por sobrecarga) de los métodos especificados en la interfaz SLICE.

La implementación del sirviente (extremadamente simple) se muestra en el siguiente listado:

LISTADO 2P.2: Sirviente de la aplicación Printer
[py/Server.py](#)

```

1  class PrinterI(Example.Printer):
2      n = 0
3
4      def write(self, message, current=None):
5          print("{0}: {1}".format(self.n, message))
6          sys.stdout.flush()
7          self.n += 1

```

2p.2. Servidor

Nuestro servidor consiste principalmente en la implementación de una clase que hereda de *Ice.Application*. De ese modo se ahorra parte del trabajo de inicialización del *communicator*¹. En esta clase debemos definir el método `run()`. Veamos el código en detalle:

LISTADO 2P.3: Servidor de la aplicación Printer
[py/Server.py](#)

```

1  class Server(Ice.Application):
2      def run(self, argv):
3          broker = self.communicator()
4          servant = PrinterI()
5
6          adapter = broker.createObjectAdapter("PrinterAdapter")
7          proxy = adapter.add(servant, broker.stringToIdentity("printer1"))
8
9          print(proxy)
10         sys.stdout.flush()
11
12         adapter.activate()
13         self.shutdownOnInterrupt()
14         broker.waitForShutdown()
15
16         return 0
17
18
19 server = Server()

```

¹El *communicator* representa el *broker* de objetos del núcleo de comunicaciones: http://en.wikipedia.org/wiki/Object_request_broker

```
20 sys.exit(server.main(sys.argv))
```

En la **línea 4** se crea el sirviente (una instancia de la clase `PrinterI`). En la **línea 14** se crea un adaptador de objetos, que es el componente encargado de multiplexar entre los objetos alojados en el servidor. El adaptador requiere un endpoint —un punto de conexión a la red materializado por un protocolo (TCP o UDP), un host y un puerto. En este caso esa información se extrae de un fichero de configuración a partir del nombre del adaptador (`PrinterAdapter` en este caso).

En la **línea 7** se registra el sirviente en el adaptador mediante el método `add()`, indicando para ello la identidad que tendrá el objeto (`printer1`). En este ejemplo la identidad es un identificador bastante simple, pero lo recomendable es utilizar una secuencia globalmente única (UUID). El método `add()` devuelve una referencia al objeto distribuido recién creado, que se denomina *proxy*.

La **línea 12** corresponde con la activación del adaptador, que se ejecuta en otro hilo. A partir de ese momento el servidor puede escuchar y procesar peticiones para sus objetos. El método `waitForShutdown()` (**línea 14**) bloquea el hilo principal hasta que el comunicador sea terminado. El método `shutdownOnInterrupt()` (**línea 13**) indica a la aplicación que termine el comunicador al recibir la señal `SIGQUIT` (Control-C).

Por último, las **líneas 19–20** contienen la función `main()` en la que se crea una instancia de la clase `Server` y se invoca su método `main()`.

Con esto tenemos el servidor completo. Pero existe un problema, el cliente necesita un modo de referenciar el objeto alojado en el servidor. ICE proporciona mecanismos de localización (que veremos en capítulos posteriores) pero en este primer ejemplo haremos algo más sencillo. Existe una forma de conseguir una representación textual² del proxy del objeto —el valor devuelto por el método `add()` del adaptador. A su vez, el cliente puede generar un objeto a partir de esa representación textual que puede utilizar para contactar con el servidor.

En su forma programática, el proxy funciona como una especie de referencia o puntero para acceder al objeto remoto.

2p.3. Cliente

La aplicación cliente únicamente debe conseguir una referencia al objeto remoto e invocar el método `write()`. El cliente también se puede implementar como una especialización de la clase `Ice.Application`. El código completo del cliente aparece a continuación:

LISTADO 2P.4: Cliente de la aplicación Printer
`python/Client.py`

```
1 import Ice
2 Ice.loadSlice('Printer.ice')
```

² *stringified proxy*

```

3  import Example
4
5
6  class Client(Ice.Application):
7      def run(self, argv):
8          proxy = self.communicator().stringToProxy(argv[1])
9          printer = Example.PrinterPrx.checkedCast(proxy)
10
11         if not printer:
12             raise RuntimeError('Invalid proxy')
13
14         printer.write('Hello World!')
15
16         return 0
17
18
19  sys.exit(Client().main(sys.argv))

```

El programa acepta por línea de comandos la representación textual del proxy del objeto remoto. A partir de ella se obtiene un objeto proxy (**línea 8**). Sin embargo, esa referencia es para un proxy genérico. Para poder invocar los métodos de la interfaz *Printer* se requiere una referencia de tipo *PrinterPrx*, es decir, un proxy a un objeto remoto *Printer*.

Para lograrlo se ha de efectuar *downcasting*³ mediante el método *PrinterPrx.checkedCast()* (**línea 9**). Gracias al soporte de introspección de los objetos remotos, ICE puede comprobar que efectivamente el objeto remoto es del tipo al que tratamos de convertirlo. Esta comprobación no se realizaría si empleáramos la modalidad *uncheckedCast()*.

Una vez conseguido el proxy del tipo correcto (objeto *printer*) se puede invocar el método remoto *write()* (**línea 14**) pasando por parámetro la cadena "Hello, World!".

2p.4. Compilación

La figura 2p.1 muestra el esquema de compilación del cliente y servidor a partir del fichero de especificación de la interfaz. Los ficheros marcados en amarillo corresponden a aquellos que el programador debe escribir o completar, mientras que los ficheros generados aparecen en verde.

2p.5. Ejecutando el servidor

Si se intenta ejecutar el servidor sin argumentos se obtiene un error:

```

$ python Server.py
2012-09-19 13:37:29.284211 Server.py: error: Traceback (most recent call last):
  File "/usr/lib/pymodules/python2.7/Ice.py", line 984, in main
    status = self.doMain(args, initData)

```

³Consiste en moldear (*cast*) un puntero o referencia de una clase a una de sus subclases, asumiendo que realmente es una instancia de ese tipo: <http://en.wikipedia.org/wiki/Downcasting>.

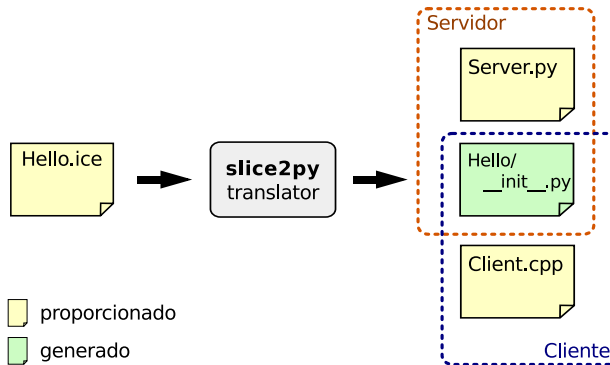


FIGURA 2P.1: Esquema de compilación de cliente y servidor en Python

```

File "/usr/lib/pymodules/python2.7/Ice.py", line 1031, in doMain
    return self.run(args)
File "Server.py", line 21, in run
    oa = ic.createObjectAdapter("PrinterAdapter")
File "/usr/lib/pymodules/python2.7/Ice.py", line 417, in createObjectAdapter
    adapter = self._impl.createObjectAdapter(name)
InitializationException: exception ::Ice::InitializationException
{
    reason = object adapter `PrinterAdapter' requires configuration
}
  
```

Como vimos en la sección 2p.2, el servidor necesita información específica que le indique los *endpoints* en los que debe escuchar el adaptador `PrinterAdapter`. Para ello, debemos proporcionar un fichero adicional (`Server.config`) cuyo contenido aparece en el siguiente listado:

LISTADO 2P.5: Servidor de la aplicación Printer
`Server.config`

```

1 PrinterAdapter.Endpoints=tcp -p 9090
  
```

Este tipo de ficheros contiene definiciones de *propiedades*, que son parejas clave=valor. La mayoría de los servicios de ICE pueden configurarse por medio de propiedades, lo que le otorga gran flexibilidad sin necesidad de recompilar el código. Esta propiedad en concreto indica que el adaptador debe utilizar un socket TCP en el puerto 9090. Al no especificar una IP, el adaptador escuchará en todas las interfaces de red del computador. El puerto tampoco es obligatorio. De no especificarlo, hubiera elegido automáticamente uno libre. Lo que sí es obligatorio es el protocolo que debe usar (en este caso TCP).

Para que el servidor cargue las propiedades del fichero de configuración, ejecute:

```

$ python Server.py --Ice.Config=Server.config
printer1 -t:tcp -h 192.168.0.12 -p 9090:tcp -h 10.1.1.10 -p 9090
  
```

En esta ocasión el programa arranca correctamente y queda ocupando la shell como corresponde a cualquier servidor. Lo que aparece en consola es, como se indicó anteriormente, la representación textual del proxy para el objeto distribuido⁴. La línea contiene varios datos:

- La identidad del objeto: `'printer1'`.
- El tipo de proxy (`'-t'`), que corresponde a *twoway*. Existen otros tipos que implican semánticas de llamada diferentes: `'-o'` para *oneway*, `'-d'` para *data-gram*, etc.
- Una lista de endpoints separados con el carácter `':'`, que corresponden con sockets, es decir, un protocolo, una dirección IP (parámetro `'-h'`) y un puerto (parámetro `'-p'`).

Concretamente, el adaptador de nuestro servidor escucha en el puerto TCP de dos interfaces de red puesto que el fichero de configuración no lo limita a una interfaz concreta.

2p.6. Ejecutando el cliente

Para ejecutar el cliente debemos indicar el proxy del objeto remoto en línea de comandos, precisamente el dato que imprime el servidor. Nótese que se debe escribir entre comillas para que la shell lo interprete como un único parámetro:

```
$ python Client.py "printer1 -t:tcp -h 192.168.0.12 -p 9090"
```

El programa se ejecuta y retorna inmediatamente. El servidor mostrará por consola la cadena `"Hello, world!"` como corresponde a la implementación del método `write()` del sirviente.

2p.7. Ejercicios

- E 2p.01** Modifica el cliente para que acepte una cadena por línea de comandos en lugar de enviar siempre `"Hello World"`. Comprueba que funciona.
- E 2p.02** Ejecuta varios clientes que envían cadenas distintas.
- E 2p.03** Con la herramienta más simple que conozcas, comprueba que el servidor efectivamente está escuchando en el puerto especificado.
- E 2p.04** Ejecuta servidor y cliente en computadores diferentes.
- E 2p.05** Modifica el sirviente para que acepte una cadena de texto en el constructor y la imprima junto con cada mensaje entrante.
- E 2p.06** Modifica el servidor creando varios sirvientes `PrinterI` y añadiéndolos al adaptador `PrinterAdapter`. Imprime los proxies de cada uno de ellos. Invócalos desde distintos clientes. Ahora escribe un cliente que crea

⁴<http://doc.zeroc.com/display/Ice/Proxy+and+Endpoint+Syntax>

varios objetos, pero que utiliza la misma instancia del sirviente para todos ellos.

- E 2p.07** Intenta ejecutar el servidor en dos terminales distintos en el mismo computador ¿qué ocurre?
- E 2p.08** Edita el fichero `Server.config` y modifica el endpoint del adaptador para que escuche en un puerto diferente. Comprueba que el cliente funciona correctamente con el nuevo puerto.
- E 2p.09** Elimina el puerto de la especificación del endpoint en `Server.config` y comprueba que servidor y cliente siguen funcionando. Ahora puedes ejecutar dos servidores en el mismo computador ¿por qué?
- E 2p.10** Añade un endpoint UDP a `Server.config`, arranca el servidor y ejecuta dos clientes que invocan el objeto utilizando cada uno un endpoint diferente. Escribe el comando `tshark`⁵ que demuestra que efectivamente un cliente utiliza transporte TCP y el otro UDP.
- E 2p.11** Añade una pausa de 3 segundos al comienzo del método `write()` del sirviente. Ejecuta dos clientes en terminales diferentes tratando de que arranquen con la mínima diferencia de tiempo posible. A la vista del resultado ¿crees que el objeto está atendiendo a los clientes de forma concurrente? ¿Por qué?
- E 2p.12** Averigua qué significa exactamente cada uno de los parámetros que aparecen en el proxy textual que imprime el servidor. Averigua cómo alterar estos parámetros mediante el API de ICE y escribe un programa que lo demuestre.
- E 2p.13** Crea una versión diferente del servidor. Éste, en lugar de imprimir el mensaje recibido en pantalla, lo enviará (añadiendo el prefijo «*redirect:*» al mensaje original) a otro objeto *Printer*. Para ello necesitarás el proxy de ese otro objeto, que puedes crear con la versión original del servidor. Utilizando el cliente original invoca a los objetos alojados en cada uno de los servidores.

⁵<http://www.wireshark.org/docs/man-pages/tshark.html>

Gestión de aplicaciones distribuidas

[C++]

En este capítulo veremos el uso básico de `IceGrid`, un servicio incluido en `ZeroC ICE` que facilita la gestión, depuración y despliegue de aplicaciones distribuidas en grid. Además se introduce `IcePatch2`, el servicio de despliegue de software. Juntos permiten aplicar procedimientos avanzados sin necesidad de escribir ni una sola línea de código. Una de las características más importantes que proporciona es la *transparencia de localización*, un concepto clave en cualquier sistema distribuido.

3c.1. Introducción

En el capítulo 2c vimos cómo desarrollar y ejecutar una aplicación distribuida básica. El servidor imprime por pantalla la versión textual del proxy para el objeto que aloja y el cliente la utiliza para obtener a su vez el objeto proxy con el que invocar al objeto remoto. Sin embargo, desde el punto de vista de la gestión, resulta muy incómodo el hecho de que los clientes deban conocer los detalles de direccionamiento de los objetos.

En una aplicación de medio o gran tamaño se utilizan cientos o miles de servidores y objetos. Es obvio que se requiere un método más potente que pueda escalar sin problema. En este capítulo utilizaremos exactamente el mismo programa del capítulo 2c y veremos cómo desplegar, arrancar y gestionar la aplicación (formada por un cliente y dos servidores) de un modo mucho más sistemático. Todo ello es posible gracias a los proxies indirectos (vea § 1.3.3.3).

3c.2. IceGrid

ICE incluye un conjunto de servicios cuidadosamente diseñados que cooperan para proporcionar una gran variedad de prestaciones avanzadas. Uno de los servicios más importantes es `IceGrid`.

IceGrid proporciona una gran variedad de funcionalidades para gestión remota de la aplicación distribuida, activación automática (implícita) de objetos, balanceo de carga y transparencia de localización.

3c.2.1. Componentes de IceGrid

IceGrid depende de una base de datos compartida llamada *IceGrid Registry*. Contiene información sobre los objetos remotos conocidos por el sistema distribuido, las aplicaciones actualmente desplegadas, los nodos de cómputo disponibles y algunos otros datos. El *Registry* es un componente clave en el sistema y sólo puede existir una instancia en cada aplicación distribuida. Se puede decir que el *Registry* es el que determina qué objetos forman parte de la aplicación. Puede haber varios *Registry* en ejecución en el mismo grid, pero corresponderían a aplicaciones distribuidas diferentes.

Además, IceGrid requiere que cada nodo de cómputo asociado al sistema ejecute un servicio llamado *IceGrid Node*. Ésta es la forma en la que IceGrid puede determinar qué computadores están disponibles para la ejecución de componentes de la aplicación.

Por último, IceGrid incluye un par de herramientas de administración que se pueden utilizar como interfaz con el usuario para controlar todas sus características. Existe una aplicación en línea de comando llamada *icegridadmin* y otra con interfaz gráfica llamada *icegridgui*. Usaremos la segunda en este capítulo.

IceGrid maneja algunos conceptos que es importante aclarar, debido a que no corresponden exactamente con el significado habitual:

Nodo

Corresponde con una instancia de *IceGrid Node*. Identifica un «nodo lógico», es decir, no tiene porqué corresponder unívocamente con un computador, pudiendo haber más de un *IceGrid Node* ejecutándose sobre un mismo computador, que a su vez pueden estar vinculados al mismo *Registry* o a distintos.

Servidor

Identifica, mediante un nombre único, a un programa que se ejecutará en un nodo. Incluye los atributos y propiedades que puedan ser necesarios para su configuración. Nótese que el programa a ejecutar puede ser también un cliente o incluso un programa que no tiene nada que ver con ICE.

Adaptador de objetos

Incluye datos específicos de un adaptador de objetos utilizado en un servidor ICE, incluyendo endpoints, objetos bien conocidos, etc.

Aplicación

Se refiere al conjunto de servicios, objetos y sus respectivas configuración que conforman la *aplicación distribuida*. Las descripciones de aplicaciones se pueden almacenar en la base de datos de IceGrid y también se pueden exportar como ficheros XML.

3c.2.2. Configuración de IceGrid

Lo primero es configurar un conjunto de computadores que utilizaremos durante la etapa de desarrollo de la aplicación. Como mínimo necesitamos un *IceGrid Registry* y un *IceGrid node*, aunque pueden ejecutarse en un mismo nodo, de hecho, en un mismo programa. Concretamente vamos a utilizar dos nodos IceGrid, y uno de ellos ejecutará también el Registry.

La configuración de IceGrid es muy similar a la de cualquier otra aplicación de ICE. Debemos proporcionar un conjunto de propiedades en forma de pares «clave=valor» en un fichero de texto plano (vea § 1.7).

Una de las propiedades que siempre debe tener la configuración de cualquier nodo IceGrid es el proxy al servicio *Locator*. Veremos la utilidad de este servicio más adelante. El *Locator* lo proporciona la instancia del Registry y tiene la estructura que se muestra en el siguiente listado:

LISTADO 3C.1: Configuración del nodo 1 (fragmento)

[icegrid/node1.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

La dirección IP 127.0.0.1 tendría que ser reemplazada por el nombre o la dirección IP del computador que ejecuta el Registry. En este ejemplo mínimo, todos los componentes estarán en el mismo computador.



Recuerda que la dirección IP 127.0.0.1 es una dirección especial que se asigna a la interfaz *loopback*. Entre otras cosas permite comunicar servidores y clientes que se ejecutan en el mismo computador sin necesidad de una NIC. Todo computador que ejecute la pila de protocolos TCP/IP tiene esta interfaz con esa dirección.

Cada nodo debe proporcionar un nombre identificativo y un directorio en el que almacenar la base de datos del nodo. El programa *icegridnode* es una aplicación ICE convencional, como nuestro servidor de impresión. Por tanto, se deben proporcionar los *endpoints* para el adaptador de objetos del nodo IceGrid.

LISTADO 3C.2: Configuración del nodo 1 (continuación)

[icegrid/node1.config](#)

```
IceGrid.Node.Name=node1
IceGrid.Node.Data=/tmp/db/node1
IceGrid.Node.Endpoints=tcp
```

Eso es todo lo que se necesita para configurar un nodo IceGrid normal. Pero necesitamos un nodo configurado especialmente para ejecutar también un Registry. Para ello utilizamos la siguiente propiedad:

LISTADO 3C.3: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Node.CollocateRegistry=1
```

Cuando esta propiedad tiene un valor distinto de cero, una instancia de IceGrid Registry se añade al adaptador que se especifique (mediante propiedades específicas). El nodo tiene tres adaptadores diferentes dado que se usan para propósitos distintos con privilegios específicos. El adaptador «Client» se utiliza para registrar el servicio Locator, por lo que debería estar configurado para usar el puerto estándar 4061. También debemos indicar un directorio para la base de datos del Registry (línea 4):

IANA

La [IANA](#) estandariza los puertos TCP y UDP para servicios comunes.

LISTADO 3C.4: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
1 IceGrid.Registry.Client.Endpoints=tcp -p 4061
2 IceGrid.Registry.Server.Endpoints=tcp
3 IceGrid.Registry.Internal.Endpoints=tcp
4 IceGrid.Registry.Data=/tmp/db/registry
```

El acceso al Registry está controlado por un verificador de permisos —un servicio simple para autenticación y autorización de usuarios. Existe una implementación hueca (*dummy*) que se suele utilizar durante el desarrollo de la aplicación:

LISTADO 3C.5: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.PermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

Por último, se debe configurar una ruta absoluta al fichero de plantillas `templates.xml` que se distribuye conjuntamente con ICE.

LISTADO 3C.6: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.DefaultTemplates=/usr/share/Ice-3.6.1/templates.xml
```



El fichero `templates.xml` puede estar en una ruta diferente a la indicada en función de la versión de ICE instalada en el sistema.

Como mencionamos anteriormente, utilizaremos dos nodos. Por eso necesitamos proporcionar otro fichero de configuración. En este caso sólo se necesitan las propiedades esenciales: El proxy al Locator, en nombre del nodo, el directorio de la bases de datos y los endpoints del adaptador del nodo:

LISTADO 3C.7: Configuración de un nodo IceGrid convencional (sin Registry)
[icegrid/node2.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
IceGrid.Node.Name=node2
IceGrid.Node.Data=/tmp/db/node2
IceGrid.Node.Endpoints=tcp
```

Recuerda que si está ejecutando los dos nodos en computadores diferentes, debes cambiar la dirección IP por la dirección que tenga el computador en la que se ejecuta el Registry.

Se requiere un fichero de configuración adicional para las aplicaciones que deseen utilizar el Locator:

LISTADO 3C.8: Configuración del Locator
[icegrid/locator.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

3c.2.3. Arrancando IceGrid

El nodo IceGrid normalmente se ejecuta como un *demonio*, es decir, un servicio que arranca automáticamente al conectar cada computador asociado al sistema distribuido y que se ejecuta en *background*. Sin embargo, en este caso y con propósitos didácticos vamos a proceder a arrancarlos manualmente y en primer plano.

Abre una ventana de terminal en el directorio `icegrid`, crea los directorios para las bases de datos del nodo y el registro, y lanza `icegridnode` con la configuración del nodo 1:

```
$ mkdir -p /tmp/db/node1
$ mkdir -p /tmp/db/registry
$ icegridnode --Ice.Config=node1.config
```

Ese terminal queda ocupado con la ejecución del nodo. Déjalo así, más adelante podrás pulsar Control-c para pararlo.

Abre una segunda ventana de terminal en el mismo directorio, crea el directorio para la base de datos y ejecuta otra instancia de `icegridnode`, esta vez con la configuración del nodo 2:

```
$ mkdir -p /tmp/db/node2
$ icegridnode --Ice.Config=node2.config
```

Si todo funciona correctamente, tendrá el mismo efecto y el terminal quedará ocupado.

Ahora procedemos a conectar con el Registry utilizando la aplicación `icegridgui` con la configuración del Locator, que puedes lanzar en un tercer terminal:

```
$ icegridgui
```

Deberías ver una ventana similar a la de la figura 3c.1.

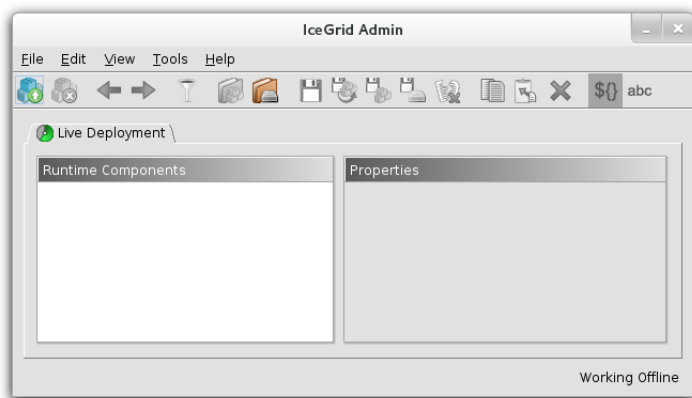



FIGURA 3C.1: IceGrid Admin: Ventana inicial

3c.3. Creando una aplicación distribuida

A continuación crearemos una nueva aplicación con `icegridgui`. Pero primero necesitamos conectar con el Registry pulsando el botón *Log into an IceGrid Registry* situado arriba a la izquierda, el primero de la barra de herramientas .

Aparecerá una ventana con una lista de conexiones como la de la figura 3c.2. Pulsa **New connection**, lo que inicia un pequeño asistente para crear una nueva conexión. Ese asistente tiene los siguientes pasos:

1. Tipo de conexión (figura 3c.3). Elige *Direct connection*.
2. Nombre de la instancia (figura 3c.4). Escribe «IceGrid».
3. Información de direccionamiento (figura 3c.5). Elige *An endpoint string*.
4. Endpoint del Registro IceGrid (figura 3c.6). Escribe «tcp -p 4061».
5. Credenciales (figura 3c.7). Escribe «user» y «pass». Por supuesto, en una aplicación en producción, se requeriría una contraseña o un mecanismo de autenticación equivalente. Recuerda que el registro está configurado con un verificador de permisos *dummy* (falso).

Una vez terminado el asistente, la lista contendrá la conexión recién configurada (figura 3c.8). Pulse **Connect**.

Cuando la aplicación ha establecido el acceso al registro debería aparecer una ventana similar a la de la figura 3c.9. Esta ventana muestra una vista lógica actualizada del sistema distribuido. Puedes ver todos los nodos involucrados en el sistema, es decir, los que están configurados con el Locator asociado al registro al que has conectado. Puedes pulsar los iconos que representan esos nodos y obtener datos actualizados con información sobre su sistema operativo, su carga, etc.

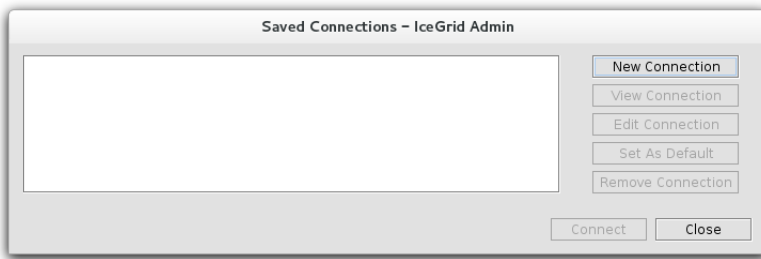


FIGURA 3C.2: IceGrid Admin: Conexiones guardadas

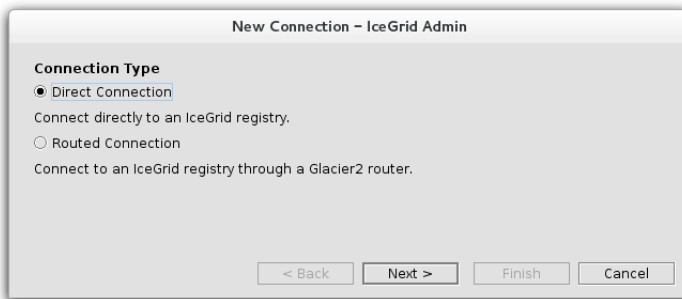


FIGURA 3C.3: IceGrid Admin: Nueva conexión

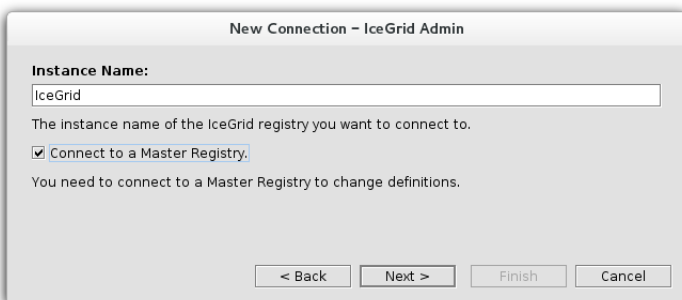


FIGURA 3C.4: IceGrid Admin: Nombre de la instancia de IceGrid



FIGURA 3C.5: IceGrid Admin: Elegir tipo de dirección



FIGURA 3C.6: IceGrid Admin: Endpoint del Registry



FIGURA 3C.7: IceGrid Admin: Credenciales

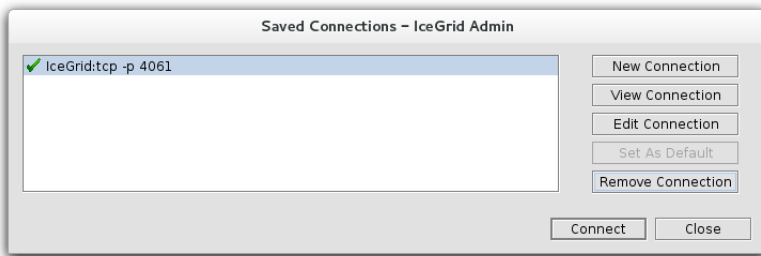
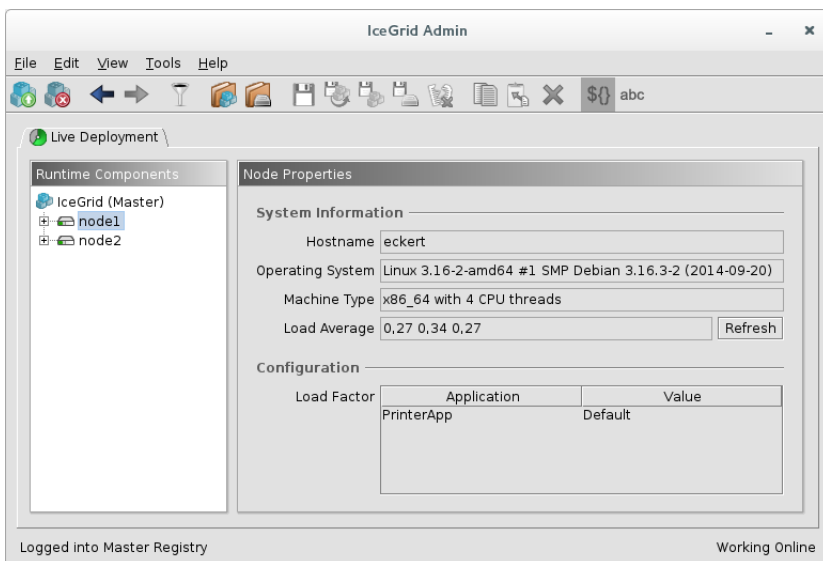


FIGURA 3C.8: IceGrid Admin: Conexión creada

FIGURA 3C.9: IceGrid Admin: Ventana *Live Deployment*

Para crear una nueva aplicación selecciona la opción de menú *File, New, Application with Default Templates from Registry* (figura 3c.10). Aparecerá una nueva pestaña llamada *NewApplication*. Edita el nombre de la aplicación, escribe «PrinterApp» y pulsa **Apply**.

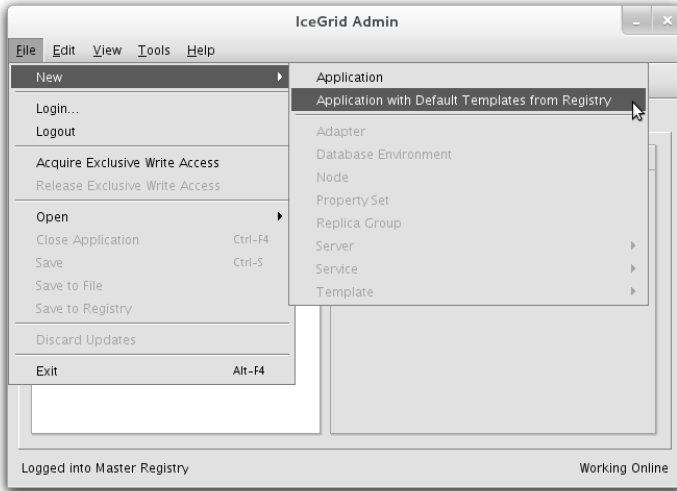


FIGURA 3c.10: IceGrid Admin: Creación de aplicaciones

Abre el menú contextual (botón derecho del ratón) de la carpeta *Nodes* situado en el panel izquierdo (figura 3c.11). Crea un nuevo nodo, cambia su nombre a *node1* y pulsa **Apply**. Crea otro nodo, nómbralo como *node2* y pulsa **Apply** de nuevo. Ahora debería haber dos nodos con los mismos nombres que los que aparecen en la pestaña *Live Deployment*. Comprueba que efectivamente los nombre corresponden (incluyendo mayúsculas y minúsculas).

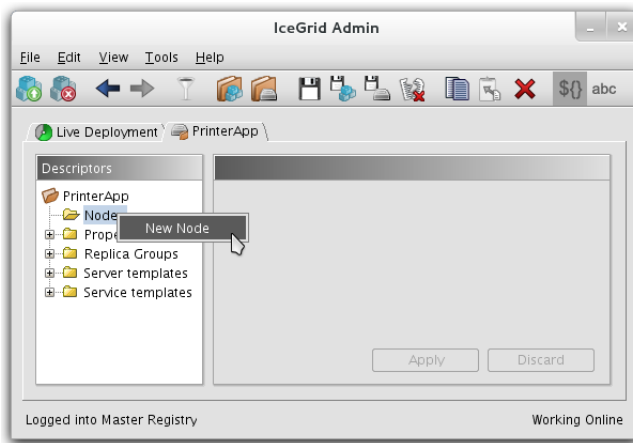


FIGURA 3c.11: IceGrid Admin: Creación de un nodo

3c.4. Despliegue de aplicaciones con IcePatch2

En el menú contextual del nodo `node1` pulsa *New Server from Template* (ver figura 3c.12). Selecciona la plantilla *IcePatch2* en la lista desplegable de la parte superior del diálogo (ver figura 3c.13). En el parámetro *directory* introduce la ruta absoluta del directorio que contiene los binarios de la aplicación «hola mundo» y pulsa **Apply**.

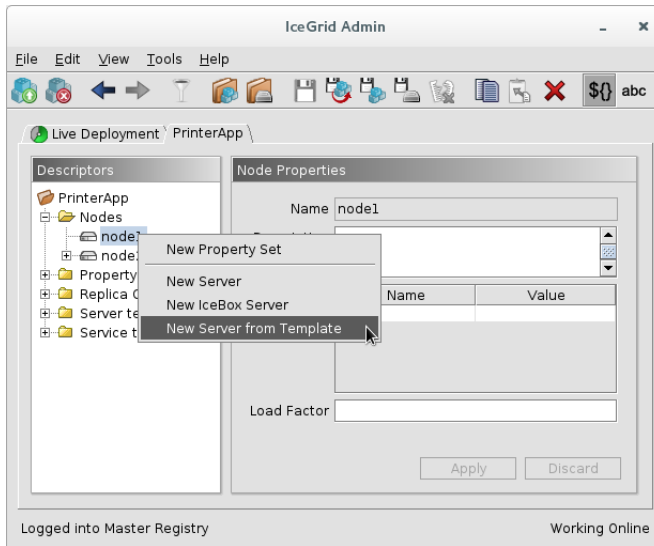


FIGURA 3c.12: IceGrid Admin: Añadiendo el servidor IcePatch2

Ahora tienes una instancia del servicio IcePatch2 configurada en el nodo `node1`. IceGrid utiliza este servicio para desplegar el software necesario a cada nodo del sistema distribuido. Puede haber varias instancias del servicio en una misma aplicación, pero es posible configurar una de ellas como la instancia por defecto. Eso es lo que vamos a hacer con la instancia recién creada. Pulsa *PrinterApp* arriba a la izquierda y en el desplegable *IcePatch2 Proxy* elije la única que aparece (ver figura 3c.14).

3c.5. Instanciación del servidor

Es hora de crear los descriptores para nuestros programas: el cliente y el servidor de «hola mundo».

- Crear un nuevo servidor y cambia el nombre por defecto a `PrinterServer1`.
- Edita la propiedad *Path to Executable* y escribe `./Server y`,
- En la propiedad *Working Directory* escribe `${application.distrib}` igual que para el cliente.

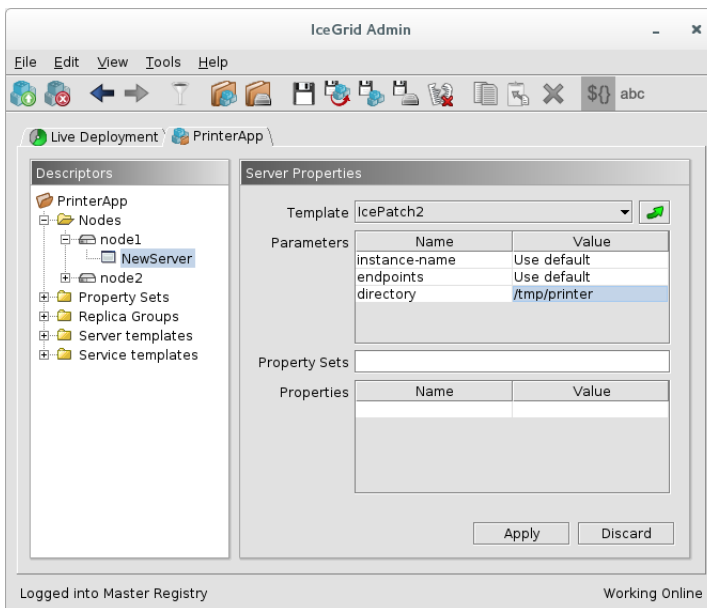


FIGURA 3C.13: IceGrid Admin: Configuración del servicio IcePatch2

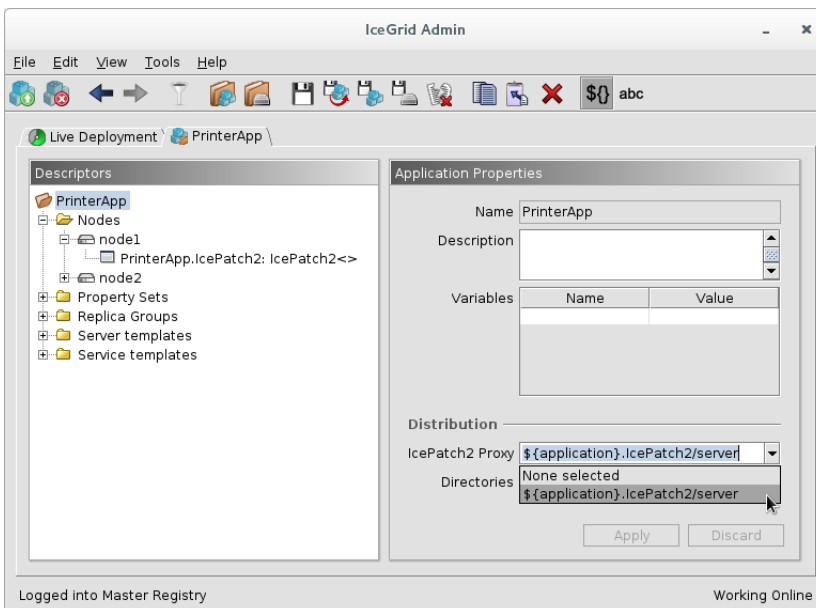


FIGURA 3C.14: IceGrid Admin: Configuración del proxy IcePatch2 de la aplicación

Tal como vimos en la sección 2c.5, el adaptador de objetos del servidor requiere configuración adicional, en concreto, debemos proporcionarle los *endpoints* en los que escuchar. Sin embargo, en este caso la configuración de ese adaptador la fijaremos a través de IceGrid.

En el menú contextual de `PrinterServer1` elige *New Adapter*. Cambia el nombre por defecto a `PrinterAdapter` y pulsa **Apply**. Este nombre debe corresponder con el nombre del adaptador definido en el código del servidor mediante el método `createObjectAdapter()`.

Sin embargo, tenemos un problema: ¿Cómo sabremos si el servidor funciona correctamente? Su única función es imprimir en su salida estándar las cadenas que se le envían. Para poder obtener remotamente la salida estándar del servidor configuramos una propiedad adicional.

- Pulsa sobre `PrinterServer1` en el lado izquierdo.
- Pulsa en la tabla `Propiedades` y añade una propiedad llamada `Ice.StdOut`.
- Como valor escribe `${application.distrib}/server-out.txt` y pulsa **Apply**.

Debería quedar como la figura 3c.15. La variable `${application.distrib}` se refiere al directorio en el que se desplegará la aplicación en el nodo. No es necesario conocer cuál será la ruta exacta. Eso es una decisión interna de `IcePatch2`.

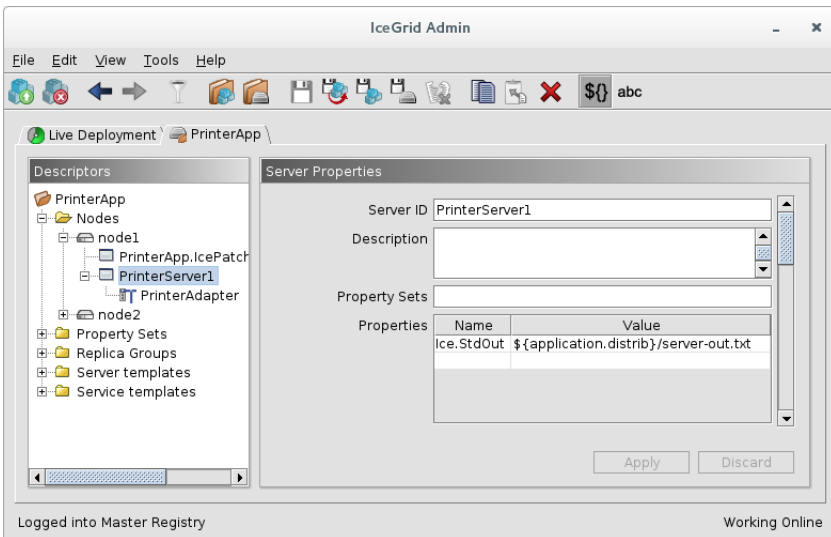


FIGURA 3C.15: IceGrid Admin: Instancia de la aplicación servidor

3c.5.1. Instanciando un segundo servidor


Crearemos ahora una segunda instancia del servidor de impresión en otro nodo. Sigue los mismos pasos que en la sección anterior con dos diferencias:

- Crea un nodo llamado `node2`.

- Crea un servidor llamado `PrinterServer2`
- En la propiedad *Path to Executable* escribe `./Server-UUID`

La única diferencia es que este servidor contendrá un objeto cuya identidad se genera aleatoriamente (ver método `addWithUUID()` del adaptador de objetos).

3c.6. Ejecutando la aplicación

Para empezar, guarda la aplicación recién definida en el registro pulsando el botón *Save to registry* 

3c.6.1. Despliegue de la aplicación

Antes de ejecutar los binarios que componen la aplicación es necesario copiarlos (desplegarlos) a sus respectivos nodos. Esa tarea corre a cargo del servicio IcePatch2 que acabas de configurar. Sin embargo, antes de poder efectuar el despliegue debes preparar el directorio con los ficheros correspondientes. Esto se consigue ejecutando el comando `icepatch2calc` en el mismo directorio que indicamos en la configuración del IcePatch2.

Suponiendo que la aplicación «hola mundo» ya compilada está en el directorio `/tmp/printer` ejecuta:

```
$ icepatch2calc /tmp/printer
```



Recuerda que en la figura 3c.13 se indicó que los binarios de la aplicación están en `/tmp/printer`, pero si están en otra parte (por ejemplo el subdirectorio `ice-hello/cpp`, debes modificar la propiedad `directory` de IcePatch2 de acuerdo con la situación real.

Ahora selecciona la solapa *Live Deployment* y elige la opción *Tools, Application, Patch Distribution* tal como muestra la figura 3c.16.

Si todo ha ido bien, la barra de estado mostrará el mensaje «Patching application ‘PrinterApp’... done».

3c.6.2. Ejecutando los servidores

Haz doble click en `node1` en la pestaña *Live Deployment*. Aparecerá su «contenido», concretamente el servidor `PrinterServer1`. Abre el menú contextual de `PrinterServer1` y pulsa *Start*. Un símbolo de «play» aparecerá sobre él y la barra de estado mostrará el mensaje «Starting server ‘PrinterServer1’... done».

Ahora que el servidor está en marcha puedes ver su salida estándar. Sólo tienes que abrir de nuevo el menú contextual de `PrinterServer1` y pulsar *Retrieve stdout*. Aparecerá una ventana similar a la de la figura 3c.17.

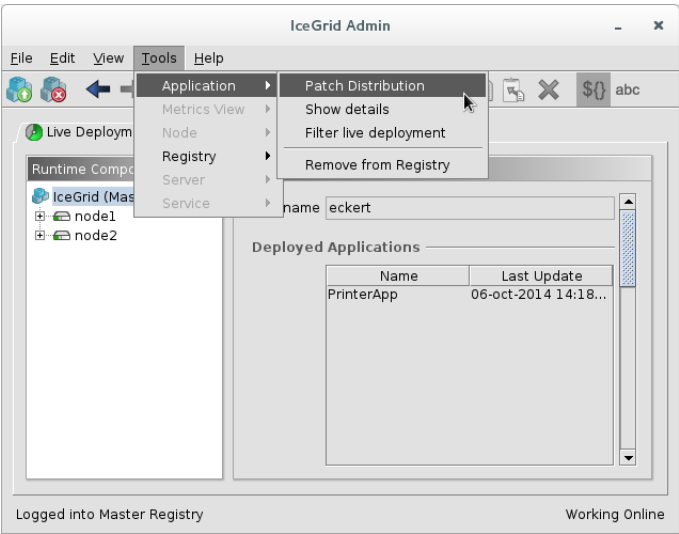


FIGURA 3C.16: IceGrid Admin: Desplegando la aplicación

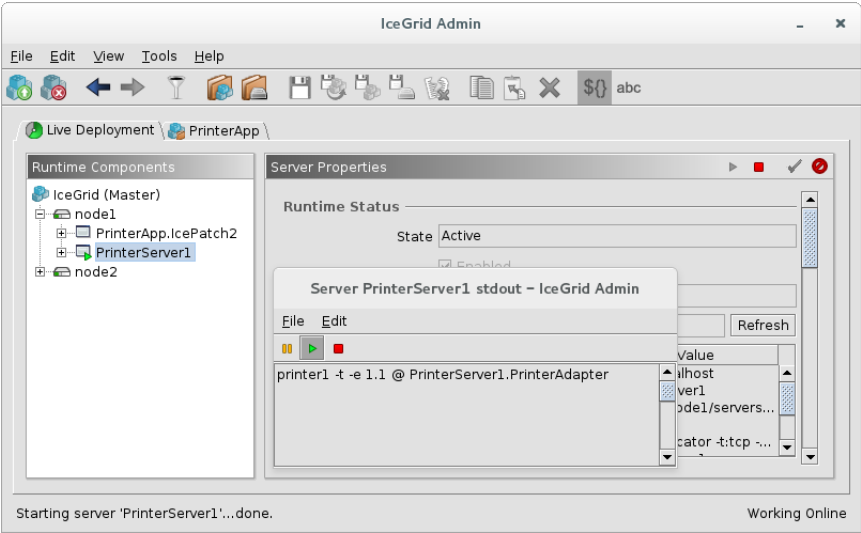


FIGURA 3C.17: IceGrid Admin: Salida estándar del servidor

Como recordarás, el servidor de impresión escribe en consola la representación textual del proxy de su objeto (ver § 2c.5). Sin embargo, la salida que ahora muestra el servidor es bastante distinta:

```
printer1 -t @ PrinterServer1.PrinterAdapter
```

El proxy que se mostraba cuando ejecutamos el servidor desde un terminal era un *proxy directo* (§ 1.3.3.2) porque contenía todos los detalles para contactar con el servidor sin ningún intermediario. Sin embargo, el que vemos ahora es un *proxy indirecto* (§ 1.3.3.3) y carece de información esencial como dirección IP y puerto del servidor.

A partir de un proxy indirecto, el cliente debe pedir al servicio Locator los detalles de conexión que correspondan, convirtiendo así un proxy indirecto en uno directo. El Locator buscará el adaptador `PrinterServer1.PrinterAdapter` y reemplazará el `@PrinterServer1.PrinterAdapter` por los endpoints de dicho adaptador. El adaptador debe ser el único con ese identificador público. Para entender la diferencia con el nombre del adaptador de objetos pulsa otra vez en la aplicación `PrinterApp` y después en el adaptador `PrinterAdapter` dentro de `PrinterServer1` (ver figura 3c.18).

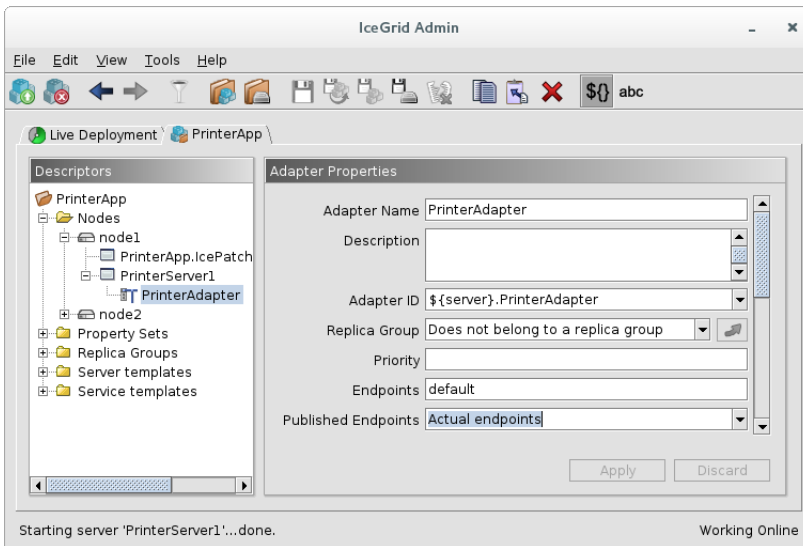


FIGURA 3C.18: IceGrid Admin: Configuración del adaptador de objetos



Nótese que el *Adapter ID* tiene un valor por defecto igual a `${server}.PrinterAdapter`. La variable `${server}` se traducirá por el nombre del servidor (`PrinterServer1` en este caso) y el resto corresponde con el nombre del adaptador de objetos. Puede parecer raro que tenga que existir un nombre interno y un identificador público. Pero esto permite crear instancias diferentes del mismo servidor.

Para ejecutar el segundo servidor, procede del mismo modo desplegando el `node2` y pulsa *Start* en el `PrinterServer2`.

3c.6.3. Ejecutando el cliente

Recuerda que el cliente acepta un argumento desde la línea de comandos: la representación textual del proxy que debe utilizar. Ahora conocemos el valor de ese parámetro, así que ya podemos ejecutar el cliente:

```
$ ./Client --Ice.Config=../icegrid/locator.config \
    "printer1 -t -e 1.1 @ PrinterServer1.PrinterAdapter"
```

Nótese el parámetro `--Ice.Config` en el que se le indica al cliente el fichero en el que aparece el valor de la propiedad `Ice.Default.Locator`. Sin esa referencia, el cliente no sabría cómo resolver proxies indirectos.

Mira la salida estándar del `PrinterServer1` (figura 3c.19). ¡Funciona!

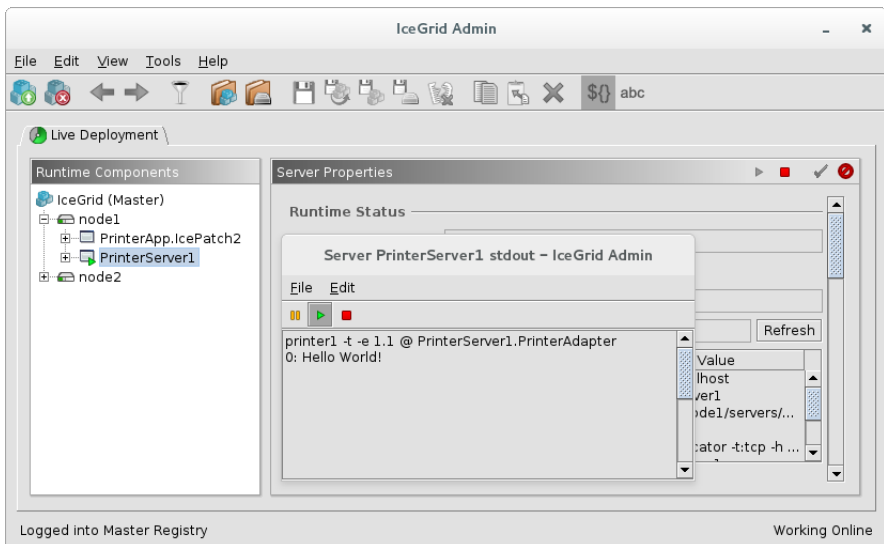


FIGURA 3c.19: IceGrid Admin: Salida estándar del servidor después de ejecutar el cliente

Sólo tenemos que usar `printer1@PrinterServer1.PrinterAdapter`. A diferencia del proxy directo, esta designación siempre será la misma, aunque el servidor se ejecute en **un nodo diferente** o el middleware elija un puerto distinto. Además,

la indirección no resulta especialmente ineficiente puesto que el cliente solo preguntará al Locator la primera vez y cacheará la respuesta. Sólo volverá a consultar al Locator si falla una invocación sobre el proxy cacheado, y todo esto de forma automática y sin la intervención del programador.



El tipo de indirección que conseguimos gracias a los proxies indirectos es lo que entendemos por **transparencia de localización**, uno de los logros más importantes en el campo de la computación distribuida heterogénea.

3c.7. Objetos bien conocidos

Algunos objetos importantes para la aplicación pueden ser incluso más fáciles de localizar. Los objetos bien conocidos pueden ser localizados simplemente por sus nombres. Es bastante sencillo conseguirlo. Ve a la pestaña `PrinterApp` y luego al adaptador `PrinterAdapter` de `PrinterServer1`. Desplaza la ventana hacia abajo hasta la tabla *Well-known objects*. Añade la identidad `printer1` y pulsa `Apply`, tal como se muestra en la figura 3c.20.

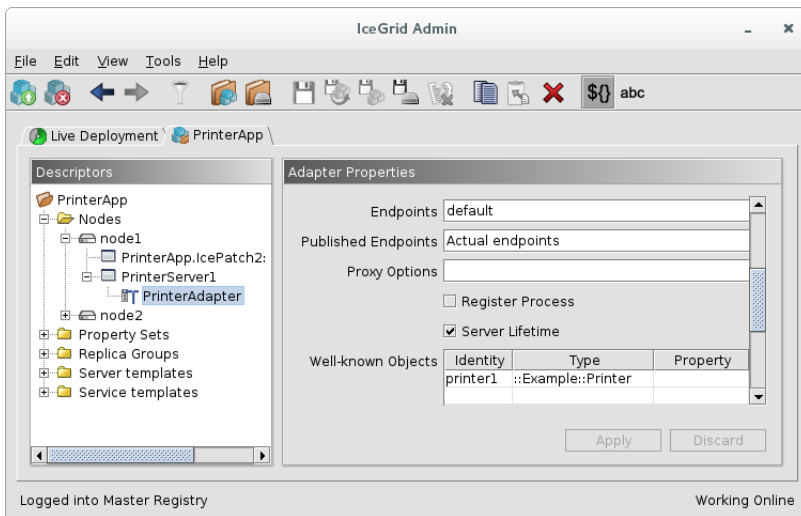


FIGURA 3C.20: IceGrid Admin: Añadiendo un objeto bien conocido

Para invocar el objeto bien conocido con el cliente simplemente debes indicar el nombre como proxy textual:

```
$ ./Client --Ice.Config=../icegrid/locator.config printer1
```

Mira la salida estándar de `PrinterServer1` y comprueba que también funciona. Una cuestión interesante es que ni siquiera ha sido necesario reiniciar el servidor. En este caso, la cadena `printer1` es otra forma de proxy indirecto. La notación

`identidad@adaptador` corresponde con un objeto registrado en un adaptador de objetos bien conocido (aquellos que tiene un *Adapter ID*), mientras que la notación `identidad` corresponde con objetos bien conocidos.

3c.8. Activación y desactivación implícita

A veces podemos ahorrar recursos parando los servidores ociosos hasta que se necesiten realmente. Los proxies indirectos son ideales para conseguirlo de forma automática. Si un cliente intenta contactar con un proxy indirecto por primera vez, hará una petición al Locator. Si falla, intentará contactar al Locator de nuevo.

El registro es capaz de usar las peticiones al Locator como solicitudes de activación implícita. Pulsa en la pestaña *PrinterApp* y luego en *PrinterServer1*. Despliega el menú *Activation mode* y selecciona *on-demand*. Ahora pulsa en **Apply** y en *Save to registry*.

Ahora ve a la pestaña *Live Deployment*, en el menú contextual de *PrinterServer1* pulsas *Stop*. Ahora vuelve a ejecutar el cliente invocando a `printer1`. El servidor `PrinterServer1` arrancará automáticamente en el momento en el que el cliente intenta conectar con él.

Además, es bastante conveniente desactivar los servicios ociosos pasado un tiempo de inactividad. Para aplicar esa posibilidad ve a la pestaña *PrinterApp*, pulsa en *PrinterServer1* y en la tabla de propiedades añade una nueva propiedad llamada `Ice.ServerIdleTime` con el valor '5'. Pulsar **Apply** y luego el botón *Save to registry*. Después de 5 segundos de inactividad, el servidor cerrará automáticamente.

3c.9. Depuración

Es fácil cometer algún error durante la configuración de la aplicación distribuido y puede resultar complicado si no se es sistemático encontrando el problema.

Si algo va mal primero debería revisar la salida de error estándar del programa que falla. Para que esta salida esté disponible desde la herramienta de administración se debe definir la propiedad `Ice.StdErr` y darle una ruta a un fichero tal que `${application.distrib}/server-err.txt`. Una vez que el programa se ejecute podrá acceder a su contenido por medio del menú contextual del nodo servidor en la opción *Retrieve stderr*. Del mismo modo se puede activar la salida estándar con la propiedad `Ice.StdOut`. Estas mismas propiedades pueden definirse en la configuración de los nodos por medio de los ficheros que aparecen en la sección 3c.2.2. Esto nos dará información extra muy útil para localizar y resolver posibles problemas.

3c.9.1. Evitando problemas con `IcePatch2`

Recuerda que si cambias y recompilas un programa debes volverlos a desplegar (ver § 3c.6.1). Sin embargo, es bastante frecuente olvidar reiniciar el propio servicio `IcePatch2`. Si el servicio no se reinicia no detectará los cambios.

Una buena solución para este problema es configurar siempre la propiedad `Ice.ServerIdleTime` en la instancia del servidor `IcePatch2` asignándole un valor pequeño.

3c.9.2. Descripción de la aplicación

Recuerda que la aplicación queda definida en un fichero XML. Toda la configuración que se realiza en la definición de la aplicación en `icegridgui` queda almacenada en un fichero bastante legible/editable manualmente. Para nuestra aplicación ese fichero (obviando las plantillas de los servicios comunes) se muestra en el listado 3c.9.

LISTADO 3c.9: Descripción de la aplicación `PrinterApp` en XML
`icegrid/printerapp-cpp.xml`

```
<node name="node1">
  <server-instance template="IcePatch2" directory="/tmp/ice-hello/cpp/dist">
    <properties>
      <property name="Ice.ServerIdleTime" value="5"/>
    </properties>
  </server-instance>
  <server id="PrinterServer1" activation="manual" exe="./Server" pwd="{application.distrib}">
    <properties>
      <property name="Ice.StdOut" value="{application.distrib}/server-out.txt"/>
    </properties>
    <adapter name="PrinterAdapter" endpoints="default" id="{server}.PrinterAdapter">
      <object identity="printer1" type="::Example::Printer"/>
    </adapter>
  </server>
</node>
<node name="node2">
```

3c.10. Receta

A continuación se listan a modo de resumen los pasos más importantes para realizar un despliegue cliente/servidor sencillo como el que se ha descrito:

- Crear un fichero de configuración para el Registry y para cada uno de los nodos del grid (§3c.2.2).
- Arrancar un `icegridnode` por cada nodo con su configuración correspondiente (§3c.2.3).
- Crear la aplicación distribuida (§3c.3).
- Crear un nodo en la aplicación para cada nodo ejecutado.
- Compilar los programas, copiar los binarios a otro directorio y ejecutar `ice-patch2calc`.
- Crear un servidor de `IcePatch2` (§3c.4) y configurarlo para desplegar el directorio del paso anterior.
- Configurar ese servidor de `IcePatch2` como el `IcePatch2` por defecto de la aplicación distribuida (figura 3c.14).

- Crear los servidores IceGrid para ambos programas (3c.5) en sus respectivos nodos.
- Desplegar la aplicación (§ 3c.6.1).
- Arrancar los servidores (§ 3c.6.2).
- Ejecutar el cliente (§ 3c.6.3).

3c.11. Ejercicios

- E 3c.01** Invoca el servidor desplegado con IceGrid ejecutando el cliente desde otro servidor IceGrid.
- E 3c.02** Modifica el código del cliente para que invoque a los dos servidores.
- E 3c.03** El servidor del node2 tiene una identidad aleatoria. Averigua cómo fijar la identidad del objeto por medio de una propiedad. Cambia el programa que ejecutan ambos servidores para ejecutar la nueva versión (ambos nodos el mismo programa) e indica por medio de una propiedad que sus identidades son respectivamente «printer1» y «printer2».
- E 3c.04** Define nodos adicionales, crea una plantilla¹ para el servidor Printer-Server. Arranca nodos adicionales y, usando la plantilla, instancia varias impresoras (objetos *Printer*) en ellos. Invoca esas impresoras con el cliente original desde la consola. Consulta el ejemplo `demopy/IceGrid/simple` de la distribución de ICE².
- E 3c.05** Explora las posibilidades de `icegridadmin`³, la herramienta de administración en línea de comandos, para obtener información y modificar la configuración de la aplicación. ¿Qué comando debes introducir para obtener la lista de nodos? ¿Y la lista de servidores?
- E 3c.06** Repite el tutorial de este capítulo usando un computador diferente para cada nodo. Esos computadores pueden ser virtuales (se aconseja `VirtualBox`).

¹<http://doc.zeroc.com/display/Ice/IceGrid+Templates>

²<http://www.zeroc.com/download/Ice/3.5/Ice-3.5.0-demos.tar.gz>

³<http://doc.zeroc.com/display/Ice/icegridadmin+Command+Line+Tool>

Gestión de aplicaciones distribuidas

[Java]

En este capítulo veremos el uso básico de `IceGrid`, un servicio incluido en `ZeroC ICE` que facilita la gestión, depuración y despliegue de aplicaciones distribuidas en grid. Además se introduce `IcePatch2`, el servicio de despliegue de software. Juntos permiten aplicar procedimientos avanzados sin necesidad de escribir ni una sola línea de código. Una de las características más importantes que proporciona es la *transparencia de localización*, un concepto clave en cualquier sistema distribuido.

3j.1. Introducción

En el capítulo 2j vimos cómo desarrollar y ejecutar una aplicación distribuida básica. El servidor imprime por pantalla la versión textual del proxy para el objeto que aloja y el cliente la utiliza para obtener a su vez el objeto proxy con el que invocar al objeto remoto. Sin embargo, desde el punto de vista de la gestión, resulta muy incómodo el hecho de que los clientes deban conocer los detalles de direccionamiento de los objetos.

En una aplicación de medio o gran tamaño se utilizan cientos o miles de servidores y objetos. Es obvio que se requiere un método más potente que pueda escalar sin problema. En este capítulo utilizaremos exactamente el mismo programa del capítulo 2j y veremos cómo desplegar, arrancar y gestionar la aplicación (formada por un cliente y dos servidores) de un modo mucho más sistemático. Todo ello es posible gracias a los proxies indirectos (vea § 1.3.3.3).

3j.2. IceGrid

ICE incluye un conjunto de servicios cuidadosamente diseñados que cooperan para proporcionar una gran variedad de prestaciones avanzadas. Uno de los servicios más importantes es `IceGrid`.

IceGrid proporciona una gran variedad de funcionalidades para gestión remota de la aplicación distribuida, activación automática (implícita) de objetos, balanceo de carga y transparencia de localización.

3j.2.1. Componentes de IceGrid

IceGrid depende de una base de datos compartida llamada *IceGrid Registry*. Contiene información sobre los objetos remotos conocidos por el sistema distribuido, las aplicaciones actualmente desplegadas, los nodos de cómputo disponibles y algunos otros datos. El *Registry* es un componente clave en el sistema y sólo puede existir una instancia en cada aplicación distribuida. Se puede decir que el *Registry* es el que determina qué objetos forman parte de la aplicación. Puede haber varios *Registry* en ejecución en el mismo grid, pero corresponderían a aplicaciones distribuidas diferentes.

Además, IceGrid requiere que cada nodo de cómputo asociado al sistema ejecute un servicio llamado *IceGrid Node*. Ésta es la forma en la que IceGrid puede determinar qué computadores están disponibles para la ejecución de componentes de la aplicación.

Por último, IceGrid incluye un par de herramientas de administración que se pueden utilizar como interfaz con el usuario para controlar todas sus características. Existe una aplicación en línea de comando llamada *icegridadmin* y otra con interfaz gráfica llamada *icegridgui*. Usaremos la segunda en este capítulo.

IceGrid maneja algunos conceptos que es importante aclarar, debido a que no corresponden exactamente con el significado habitual:

Nodo

Corresponde con una instancia de *IceGrid Node*. Identifica un «nodo lógico», es decir, no tiene porqué corresponder unívocamente con un computador, pudiendo haber más de un *IceGrid Node* ejecutándose sobre un mismo computador, que a su vez pueden estar vinculados al mismo *Registry* o a distintos.

Servidor

Identifica, mediante un nombre único, a un programa que se ejecutará en un nodo. Incluye los atributos y propiedades que puedan ser necesarios para su configuración. Nótese que el programa a ejecutar puede ser también un cliente o incluso un programa que no tiene nada que ver con ICE.

Adaptador de objetos

Incluye datos específicos de un adaptador de objetos utilizado en un servidor ICE, incluyendo endpoints, objetos bien conocidos, etc.

Aplicación

Se refiere al conjunto de servicios, objetos y sus respectivas configuración que conforman la *aplicación distribuida*. Las descripciones de aplicaciones se pueden almacenar en la base de datos de IceGrid y también se pueden exportar como ficheros XML.

3j.2.2. Configuración de IceGrid

Lo primero es configurar un conjunto de computadores que utilizaremos durante la etapa de desarrollo de la aplicación. Como mínimo necesitamos un *IceGrid Registry* y un *IceGrid node*, aunque pueden ejecutarse en un mismo nodo, de hecho, en un mismo programa. Concretamente vamos a utilizar dos nodos IceGrid, y uno de ellos ejecutará también el Registry.

La configuración de IceGrid es muy similar a la de cualquier otra aplicación de ICE. Debemos proporcionar un conjunto de propiedades en forma de pares «clave=valor» en un fichero de texto plano (vea § 1.7).

Una de las propiedades que siempre debe tener la configuración de cualquier nodo IceGrid es el proxy al servicio *Locator*. Veremos la utilidad de este servicio más adelante. El *Locator* lo proporciona la instancia del Registry y tiene la estructura que se muestra en el siguiente listado:

LISTADO 3j.1: Configuración del nodo 1 (fragmento)
[icegrid/node1.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

La dirección IP 127.0.0.1 tendría que ser reemplazada por el nombre o la dirección IP del computador que ejecuta el Registry. En este ejemplo mínimo, todos los componentes estarán en el mismo computador.



Recuerda que la dirección IP 127.0.0.1 es una dirección especial que se asigna a la interfaz *loopback*. Entre otras cosas permite comunicar servidores y clientes que se ejecutan en el mismo computador sin necesidad de una NIC. Todo computador que ejecute la pila de protocolos TCP/IP tiene esta interfaz con esa dirección.

Cada nodo debe proporcionar un nombre identificativo y un directorio en el que almacenar la base de datos del nodo. El programa *icegridnode* es una aplicación ICE convencional, como nuestro servidor de impresión. Por tanto, se deben proporcionar los *endpoints* para el adaptador de objetos del nodo IceGrid.

LISTADO 3j.2: Configuración del nodo 1 (continuación)
[icegrid/node1.config](#)

```
IceGrid.Node.Name=node1
IceGrid.Node.Data=/tmp/db/node1
IceGrid.Node.Endpoints=tcp
```

Eso es todo lo que se necesita para configurar un nodo IceGrid normal. Pero necesitamos un nodo configurado especialmente para ejecutar también un Registry. Para ello utilizamos la siguiente propiedad:

LISTADO 3J.3: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Node.CollocateRegistry=1
```

Cuando esta propiedad tiene un valor distinto de cero, una instancia de IceGrid Registry se añade al adaptador que se especifique (mediante propiedades específicas). El nodo tiene tres adaptadores diferentes dado que se usan para propósitos distintos con privilegios específicos. El adaptador «Client» se utiliza para registrar el servicio Locator, por lo que debería estar configurado para usar el puerto estándar 4061. También debemos indicar un directorio para la base de datos del Registry (línea 4):

IANA

La [IANA](#) estandariza los puertos TCP y UDP para servicios comunes.

LISTADO 3J.4: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
1 IceGrid.Registry.Client.Endpoints=tcp -p 4061
2 IceGrid.Registry.Server.Endpoints=tcp
3 IceGrid.Registry.Internal.Endpoints=tcp
4 IceGrid.Registry.Data=/tmp/db/registry
```

El acceso al Registry está controlado por un verificador de permisos —un servicio simple para autenticación y autorización de usuarios. Existe una implementación hueca (*dummy*) que se suele utilizar durante el desarrollo de la aplicación:

LISTADO 3J.5: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.PermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

Por último, se debe configurar una ruta absoluta al fichero de plantillas `templates.xml` que se distribuye conjuntamente con ICE.

LISTADO 3J.6: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.DefaultTemplates=/usr/share/Ice-3.6.1/templates.xml
```



El fichero `templates.xml` puede estar en una ruta diferente a la indicada en función de la versión de ICE instalada en el sistema.

Como mencionamos anteriormente, utilizaremos dos nodos. Por eso necesitamos proporcionar otro fichero de configuración. En este caso sólo se necesitan las propiedades esenciales: El proxy al Locator, en nombre del nodo, el directorio de la bases de datos y los endpoints del adaptador del nodo:

LISTADO 3J.7: Configuración de un nodo IceGrid convencional (sin Registry)

[icegrid/node2.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
IceGrid.Node.Name=node2
IceGrid.Node.Data=/tmp/db/node2
IceGrid.Node.Endpoints=tcp
```

Recuerda que si está ejecutando los dos nodos en computadores diferentes, debes cambiar la dirección IP por la dirección que tenga el computador en la que se ejecuta el Registry.

Se requiere un fichero de configuración adicional para las aplicaciones que deseen utilizar el Locator:

LISTADO 3J.8: Configuración del Locator

[icegrid/locator.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

3j.2.3. Arrancando IceGrid

El nodo IceGrid normalmente se ejecuta como un *demonio*, es decir, un servicio que arranca automáticamente al conectar cada computador asociado al sistema distribuido y que se ejecuta en *background*. Sin embargo, en este caso y con propósitos didácticos vamos a proceder a arrancarlos manualmente y en primer plano.

Abre una ventana de terminal en el directorio `icegrid`, crea los directorios para las bases de datos del nodo y el registro, y lanza `icegridnode` con la configuración del nodo 1:

```
$ mkdir -p /tmp/db/node1
$ mkdir -p /tmp/db/registry
$ icegridnode --Ice.Config=node1.config
```

Ese terminal queda ocupado con la ejecución del nodo. Déjalo así, más adelante podrás pulsar Control-c para pararlo.

Abre una segunda ventana de terminal en el mismo directorio, crea el directorio para la base de datos y ejecuta otra instancia de `icegridnode`, esta vez con la configuración del nodo 2:

```
$ mkdir -p /tmp/db/node2
$ icegridnode --Ice.Config=node2.config
```

Si todo funciona correctamente, tendrá el mismo efecto y el terminal quedará ocupado.

Ahora procedemos a conectar con el Registry utilizando la aplicación `icegridgui` con la configuración del Locator, que puedes lanzar en un tercer terminal:

```
$ icegridgui
```

Deberías ver una ventana similar a la de la figura 3j.1.

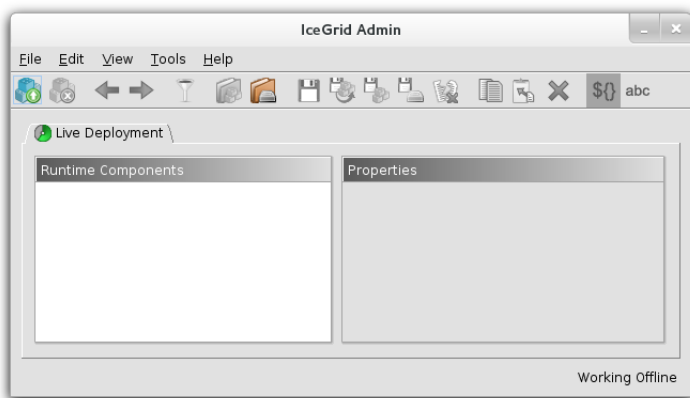



FIGURA 3j.1: IceGrid Admin: Ventana inicial

3j.3. Creando una aplicación distribuida

A continuación crearemos una nueva aplicación con `icegridgui`. Pero primero necesitamos conectar con el Registry pulsando el botón *Log into an IceGrid Registry* situado arriba a la izquierda, el primero de la barra de herramientas .

Aparecerá una ventana con una lista de conexiones como la de la figura 3j.2. Pulsa **New connection**, lo que inicia un pequeño asistente para crear una nueva conexión. Ese asistente tiene los siguientes pasos:

1. Tipo de conexión (figura 3j.3). Elige *Direct connection*.
2. Nombre de la instancia (figura 3j.4). Escribe «IceGrid».
3. Información de direccionamiento (figura 3j.5). Elige *An endpoint string*.
4. Endpoint del Registro IceGrid (figura 3j.6). Escribe «tcp -p 4061».
5. Credenciales (figura 3j.7). Escribe «user» y «pass». Por supuesto, en una aplicación en producción, se requeriría una contraseña o un mecanismo de autenticación equivalente. Recuerda que el registro está configurado con un verificador de permisos *dummy* (falso).

Una vez terminado el asistente, la lista contendrá la conexión recién configurada (figura 3j.8). Pulse **Connect**.

Cuando la aplicación ha establecido el acceso al registro debería aparecer una ventana similar a la de la figura 3j.9. Esta ventana muestra una vista lógica actualizada del sistema distribuido. Puedes ver todos los nodos involucrados en el sistema, es decir, los que están configurados con el Locator asociado al registro al que has conectado. Puedes pulsar los iconos que representan esos nodos y obtener datos actualizados con información sobre su sistema operativo, su carga, etc.

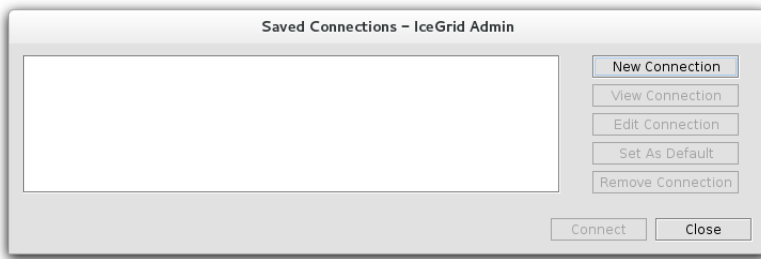


FIGURA 3J.2: IceGrid Admin: Conexiones guardadas

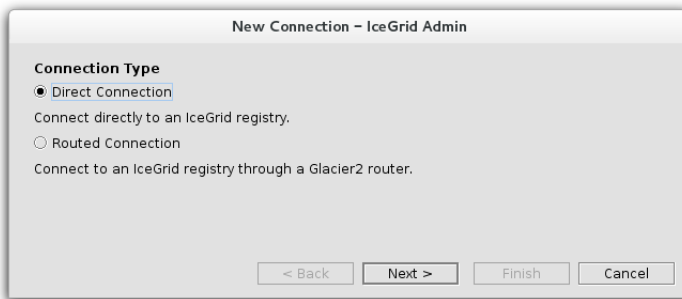


FIGURA 3J.3: IceGrid Admin: Nueva conexión

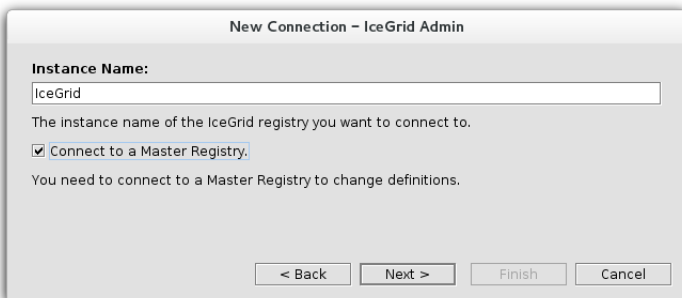
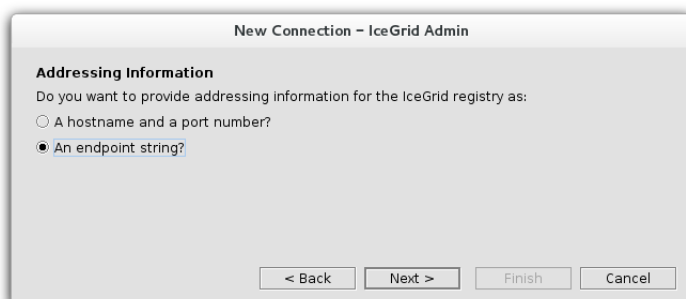


FIGURA 3J.4: IceGrid Admin: Nombre de la instancia de IceGrid



New Connection - IceGrid Admin

Addressing Information

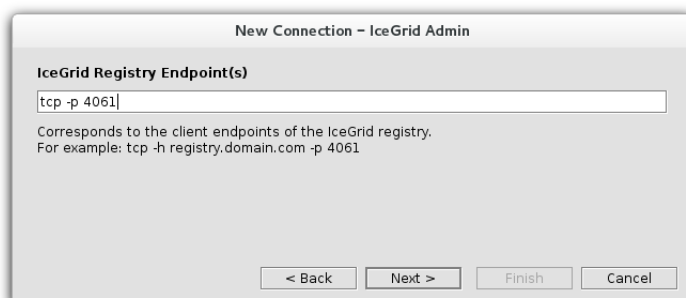
Do you want to provide addressing information for the IceGrid registry as:

☐ A hostname and a port number?

☒ An endpoint string?

< Back Next > Finish Cancel

FIGURA 3J.5: IceGrid Admin: Elegir tipo de dirección



New Connection - IceGrid Admin

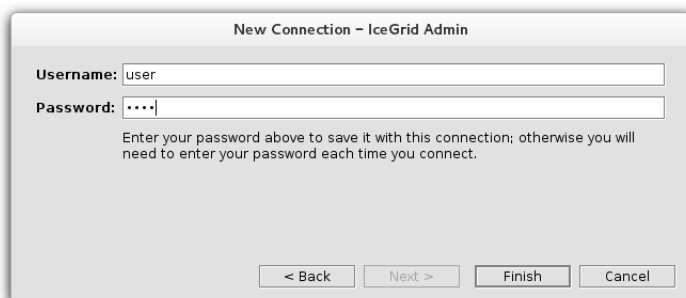
IceGrid Registry Endpoint(s)

tcp -p 4061

Corresponds to the client endpoints of the IceGrid registry.
For example: tcp -h registry.domain.com -p 4061

< Back Next > Finish Cancel

FIGURA 3J.6: IceGrid Admin: Endpoint del Registry



New Connection - IceGrid Admin

Username: user

Password:

Enter your password above to save it with this connection; otherwise you will need to enter your password each time you connect.

< Back Next > Finish Cancel

FIGURA 3J.7: IceGrid Admin: Credenciales

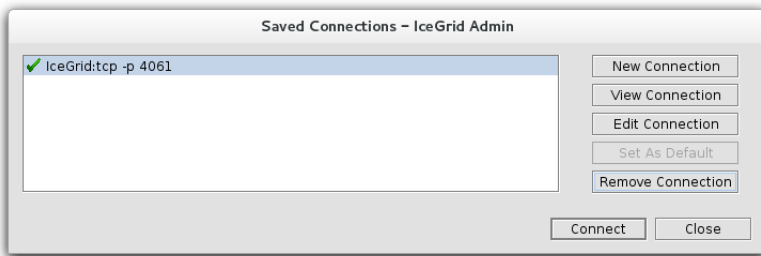
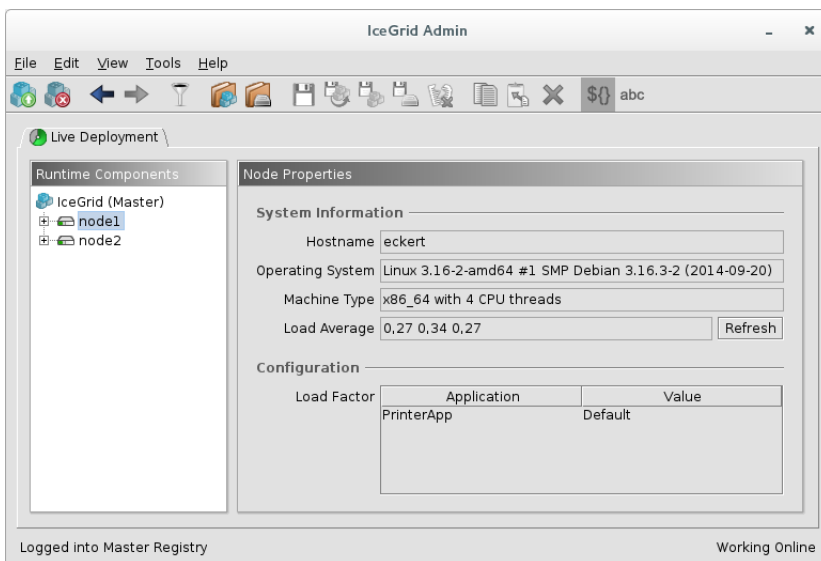


FIGURA 3J.8: IceGrid Admin: Conexión creada

FIGURA 3J.9: IceGrid Admin: Ventana *Live Deployment*

Para crear una nueva aplicación selecciona la opción de menú *File, New, Application with Default Templates from Registry* (figura 3j.10). Aparecerá una nueva pestaña llamada *NewApplication*. Edita el nombre de la aplicación, escribe «PrinterApp» y pulsa **Apply**.

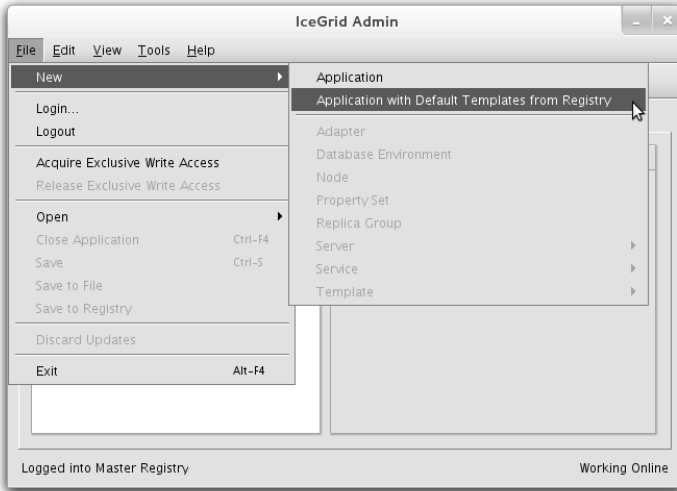


FIGURA 3J.10: IceGrid Admin: Creación de aplicaciones

Abre el menú contextual (botón derecho del ratón) de la carpeta *Nodes* situado en el panel izquierdo (figura 3j.11). Crea un nuevo nodo, cambia su nombre a *node1* y pulsa **Apply**. Crea otro nodo, nómbralo como *node2* y pulsa **Apply** de nuevo. Ahora debería haber dos nodos con los mismos nombres que los que aparecen en la pestaña *Live Deployment*. Comprueba que efectivamente los nombre corresponden (incluyendo mayúsculas y minúsculas).

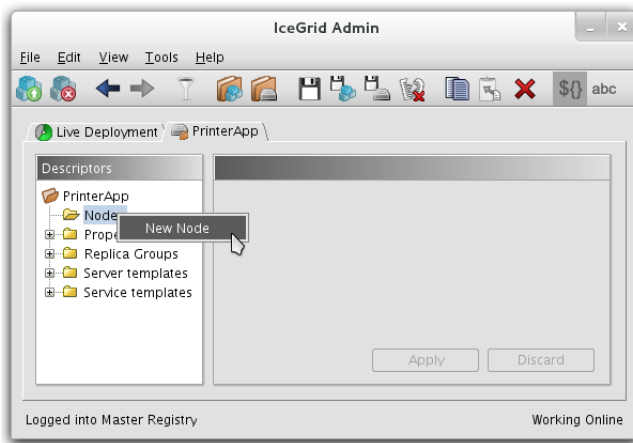


FIGURA 3J.11: IceGrid Admin: Creación de un nodo

3j.4. Despliegue de aplicaciones con IcePatch2

En el menú contextual del nodo `node1` pulsa *New Server from Template* (ver figura 3j.12). Selecciona la plantilla *IcePatch2* en la lista desplegable de la parte superior del diálogo (ver figura 3j.13). En el parámetro *directory* introduce la ruta absoluta del directorio que contiene los binarios de la aplicación «hola mundo» y pulsa **Apply**.

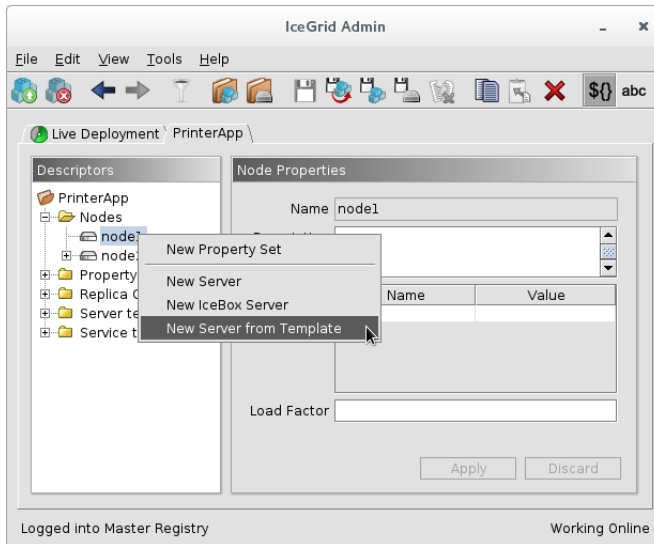


FIGURA 3J.12: IceGrid Admin: Añadiendo el servidor IcePatch2

Ahora tienes una instancia del servicio IcePatch2 configurada en el nodo `node1`. IceGrid utiliza este servicio para desplegar el software necesario a cada nodo del sistema distribuido. Puede haber varias instancias del servicio en una misma aplicación, pero es posible configurar una de ellas como la instancia por defecto. Eso es lo que vamos a hacer con la instancia recién creada. Pulsa *PrinterApp* arriba a la izquierda y en el desplegable *IcePatch2 Proxy* elije la única que aparece (ver figura 3j.14).

3j.5. Instanciación del servidor

Es hora de crear los descriptores para nuestros programas: el cliente y el servidor de «hola mundo».

- Crear un nuevo servidor y cambia el nombre por defecto a `PrinterServer1`.
- Edita la propiedad *Path to Executable* y escribe `java`, en *Command Arguments* escribe «-classpath `./usr/share/java/Ice.jar` `Server`» y,
- En la propiedad *Working Directory* escribe `${application.distrib}` igual que para el cliente.

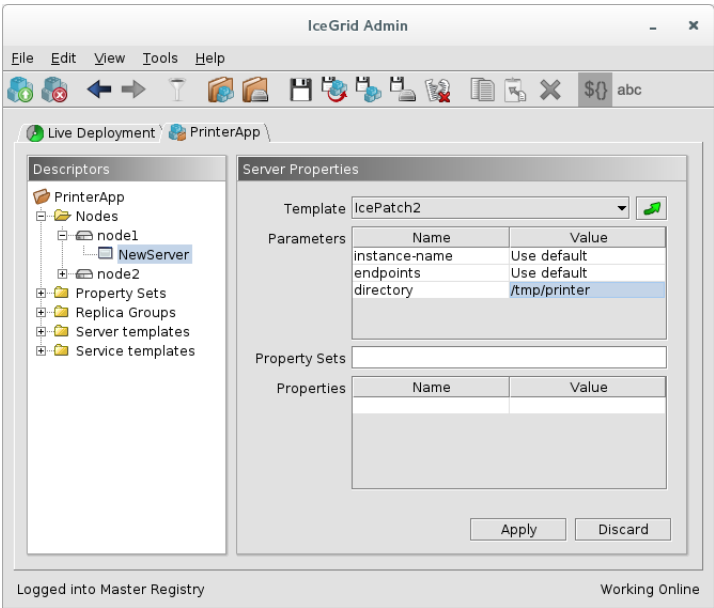


FIGURA 3J.13: IceGrid Admin: Configuración del servicio IcePatch2

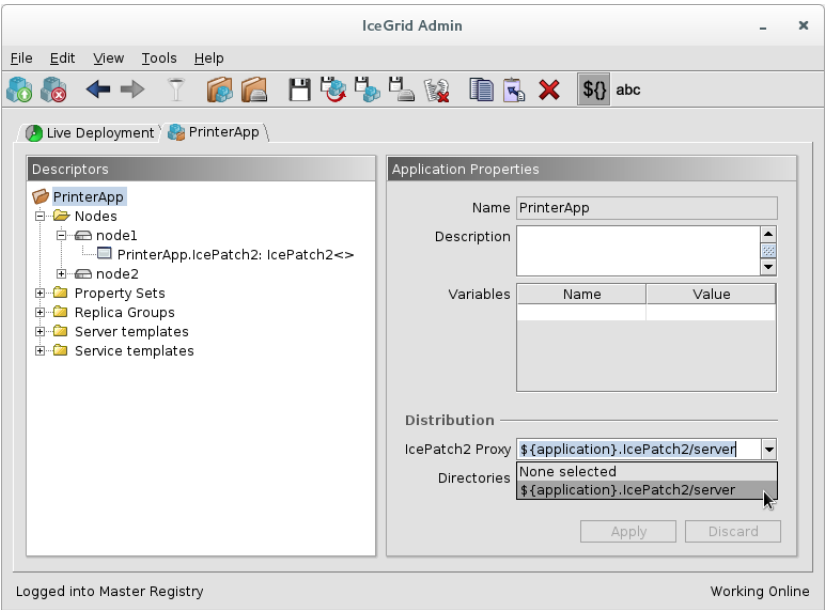


FIGURA 3J.14: IceGrid Admin: Configuración del proxy ICEPATCH2 de la aplicación

Tal como vimos en la sección 2j.5, el adaptador de objetos del servidor requiere configuración adicional, en concreto, debemos proporcionarle los *endpoints* en los que escuchar. Sin embargo, en este caso la configuración de ese adaptador la fijaremos a través de IceGrid.

En el menú contextual de `PrinterServer1` elige *New Adapter*. Cambia el nombre por defecto a `PrinterAdapter` y pulsa **Apply**. Este nombre debe corresponder con el nombre del adaptador definido en el código del servidor mediante el método `createObjectAdapter()`.

Sin embargo, tenemos un problema: ¿Cómo sabremos si el servidor funciona correctamente? Su única función es imprimir en su salida estándar las cadenas que se le envían. Para poder obtener remotamente la salida estándar del servidor configuramos una propiedad adicional.

- Pulsa sobre `PrinterServer1` en el lado izquierdo.
- Pulsa en la tabla `Propiedades` y añade una propiedad llamada `Ice.StdOut`.
- Como valor escribe `${application.distrib}/server-out.txt` y pulsa **Apply**.

Debería quedar como la figura 3j.15. La variable `${application.distrib}` se refiere al directorio en el que se desplegará la aplicación en el nodo. No es necesario conocer cuál será la ruta exacta. Eso es una decisión interna de `IcePatch2`.

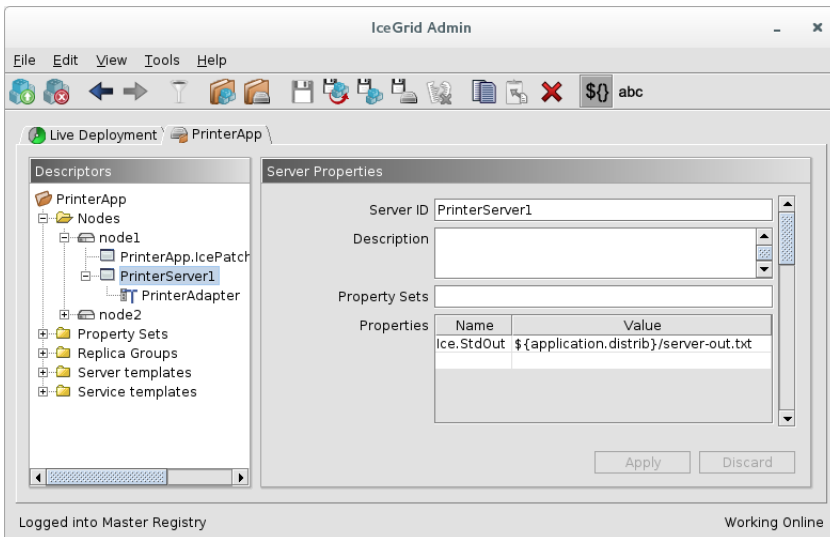


FIGURA 3j.15: IceGrid Admin: Instancia de la aplicación servidor

3j.5.1. Instanciando un segundo servidor


Crearemos ahora una segunda instancia del servidor de impresión en otro nodo. Sigue los mismos pasos que en la sección anterior con dos diferencias:

- Crea un nodo llamado `node2`.

- Crea un servidor llamado `PrinterServer2`
- En la propiedad *Path to Executable* escribe `java`, en *Command Arguments* escribe «`-classpath ./usr/share/java/Ice.jar Server-UUID`»

La única diferencia es que este servidor contendrá un objeto cuya identidad se genera aleatoriamente (ver método `addWithUUID()` del adaptador de objetos).

3j.6. Ejecutando la aplicación

Para empezar, guarda la aplicación recién definida en el registro pulsando el botón *Save to registry* 

3j.6.1. Despliegue de la aplicación

Antes de ejecutar los binarios que componen la aplicación es necesario copiarlos (desplegarlos) a sus respectivos nodos. Esa tarea corre a cargo del servicio `IcePatch2` que acabas de configurar. Sin embargo, antes de poder efectuar el despliegue debes preparar el directorio con los ficheros correspondientes. Esto se consigue ejecutando el comando `icepatch2calc` en el mismo directorio que indicamos en la configuración del `IcePatch2`.

Suponiendo que la aplicación «hola mundo» ya compilada está en el directorio `/tmp/printer` ejecuta:

```
$ icepatch2calc /tmp/printer
```



Recuerda que en la figura 3j.13 se indicó que los binarios de la aplicación están en `/tmp/printer`, pero si están en otra parte (por ejemplo el subdirectorio `ice-hello/java`, debes modificar la propiedad `directory` de `IcePatch2` de acuerdo con la situación real.

Ahora selecciona la solapa *Live Deployment* y elige la opción *Tools, Application, Patch Distribution* tal como muestra la figura 3j.16.

Si todo ha ido bien, la barra de estado mostrará el mensaje «Patching application ‘PrinterApp’... done».

3j.6.2. Ejecutando los servidores

Haz doble click en `node1` en la pestaña *Live Deployment*. Aparecerá su «contenido», concretamente el servidor `PrinterServer1`. Abre el menú contextual de `PrinterServer1` y pulsa *Start*. Un símbolo de «play» aparecerá sobre él y la barra de estado mostrará el mensaje «Starting server ‘PrinterServer1’... done».

Ahora que el servidor está en marcha puedes ver su salida estándar. Sólo tienes que abrir de nuevo el menú contextual de `PrinterServer1` y pulsar *Retrieve stdout*. Aparecerá una ventana similar a la de la figura 3j.17.

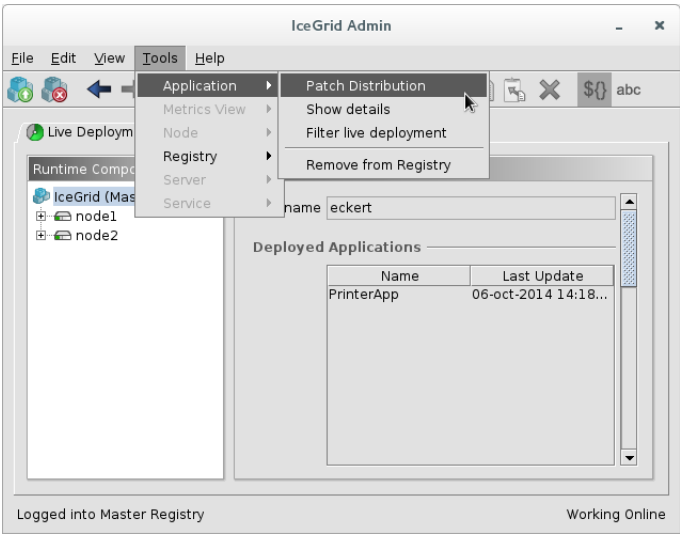


FIGURA 3J.16: IceGrid Admin: Desplegando la aplicación

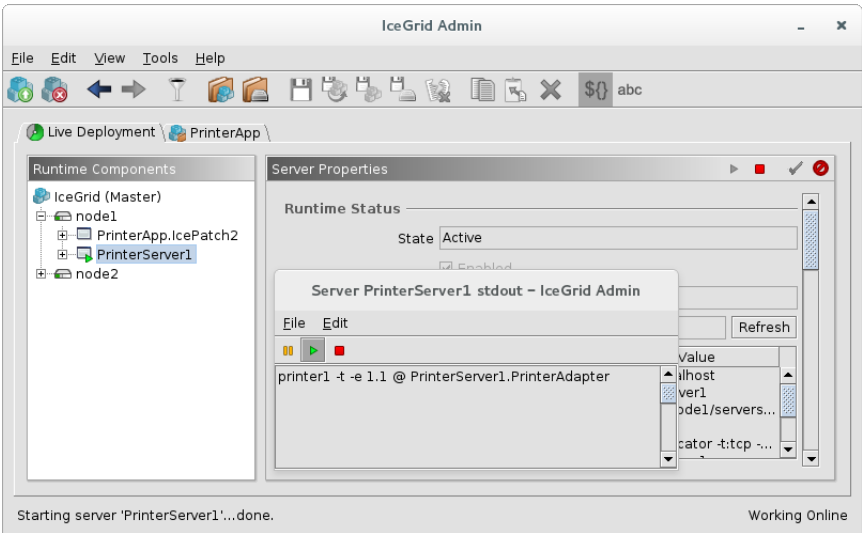


FIGURA 3J.17: IceGrid Admin: Salida estándar del servidor

Como recordarás, el servidor de impresión escribe en consola la representación textual del proxy de su objeto (ver § 2j.5). Sin embargo, la salida que ahora muestra el servidor es bastante distinta:

```
printer1 -t @ PrinterServer1.PrinterAdapter
```

El proxy que se mostraba cuando ejecutamos el servidor desde un terminal era un *proxy directo* (§ 1.3.3.2) porque contenía todos los detalles para contactar con el servidor sin ningún intermediario. Sin embargo, el que vemos ahora es un *proxy indirecto* (§ 1.3.3.3) y carece de información esencial como dirección IP y puerto del servidor.

A partir de un proxy indirecto, el cliente debe pedir al servicio Locator los detalles de conexión que correspondan, convirtiendo así un proxy indirecto en uno directo. El Locator buscará el adaptador `PrinterServer1.PrinterAdapter` y reemplazará el `@PrinterServer1.PrinterAdapter` por los endpoints de dicho adaptador. El adaptador debe ser el único con ese identificador público. Para entender la diferencia con el nombre del adaptador de objetos pulsa otra vez en la aplicación `PrinterApp` y después en el adaptador `PrinterAdapter` dentro de `PrinterServer1` (ver figura 3j.18).

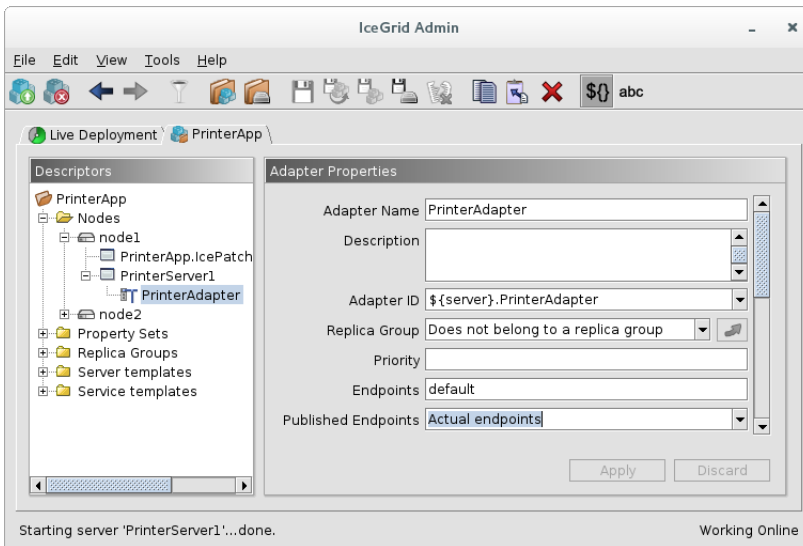


FIGURA 3j.18: IceGrid Admin: Configuración del adaptador de objetos



Nótese que el *Adapter ID* tiene un valor por defecto igual a `${server}.PrinterAdapter`. La variable `${server}` se traducirá por el nombre del servidor (`PrinterServer1` en este caso) y el resto corresponde con el nombre del adaptador de objetos. Puede parecer raro que tenga que existir un nombre interno y un identificador público. Pero esto permite crear instancias diferentes del mismo servidor.

Para ejecutar el segundo servidor, procede del mismo modo desplegando el `node2` y pulsa *Start* en el `PrinterServer2`.

3j.6.3. Ejecutando el cliente

Recuerda que el cliente acepta un argumento desde la línea de comandos: la representación textual del proxy que debe utilizar. Ahora conocemos el valor de ese parámetro, así que ya podemos ejecutar el cliente:

```
$ CLASSPATH=./usr/share/java/Ice.jar
$ java -classpath $CLASSPATH Client "printer1 -t -e 1.1 @ PrinterServer1.PrinterAdapter"
```

Nótese el parámetro `--Ice.Config` en el que se le indica al cliente el fichero en el que aparece el valor de la propiedad `Ice.Default.Locator`. Sin esa referencia, el cliente no sabría cómo resolver proxies indirectos.

Mira la salida estándar del `PrinterServer1` (figura 3j.19). ¡Funciona!

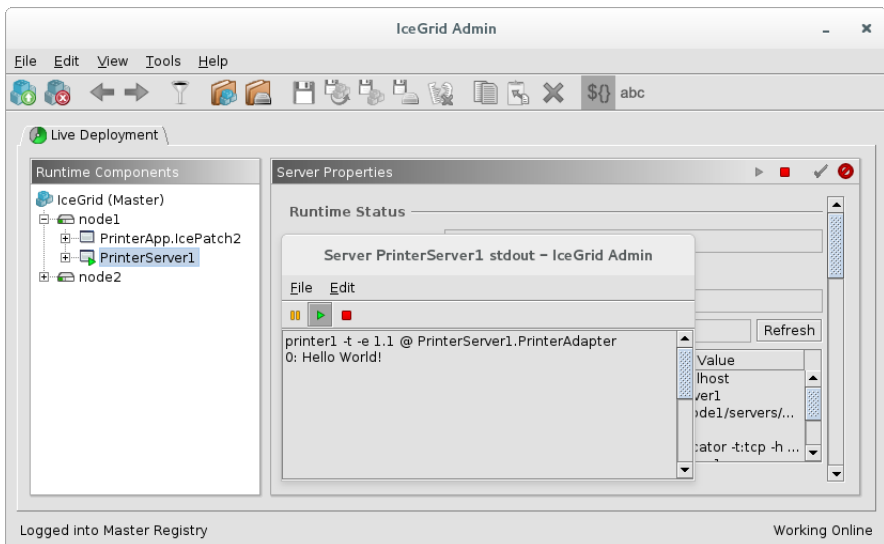


FIGURA 3j.19: IceGrid Admin: Salida estándar del servidor después de ejecutar el cliente

Sólo tenemos que usar `printer1@PrinterServer1.PrinterAdapter`. A diferencia del proxy directo, esta designación siempre será la misma, aunque el servidor se ejecute en **un nodo diferente** o el middleware elija un puerto distinto. Además,

la indirección no resulta especialmente ineficiente puesto que el cliente solo preguntará al Locator la primera vez y cacheará la respuesta. Sólo volverá a consultar al Locator si falla una invocación sobre el proxy cacheado, y todo esto de forma automática y sin la intervención del programador.



El tipo de indirección que conseguimos gracias a los proxies indirectos es lo que entendemos por **transparencia de localización**, uno de los logros más importantes en el campo de la computación distribuida heterogénea.

3j.7. Objetos bien conocidos

Algunos objetos importantes para la aplicación pueden ser incluso más fáciles de localizar. Los objetos bien conocidos pueden ser localizados simplemente por sus nombres. Es bastante sencillo conseguirlo. Ve a la pestaña `PrinterApp` y luego al adaptador `PrinterAdapter` de `PrinterServer1`. Desplaza la ventana hacia abajo hasta la tabla *Well-known objects*. Añade la identidad `printer1` y pulsa `Apply`, tal como se muestra en la figura 3j.20.

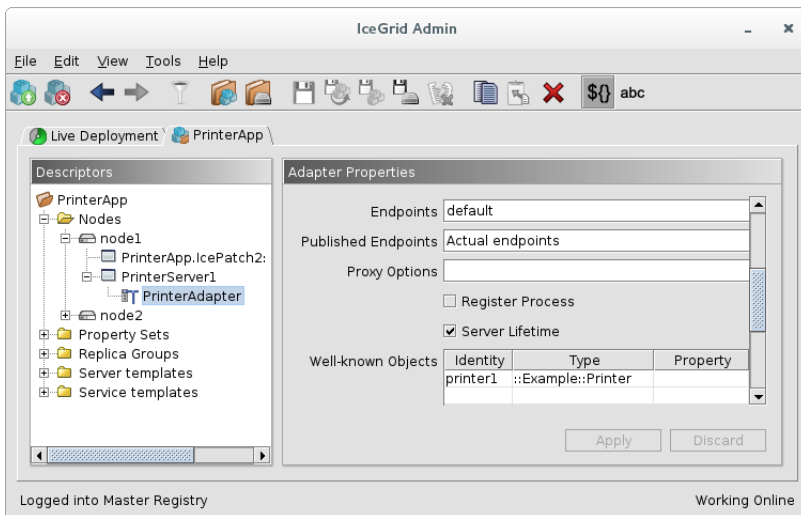


FIGURA 3J.20: IceGrid Admin: Añadiendo un objeto bien conocido

Para invocar el objeto bien conocido con el cliente simplemente debes indicar el nombre como proxy textual:

```
$ java -classpath ../usr/share/java/Ice.jar Client --Ice.Config=../icegrid/locator.config printer1
```

Mira la salida estándar de `PrinterServer1` y comprueba que también funciona. Una cuestión interesante es que ni siquiera ha sido necesario reiniciar el servidor.

En este caso, la cadena `printer1` es otra forma de proxy indirecto. La notación `identidad@adaptador` corresponde con un objeto registrado en un adaptador de objetos bien conocido (aquellos que tiene un *Adapter ID*), mientras que la notación `identidad` corresponde con objetos bien conocidos.

3j.8. Activación y desactivación implícita

A veces podemos ahorrar recursos parando los servidores ociosos hasta que se necesiten realmente. Los proxies indirectos son ideales para conseguirlo de forma automática. Si un cliente intenta contactar con un proxy indirecto por primera vez, hará una petición al Locator. Si falla, intentará contactar al Locator de nuevo.

El registro es capaz de usar las peticiones al Locator como solicitudes de activación implícita. Pulsa en la pestaña *PrinterApp* y luego en *PrinterServer1*. Despliega el menú *Activation mode* y selecciona *on-demand*. Ahora pulsa en **Apply** y en *Save to registry*.

Ahora ve a la pestaña *Live Deployment*, en el menú contextual de *PrinterServer1* pulsas *Stop*. Ahora vuelve a ejecutar el cliente invocando a `printer1`. El servidor *PrinterServer1* arrancará automáticamente en el momento en el que el cliente intenta conectar con él.

Además, es bastante conveniente desactivar los servicios ociosos pasado un tiempo de inactividad. Para aplicar esa posibilidad ve a la pestaña *PrinterApp*, pulsa en *PrinterServer1* y en la tabla de propiedades añade una nueva propiedad llamada `Ice.ServerIdleTime` con el valor '5'. Pulsar **Apply** y luego el botón *Save to registry*. Después de 5 segundos de inactividad, el servidor cerrará automáticamente.

3j.9. Depuración

Es fácil cometer algún error durante la configuración de la aplicación distribuido y puede resultar complicado si no se es sistemático encontrando el problema.

Si algo va mal primero debería revisar la salida de error estándar del programa que falla. Para que esta salida esté disponible desde la herramienta de administración se debe definir la propiedad `Ice.StdErr` y darle una ruta a un fichero tal que `${application.distrib}/server-err.txt`. Una vez que el programa se ejecute podrá acceder a su contenido por medio del menú contextual del nodo servidor en la opción *Retrieve stderr*. Del mismo modo se puede activar la salida estándar con la propiedad `Ice.Stdout`. Estas mismas propiedades pueden definirse en la configuración de los nodos por medio de los ficheros que aparecen en la sección 3j.2.2. Esto nos dará información extra muy útil para localizar y resolver posibles problemas.

3j.9.1. Evitando problemas con `IcePatch2`

Recuerda que si cambias y recompilas un programa debes volverlos a desplegar (ver §3j.6.1). Sin embargo, es bastante frecuente olvidar reiniciar el propio servicio `IcePatch2`. Si el servicio no se reinicia no detectará los cambios.

Una buena solución para este problema es configurar siempre la propiedad `Ice.ServerIdleTime` en la instancia del servidor `IcePatch2` asignándole un valor pequeño.

3j.9.2. Descripción de la aplicación

Recuerda que la aplicación queda definida en un fichero XML. Toda la configuración que se realiza en la definición de la aplicación en `icegridgui` queda almacenada en un fichero bastante legible/editable manualmente. Para nuestra aplicación ese fichero (obviando las plantillas de los servicios comunes) se muestra en el listado 3j.9.

LISTADO 3j.9: Descripción de la aplicación `PrinterApp` en XML
`icegrid/printerapp-cpp.xml`

```
<node name="node1">
  <server id="PrinterServer" activation="manual" exe="java" pwd="{application.distrib}">
    <option>-classpath</option>
    <option>usr/share/java/Ice.jar:.</option>
    <option>Server</option>
    <properties>
      <property name="Ice.StdOut" value="{application.distrib}/server-out.txt"/>
      <property name="Ice.StdErr" value="{application.distrib}/server-err.txt"/>
    </properties>
    <adapter name="PrinterAdapter" endpoints="default" id="{server}.PrinterAdapter">
      <object identity="printer1"/>
    </adapter>
  </server>
  <server-instance template="IcePatch2" directory="/tmp/ice-hello/java/dist">
    <properties>
      <property name="Ice.ServerIdleTime" value="5"/>
    </properties>
  </server-instance>
</node>
</application>
</icegrid>
```

3j.10. Receta

A continuación se listan a modo de resumen los pasos más importantes para realizar un despliegue cliente/servidor sencillo como el que se ha descrito:

- Crear un fichero de configuración para el Registry y para cada uno de los nodos del grid (§3j.2.2).
- Arrancar un `icegridnode` por cada nodo con su configuración correspondiente (§3j.2.3).
- Crear la aplicación distribuida (§3j.3).
- Crear un nodo en la aplicación para cada nodo ejecutado.
- Compilar los programas, copiar los binarios a otro directorio y ejecutar `ice-patch2calc`.
- Crear un servidor de `IcePatch2` (§3j.4) y configurarlo para desplegar el directorio del paso anterior.

- Configurar ese servidor de IcePatch2 como el IcePatch2 por defecto de la aplicación distribuida (figura 3j.14).
- Crear los servidores IceGrid para ambos programas (3j.5) en sus respectivos nodos.
- Desplegar la aplicación (§ 3j.6.1).
- Arrancar los servidores (§ 3j.6.2).
- Ejecutar el cliente (§ 3j.6.3).

3j.11. Ejercicios

- E 3j.01** Invoca el servidor desplegado con IceGrid ejecutando el cliente desde otro servidor IceGrid.
- E 3j.02** Modifica el código del cliente para que invoque a los dos servidores.
- E 3j.03** El servidor del `node2` tiene una identidad aleatoria. Averigua cómo fijar la identidad del objeto por medio de una propiedad. Cambia el programa que ejecutan ambos servidores para ejecutar la nueva versión (ambos nodos el mismo programa) e indica por medio de una propiedad que sus identidades son respectivamente «printer1» y «printer2».
- E 3j.04** Define nodos adicionales, crea una plantilla¹ para el servidor `PrinterServer`. Arranca nodos adicionales y, usando la plantilla, instancia varias impresoras (objetos *Printer*) en ellos. Invoca esas impresoras con el cliente original desde la consola. Consulta el ejemplo `demopy/IceGrid/simple` de la distribución de ICE².
- E 3j.05** Explora las posibilidades de `icegridadmin`³, la herramienta de administración en línea de comandos, para obtener información y modificar la configuración de la aplicación. ¿Qué comando debes introducir para obtener la lista de nodos? ¿Y la lista de servidores?
- E 3j.06** Repite el tutorial de este capítulo usando un computador diferente para cada nodo. Esos computadores pueden ser virtuales (se aconseja `VirtualBox`).

¹<http://doc.zeroc.com/display/Ice/IceGrid+Templates>

²<http://www.zeroc.com/download/Ice/3.5/Ice-3.5.0-demos.tar.gz>

³<http://doc.zeroc.com/display/Ice/icegridadmin+Command+Line+Tool>

Gestión de aplicaciones distribuidas

[Python]

En este capítulo veremos el uso básico de `IceGrid`, un servicio incluido en `ZeroC ICE` que facilita la gestión, depuración y despliegue de aplicaciones distribuidas en grid. Además se introduce `IcePatch2`, el servicio de despliegue de software. Juntos permiten aplicar procedimientos avanzados sin necesidad de escribir ni una sola línea de código. Una de las características más importantes que proporciona es la *transparencia de localización*, un concepto clave en cualquier sistema distribuido.

3p.1. Introducción

En el capítulo 2p vimos cómo desarrollar y ejecutar una aplicación distribuida básica. El servidor imprime por pantalla la versión textual del proxy para el objeto que aloja y el cliente la utiliza para obtener a su vez el objeto proxy con el que invocar al objeto remoto. Sin embargo, desde el punto de vista de la gestión, resulta muy incómodo el hecho de que los clientes deban conocer los detalles de direccionamiento de los objetos.

En una aplicación de medio o gran tamaño se utilizan cientos o miles de servidores y objetos. Es obvio que se requiere un método más potente que pueda escalar sin problema. En este capítulo utilizaremos exactamente el mismo programa del capítulo 2p y veremos cómo desplegar, arrancar y gestionar la aplicación (formada por un cliente y dos servidores) de un modo mucho más sistemático. Todo ello es posible gracias a los proxies indirectos (vea § 1.3.3.3).

3p.2. IceGrid

ICE incluye un conjunto de servicios cuidadosamente diseñados que cooperan para proporcionar una gran variedad de prestaciones avanzadas. Uno de los servicios más importantes es `IceGrid`.

IceGrid proporciona una gran variedad de funcionalidades para gestión remota de la aplicación distribuida, activación automática (implícita) de objetos, balanceo de carga y transparencia de localización.

3p.2.1. Componentes de IceGrid

IceGrid depende de una base de datos compartida llamada *IceGrid Registry*. Contiene información sobre los objetos remotos conocidos por el sistema distribuido, las aplicaciones actualmente desplegadas, los nodos de cómputo disponibles y algunos otros datos. El *Registry* es un componente clave en el sistema y sólo puede existir una instancia en cada aplicación distribuida. Se puede decir que el *Registry* es el que determina qué objetos forman parte de la aplicación. Puede haber varios *Registry* en ejecución en el mismo grid, pero corresponderían a aplicaciones distribuidas diferentes.

Además, IceGrid requiere que cada nodo de cómputo asociado al sistema ejecute un servicio llamado *IceGrid Node*. Ésta es la forma en la que IceGrid puede determinar qué computadores están disponibles para la ejecución de componentes de la aplicación.

Por último, IceGrid incluye un par de herramientas de administración que se pueden utilizar como interfaz con el usuario para controlar todas sus características. Existe una aplicación en línea de comando llamada *icegridadmin* y otra con interfaz gráfica llamada *icegridgui*. Usaremos la segunda en este capítulo.

IceGrid maneja algunos conceptos que es importante aclarar, debido a que no corresponden exactamente con el significado habitual:

Nodo

Corresponde con una instancia de *IceGrid Node*. Identifica un «nodo lógico», es decir, no tiene porqué corresponder unívocamente con un computador, pudiendo haber más de un *IceGrid Node* ejecutándose sobre un mismo computador, que a su vez pueden estar vinculados al mismo *Registry* o a distintos.

Servidor

Identifica, mediante un nombre único, a un programa que se ejecutará en un nodo. Incluye los atributos y propiedades que puedan ser necesarios para su configuración. Nótese que el programa a ejecutar puede ser también un cliente o incluso un programa que no tiene nada que ver con ICE.

Adaptador de objetos

Incluye datos específicos de un adaptador de objetos utilizado en un servidor ICE, incluyendo endpoints, objetos bien conocidos, etc.

Aplicación

Se refiere al conjunto de servicios, objetos y sus respectivas configuración que conforman la *aplicación distribuida*. Las descripciones de aplicaciones se pueden almacenar en la base de datos de IceGrid y también se pueden exportar como ficheros XML.

3p.2.2. Configuración de IceGrid

Lo primero es configurar un conjunto de computadores que utilizaremos durante la etapa de desarrollo de la aplicación. Como mínimo necesitamos un *IceGrid Registry* y un *IceGrid node*, aunque pueden ejecutarse en un mismo nodo, de hecho, en un mismo programa. Concretamente vamos a utilizar dos nodos IceGrid, y uno de ellos ejecutará también el Registry.

La configuración de IceGrid es muy similar a la de cualquier otra aplicación de ICE. Debemos proporcionar un conjunto de propiedades en forma de pares «clave=valor» en un fichero de texto plano (vea § 1.7).

Una de las propiedades que siempre debe tener la configuración de cualquier nodo IceGrid es el proxy al servicio *Locator*. Veremos la utilidad de este servicio más adelante. El *Locator* lo proporciona la instancia del Registry y tiene la estructura que se muestra en el siguiente listado:

LISTADO 3P.1: Configuración del nodo 1 (fragmento)

[icegrid/node1.config](#)

```
Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

La dirección IP 127.0.0.1 tendría que ser reemplazada por el nombre o la dirección IP del computador que ejecuta el Registry. En este ejemplo mínimo, todos los componentes estarán en el mismo computador.



Recuerda que la dirección IP 127.0.0.1 es una dirección especial que se asigna a la interfaz *loopback*. Entre otras cosas permite comunicar servidores y clientes que se ejecutan en el mismo computador sin necesidad de una NIC. Todo computador que ejecute la pila de protocolos TCP/IP tiene esta interfaz con esa dirección.

Cada nodo debe proporcionar un nombre identificativo y un directorio en el que almacenar la base de datos del nodo. El programa *icegridnode* es una aplicación ICE convencional, como nuestro servidor de impresión. Por tanto, se deben proporcionar los *endpoints* para el adaptador de objetos del nodo IceGrid.

LISTADO 3P.2: Configuración del nodo 1 (continuación)

[icegrid/node1.config](#)

```
IceGrid.Node.Name=node1
IceGrid.Node.Data=/tmp/db/node1
IceGrid.Node.Endpoints=tcp
```

Eso es todo lo que se necesita para configurar un nodo IceGrid normal. Pero necesitamos un nodo configurado especialmente para ejecutar también un Registry. Para ello utilizamos la siguiente propiedad:

LISTADO 3P.3: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Node.CollocateRegistry=1
```

Cuando esta propiedad tiene un valor distinto de cero, una instancia de IceGrid Registry se añade al adaptador que se especifique (mediante propiedades específicas). El nodo tiene tres adaptadores diferentes dado que se usan para propósitos distintos con privilegios específicos. El adaptador «Client» se utiliza para registrar el servicio Locator, por lo que debería estar configurado para usar el puerto estándar 4061. También debemos indicar un directorio para la base de datos del Registry (línea 4):

IANA

La [IANA](#) estandariza los puertos TCP y UDP para servicios comunes.

LISTADO 3P.4: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
1 IceGrid.Registry.Client.Endpoints=tcp -p 4061
2 IceGrid.Registry.Server.Endpoints=tcp
3 IceGrid.Registry.Internal.Endpoints=tcp
4 IceGrid.Registry.Data=/tmp/db/registry
```

El acceso al Registry está controlado por un verificador de permisos —un servicio simple para autenticación y autorización de usuarios. Existe una implementación hueca (*dummy*) que se suele utilizar durante el desarrollo de la aplicación:

LISTADO 3P.5: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.PermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

Por último, se debe configurar una ruta absoluta al fichero de plantillas `templates.xml` que se distribuye conjuntamente con ICE.

LISTADO 3P.6: Configuración del Registry (continuación)
[icegrid/node1.config](#)

```
IceGrid.Registry.DefaultTemplates=/usr/share/Ice-3.6.1/templates.xml
```



El fichero `templates.xml` puede estar en una ruta diferente a la indicada en función de la versión de ICE instalada en el sistema.

Como mencionamos anteriormente, utilizaremos dos nodos. Por eso necesitamos proporcionar otro fichero de configuración. En este caso sólo se necesitan las propiedades esenciales: El proxy al Locator, en nombre del nodo, el directorio de la bases de datos y los endpoints del adaptador del nodo:

LISTADO 3P.7: Configuración de un nodo IceGrid convencional (sin Registry)
[icegrid/node2.config](#)

```
Ice.Default Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
IceGrid.Node.Name=node2
IceGrid.Node.Data=/tmp/db/node2
IceGrid.Node.Endpoints=tcp
```

Recuerda que si está ejecutando los dos nodos en computadores diferentes, debes cambiar la dirección IP por la dirección que tenga el computador en la que se ejecuta el Registry.

Se requiere un fichero de configuración adicional para las aplicaciones que deseen utilizar el Locator:

LISTADO 3P.8: Configuración del Locator
[icegrid/locator.config](#)

```
Ice.Default Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

3p.2.3. Arrancando IceGrid

El nodo IceGrid normalmente se ejecuta como un *demonio*, es decir, un servicio que arranca automáticamente al conectar cada computador asociado al sistema distribuido y que se ejecuta en *background*. Sin embargo, en este caso y con propósitos didácticos vamos a proceder a arrancarlos manualmente y en primer plano.

Abre una ventana de terminal en el directorio `icegrid`, crea los directorios para las bases de datos del nodo y el registro, y lanza `icegridnode` con la configuración del nodo 1:

```
$ mkdir -p /tmp/db/node1
$ mkdir -p /tmp/db/registry
$ icegridnode --Ice.Config=node1.config
```

Ese terminal queda ocupado con la ejecución del nodo. Déjalo así, más adelante podrás pulsar Control-c para pararlo.

Abre una segunda ventana de terminal en el mismo directorio, crea el directorio para la base de datos y ejecuta otra instancia de `icegridnode`, esta vez con la configuración del nodo 2:

```
$ mkdir -p /tmp/db/node2
$ icegridnode --Ice.Config=node2.config
```

Si todo funciona correctamente, tendrá el mismo efecto y el terminal quedará ocupado.

Ahora procedemos a conectar con el Registry utilizando la aplicación `icegridgui` con la configuración del Locator, que puedes lanzar en un tercer terminal:

```
$ icegridgui
```

Deberías ver una ventana similar a la de la figura 3p.1.

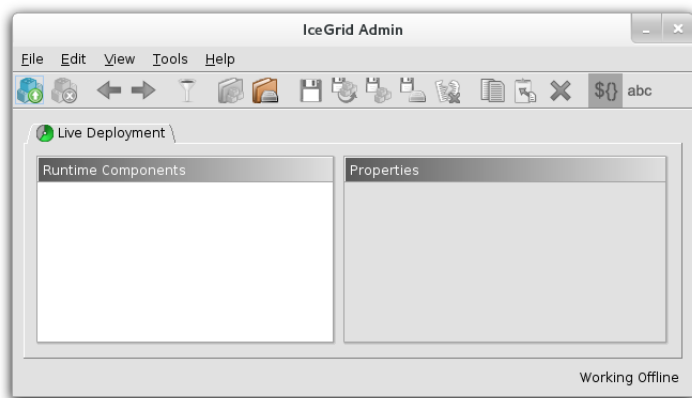



FIGURA 3P.1: IceGrid Admin: Ventana inicial

3p.3. Creando una aplicación distribuida

A continuación crearemos una nueva aplicación con `icegridgui`. Pero primero necesitamos conectar con el Registry pulsando el botón *Log into an IceGrid Registry* situado arriba a la izquierda, el primero de la barra de herramientas .

Aparecerá una ventana con una lista de conexiones como la de la figura 3p.2. Pulsa **New connection**, lo que inicia un pequeño asistente para crear una nueva conexión. Ese asistente tiene los siguientes pasos:

1. Tipo de conexión (figura 3p.3). Elige *Direct connection*.
2. Nombre de la instancia (figura 3p.4). Escribe «IceGrid».
3. Información de direccionamiento (figura 3p.5). Elige *An endpoint string*.
4. Endpoint del Registro IceGrid (figura 3p.6). Escribe «tcp -p 4061».
5. Credenciales (figura 3p.7). Escribe «user» y «pass». Por supuesto, en una aplicación en producción, se requeriría una contraseña o un mecanismo de autenticación equivalente. Recuerda que el registro está configurado con un verificador de permisos *dummy* (falso).

Una vez terminado el asistente, la lista contendrá la conexión recién configurada (figura 3p.8). Pulse **Connect**.

Cuando la aplicación ha establecido el acceso al registro debería aparecer una ventana similar a la de la figura 3p.9. Esta ventana muestra una vista lógica actualizada del sistema distribuido. Puedes ver todos los nodos involucrados en el sistema, es decir, los que están configurados con el Locator asociado al registro al que has conectado. Puedes pulsar los iconos que representan esos nodos y obtener datos actualizados con información sobre su sistema operativo, su carga, etc.

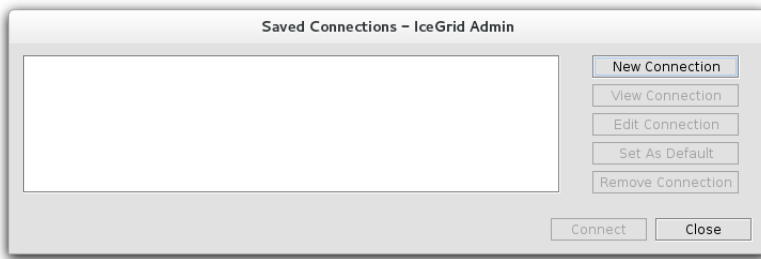


FIGURA 3P.2: IceGrid Admin: Conexiones guardadas

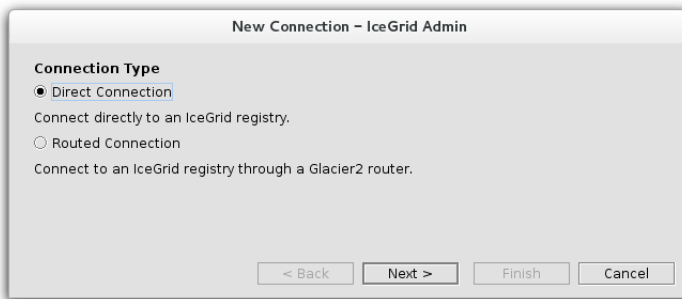


FIGURA 3P.3: IceGrid Admin: Nueva conexión

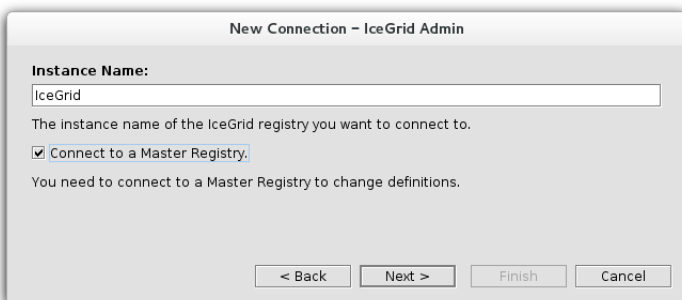


FIGURA 3P.4: IceGrid Admin: Nombre de la instancia de IceGrid

New Connection - IceGrid Admin

Addressing Information

Do you want to provide addressing information for the IceGrid registry as:

☐ A hostname and a port number?

☒ An endpoint string?

< Back Next > Finish Cancel

FIGURA 3P.5: IceGrid Admin: Elegir tipo de dirección

New Connection - IceGrid Admin

IceGrid Registry Endpoint(s)

tcp -p 4061

Corresponds to the client endpoints of the IceGrid registry.
For example: tcp -h registry.domain.com -p 4061

< Back Next > Finish Cancel

FIGURA 3P.6: IceGrid Admin: Endpoint del Registry

New Connection - IceGrid Admin

Username: user

Password:

Enter your password above to save it with this connection; otherwise you will need to enter your password each time you connect.

< Back Next > Finish Cancel

FIGURA 3P.7: IceGrid Admin: Credenciales

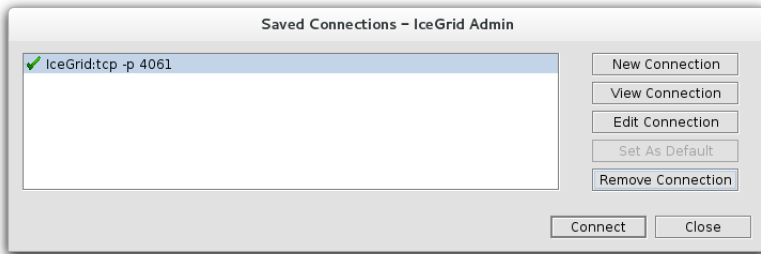
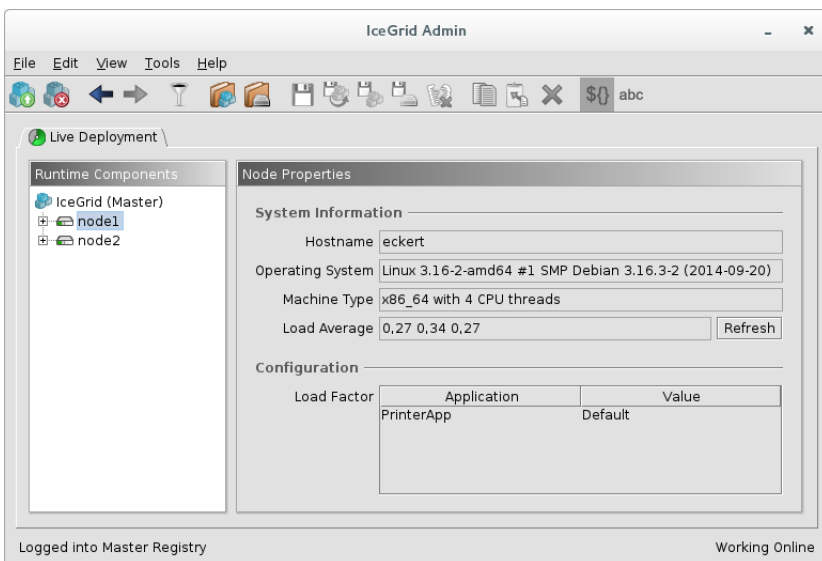


FIGURA 3P.8: IceGrid Admin: Conexión creada

FIGURA 3P.9: IceGrid Admin: Ventana *Live Deployment*

Para crear una nueva aplicación selecciona la opción de menú *File, New, Application with Default Templates from Registry* (figura 3p.10). Aparecerá una nueva pestaña llamada *NewApplication*. Edita el nombre de la aplicación, escribe «PrinterApp» y pulsa **Apply**.

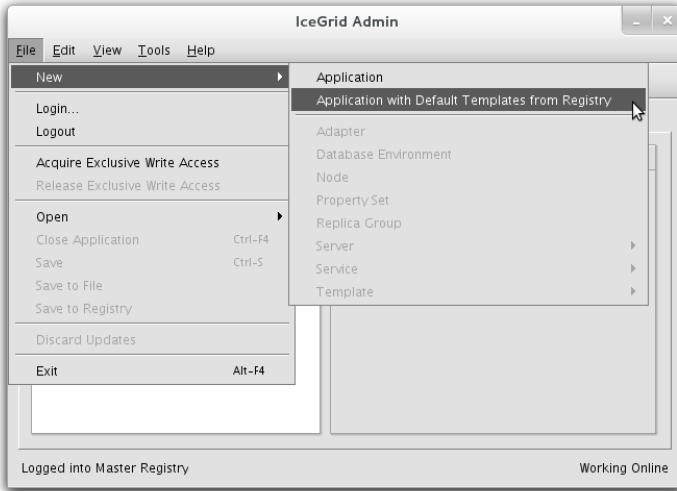


FIGURA 3P.10: IceGrid Admin: Creación de aplicaciones

Abre el menú contextual (botón derecho del ratón) de la carpeta *Nodes* situado en el panel izquierdo (figura 3p.11). Crea un nuevo nodo, cambia su nombre a *node1* y pulsa **Apply**. Crea otro nodo, nómbralo como *node2* y pulsa **Apply** de nuevo. Ahora debería haber dos nodos con los mismos nombres que los que aparecen en la pestaña *Live Deployment*. Comprueba que efectivamente los nombre corresponden (incluyendo mayúsculas y minúsculas).

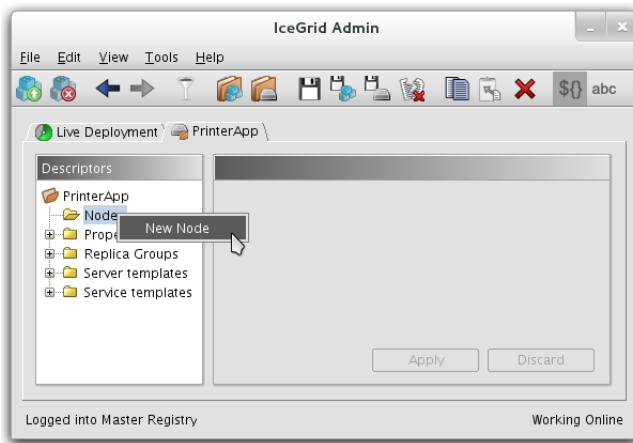


FIGURA 3P.11: IceGrid Admin: Creación de un nodo

3p.4. Despliegue de aplicaciones con IcePatch2

En el menú contextual del nodo `node1` pulsa *New Server from Template* (ver figura 3p.12). Selecciona la plantilla *IcePatch2* en la lista desplegable de la parte superior del diálogo (ver figura 3p.13). En el parámetro *directory* introduce la ruta absoluta del directorio que contiene los binarios de la aplicación «hola mundo» y pulsa **Apply**.

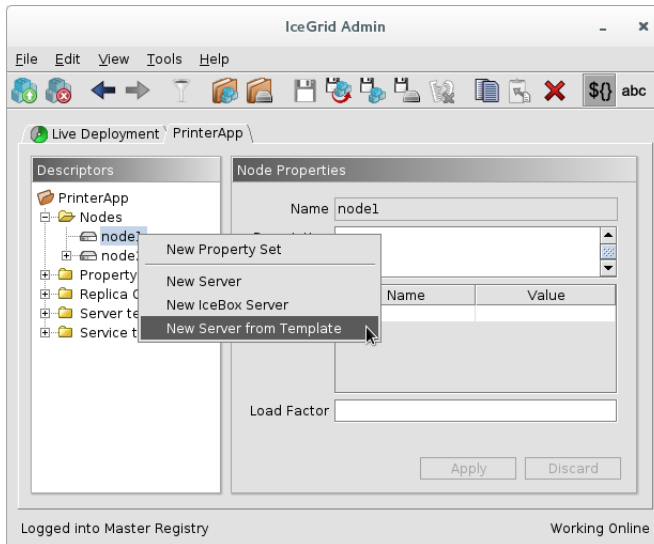


FIGURA 3P.12: IceGrid Admin: Añadiendo el servidor IcePatch2

Ahora tienes una instancia del servicio IcePatch2 configurada en el nodo `node1`. IceGrid utiliza este servicio para desplegar el software necesario a cada nodo del sistema distribuido. Puede haber varias instancias del servicio en una misma aplicación, pero es posible configurar una de ellas como la instancia por defecto. Eso es lo que vamos a hacer con la instancia recién creada. Pulsa *PrinterApp* arriba a la izquierda y en el desplegable *IcePatch2 Proxy* elije la única que aparece (ver figura 3p.14).

3p.5. Instanciación del servidor

Es hora de crear los descriptores para nuestros programas: el cliente y el servidor de «hola mundo».

- Crear un nuevo servidor y cambia el nombre por defecto a `PrinterServer1`.
- Edita la propiedad *Path to Executable* y escribe `./Server.py` y,
- En la propiedad *Working Directory* escribe `${application.distrib}` igual que para el cliente.

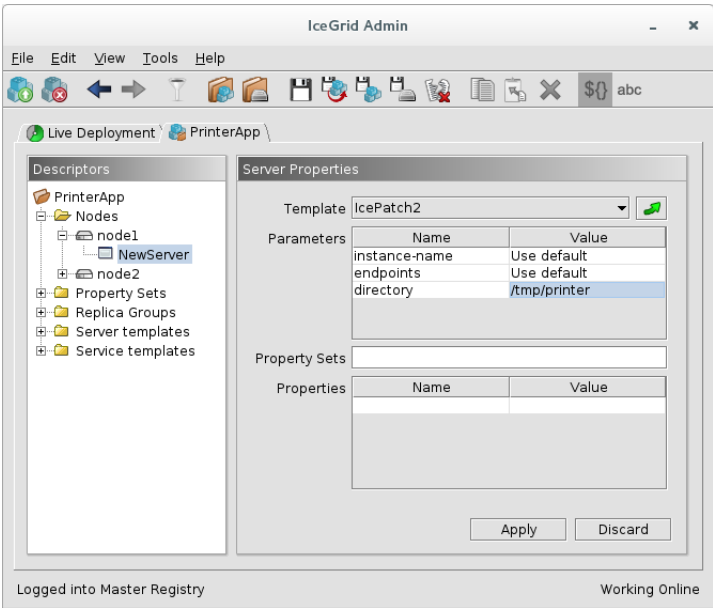


FIGURA 3P.13: IceGrid Admin: Configuración del servicio IcePatch2

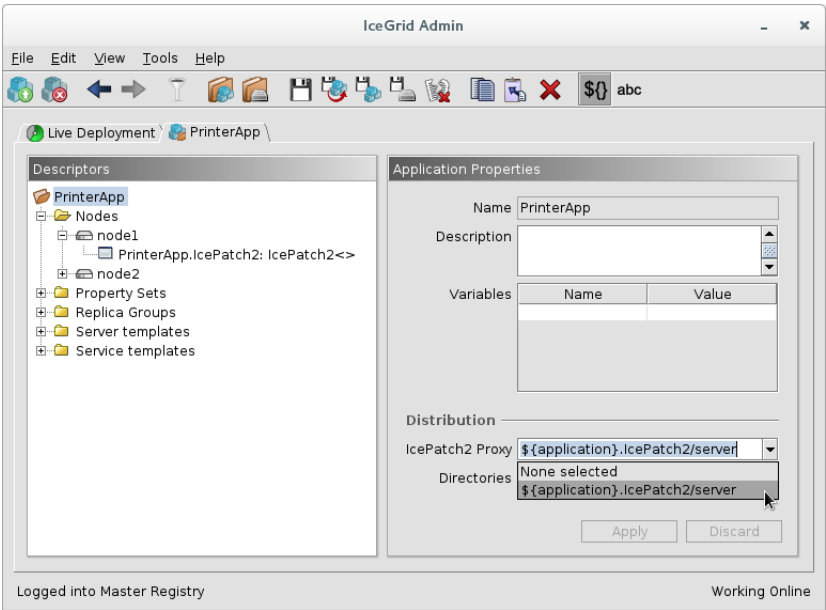


FIGURA 3P.14: IceGrid Admin: Configuración del proxy ICEPATCH2 de la aplicación

Tal como vimos en la sección 2p.5, el adaptador de objetos del servidor requiere configuración adicional, en concreto, debemos proporcionarle los *endpoints* en los que escuchar. Sin embargo, en este caso la configuración de ese adaptador la fijaremos a través de IceGrid.

En el menú contextual de `PrinterServer1` elige *New Adapter*. Cambia el nombre por defecto a `PrinterAdapter` y pulsa **Apply**. Este nombre debe corresponder con el nombre del adaptador definido en el código del servidor mediante el método `createObjectAdapter()`.

Sin embargo, tenemos un problema: ¿Cómo sabremos si el servidor funciona correctamente? Su única función es imprimir en su salida estándar las cadenas que se le envían. Para poder obtener remotamente la salida estándar del servidor configuramos una propiedad adicional.

- Pulsa sobre `PrinterServer1` en el lado izquierdo.
- Pulsa en la tabla `Propierties` y añade una propiedad llamada `Ice.StdOut`.
- Como valor escribe `${application.distrib}/server-out.txt` y pulsa **Apply**.

Debería quedar como la figura 3p.15. La variable `${application.distrib}` se refiere al directorio en el que se desplegará la aplicación en el nodo. No es necesario conocer cuál será la ruta exacta. Eso es una decisión interna de `IcePatch2`.

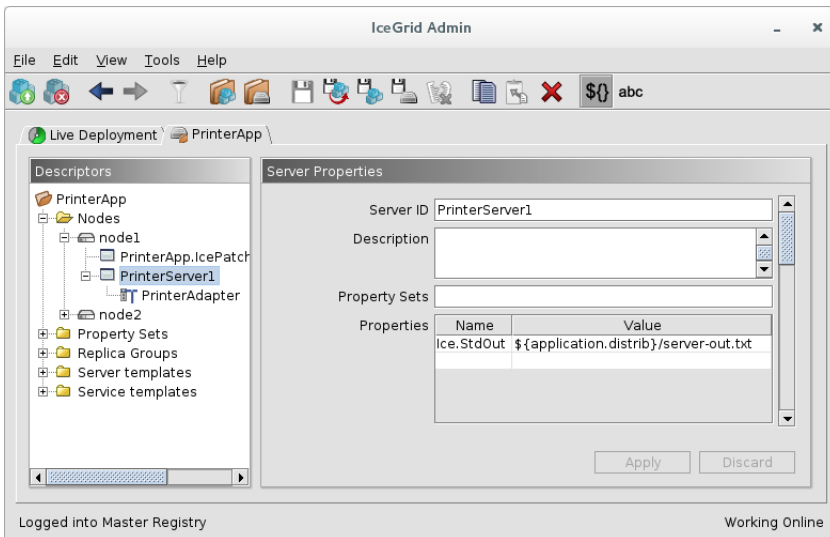


FIGURA 3P.15: IceGrid Admin: Instancia de la aplicación servidor

3p.5.1. Instanciando un segundo servidor


Crearemos ahora una segunda instancia del servidor de impresión en otro nodo. Sigue los mismos pasos que en la sección anterior con dos diferencias:

- Crea un nodo llamado `node2`.

- Crea un servidor llamado `PrinterServer2`
- En la propiedad *Path to Executable* escribe `./Server-UUID.py`

La única diferencia es que este servidor contendrá un objeto cuya identidad se genera aleatoriamente (ver método `addWithUUID()` del adaptador de objetos).

3p.6. Ejecutando la aplicación

Para empezar, guarda la aplicación recién definida en el registro pulsando el botón *Save to registry* 

3p.6.1. Despliegue de la aplicación

Antes de ejecutar los binarios que componen la aplicación es necesario copiarlos (desplegarlos) a sus respectivos nodos. Esa tarea corre a cargo del servicio IcePatch2 que acabas de configurar. Sin embargo, antes de poder efectuar el despliegue debes preparar el directorio con los ficheros correspondientes. Esto se consigue ejecutando el comando `icepatch2calc` en el mismo directorio que indicamos en la configuración del IcePatch2.

Suponiendo que la aplicación «hola mundo» ya compilada está en el directorio `/tmp/printer` ejecuta:

```
$ icepatch2calc /tmp/printer
```



Recuerda que en la figura 3p.13 se indicó que los binarios de la aplicación están en `/tmp/printer`, pero si están en otra parte (por ejemplo el subdirectorio `ice-hello/py`, debes modificar la propiedad `directory` de IcePatch2 de acuerdo con la situación real.

Ahora selecciona la solapa *Live Deployment* y elige la opción *Tools, Application, Patch Distribution* tal como muestra la figura 3p.16.

Si todo ha ido bien, la barra de estado mostrará el mensaje «Patching application 'PrinterApp'... done».

3p.6.2. Ejecutando los servidores

Haz doble click en `node1` en la pestaña *Live Deployment*. Aparecerá su «contenido», concretamente el servidor `PrinterServer1`. Abre el menú contextual de `PrinterServer1` y pulsa *Start*. Un símbolo de «play» aparecerá sobre él y la barra de estado mostrará el mensaje «Starting server 'PrinterServer1'... done».

Ahora que el servidor está en marcha puedes ver su salida estándar. Sólo tienes que abrir de nuevo el menú contextual de `PrinterServer1` y pulsar *Retrieve stdout*. Aparecerá una ventana similar a la de la figura 3p.17.

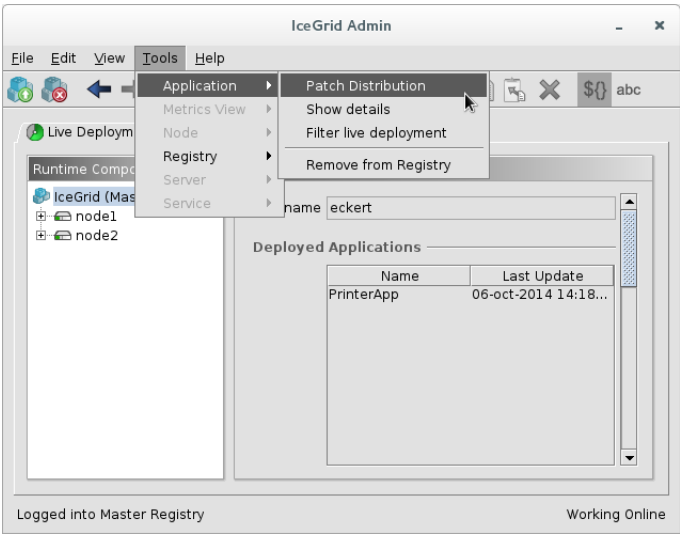


FIGURA 3P.16: IceGrid Admin: Desplegando la aplicación

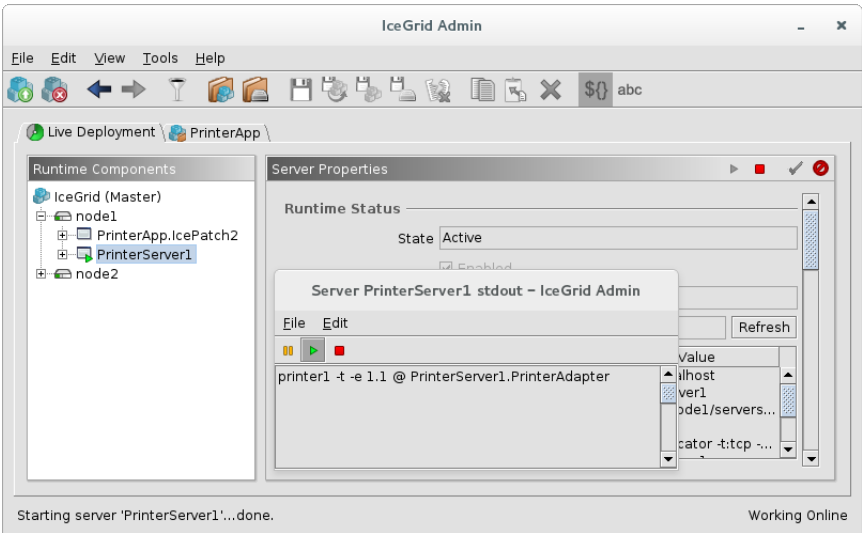


FIGURA 3P.17: IceGrid Admin: Salida estándar del servidor

Como recordarás, el servidor de impresión escribe en consola la representación textual del proxy de su objeto (ver §2p.5). Sin embargo, la salida que ahora muestra el servidor es bastante distinta:

```
printer1 -t @ PrinterServer1.PrinterAdapter
```

El proxy que se mostraba cuando ejecutamos el servidor desde un terminal era un *proxy directo* (§1.3.3.2) porque contenía todos los detalles para contactar con el servidor sin ningún intermediario. Sin embargo, el que vemos ahora es un *proxy indirecto* (§1.3.3.3) y carece de información esencial como dirección IP y puerto del servidor.

A partir de un proxy indirecto, el cliente debe pedir al servicio Locator los detalles de conexión que correspondan, convirtiendo así un proxy indirecto en uno directo. El Locator buscará el adaptador `PrinterServer1.PrinterAdapter` y reemplazará el `@PrinterServer1.PrinterAdapter` por los endpoints de dicho adaptador. El adaptador debe ser el único con ese identificador público. Para entender la diferencia con el nombre del adaptador de objetos pulsa otra vez en la aplicación `PrinterApp` y después en el adaptador `PrinterAdapter` dentro de `PrinterServer1` (ver figura 3p.18).

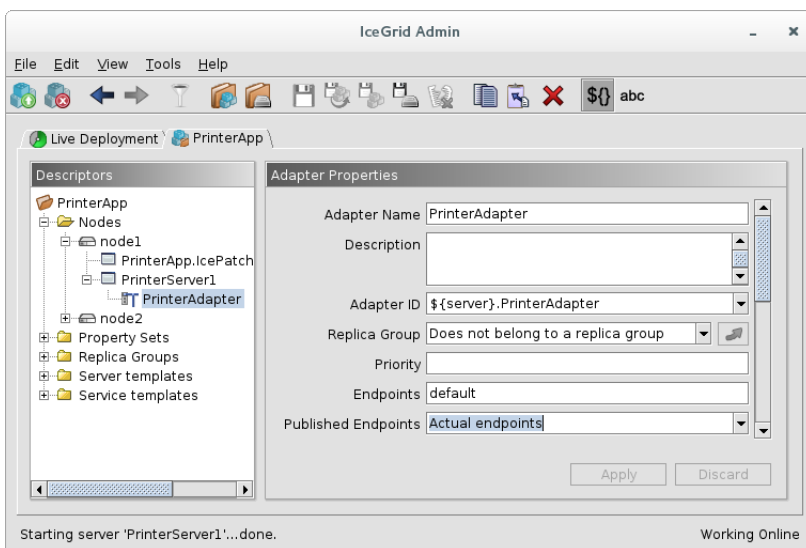


FIGURA 3P.18: IceGrid Admin: Configuración del adaptador de objetos



Nótese que el *Adapter ID* tiene un valor por defecto igual a `${server}.PrinterAdapter`. La variable `${server}` se traducirá por el nombre del servidor (`PrinterServer1` en este caso) y el resto corresponde con el nombre del adaptador de objetos. Puede parecer raro que tenga que existir un nombre interno y un identificador público. Pero esto permite crear instancias diferentes del mismo servidor.

Para ejecutar el segundo servidor, procede del mismo modo desplegando el `node2` y pulsa *Start* en el `PrinterServer2`.

3p.6.3. Ejecutando el cliente

Recuerda que el cliente acepta un argumento desde la línea de comandos: la representación textual del proxy que debe utilizar. Ahora conocemos el valor de ese parámetro, así que ya podemos ejecutar el cliente:

```
$ ./Client.py --Ice.Config=../icegrid/locator.config \
    "printer1 -t -e 1.1 @ PrinterServer1.PrinterAdapter"
```

Nótese el parámetro `--Ice.Config` en el que se le indica al cliente el fichero en el que aparece el valor de la propiedad `Ice.Default.Locator`. Sin esa referencia, el cliente no sabría cómo resolver proxies indirectos.

Mira la salida estándar del `PrinterServer1` (figura 3p.19). ¡Funciona!

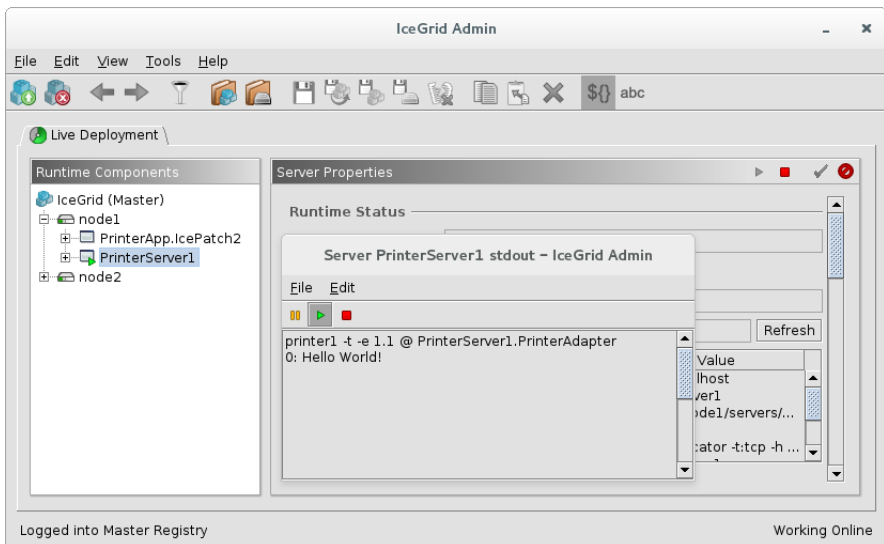


FIGURA 3P.19: IceGrid Admin: Salida estándar del servidor después de ejecutar el cliente

Sólo tenemos que usar `printer1@PrinterServer1.PrinterAdapter`. A diferencia del proxy directo, esta designación siempre será la misma, aunque el servidor se ejecute en **un nodo diferente** o el middleware elija un puerto distinto. Además,

la indirección no resulta especialmente ineficiente puesto que el cliente solo preguntará al Locator la primera vez y cacheará la respuesta. Sólo volverá a consultar al Locator si falla una invocación sobre el proxy cacheado, y todo esto de forma automática y sin la intervención del programador.



El tipo de indirección que conseguimos gracias a los proxies indirectos es lo que entendemos por **transparencia de localización**, uno de los logros más importantes en el campo de la computación distribuida heterogénea.

3p.7. Objetos bien conocidos

Algunos objetos importantes para la aplicación pueden ser incluso más fáciles de localizar. Los objetos bien conocidos pueden ser localizados simplemente por sus nombres. Es bastante sencillo conseguirlo. Ve a la pestaña `PrinterApp` y luego al adaptador `PrinterAdapter` de `PrinterServer1`. Desplaza la ventana hacia abajo hasta la tabla *Well-known objects*. Añade la identidad `printer1` y pulsa `Apply`, tal como se muestra en la figura 3p.20.

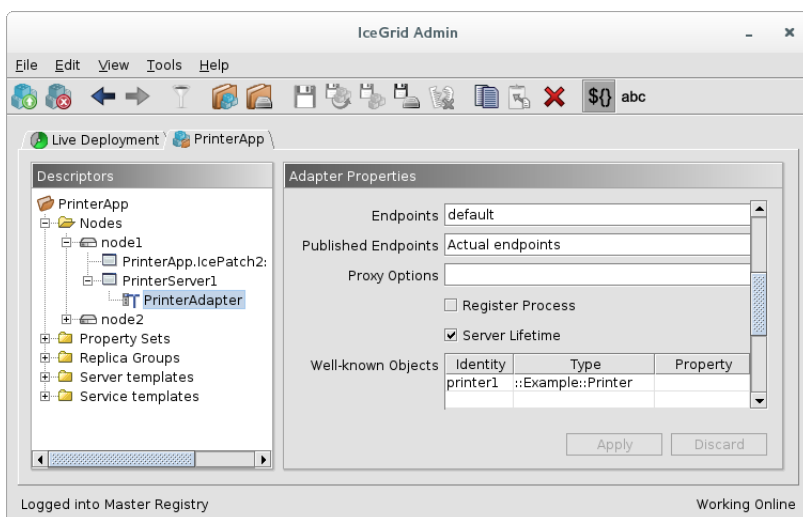


FIGURA 3P.20: IceGrid Admin: Añadiendo un objeto bien conocido

Para invocar el objeto bien conocido con el cliente simplemente debes indicar el nombre como proxy textual:

```
$ ./Client.py --Ice.Config=../icegrid/locator.config printer1
```

Mira la salida estándar de `PrinterServer1` y comprueba que también funciona. Una cuestión interesante es que ni siquiera ha sido necesario reiniciar el servidor. En este caso, la cadena `printer1` es otra forma de proxy indirecto. La notación

`identidad@adaptador` corresponde con un objeto registrado en un adaptador de objetos bien conocido (aquellos que tiene un *Adapter ID*), mientras que la notación `identidad` corresponde con objetos bien conocidos.

3p.8. Activación y desactivación implícita

A veces podemos ahorrar recursos parando los servidores ociosos hasta que se necesiten realmente. Los proxies indirectos son ideales para conseguirlo de forma automática. Si un cliente intenta contactar con un proxy indirecto por primera vez, hará una petición al Locator. Si falla, intentará contactar al Locator de nuevo.

El registro es capaz de usar las peticiones al Locator como solicitudes de activación implícita. Pulsa en la pestaña *PrinterApp* y luego en *PrinterServer1*. Despliega el menú *Activation mode* y selecciona *on-demand*. Ahora pulsa en **Apply** y en *Save to registry*.

Ahora ve a la pestaña *Live Deployment*, en el menú contextual de *PrinterServer1* pulsas *Stop*. Ahora vuelve a ejecutar el cliente invocando a `printer1`. El servidor `PrinterServer1` arrancará automáticamente en el momento en el que el cliente intenta conectar con él.

Además, es bastante conveniente desactivar los servicios ociosos pasado un tiempo de inactividad. Para aplicar esa posibilidad ve a la pestaña *PrinterApp*, pulsa en *PrinterServer1* y en la tabla de propiedades añade una nueva propiedad llamada `Ice.ServerIdleTime` con el valor '5'. Pulsar **Apply** y luego el botón *Save to registry*. Después de 5 segundos de inactividad, el servidor cerrará automáticamente.

3p.9. Depuración

Es fácil cometer algún error durante la configuración de la aplicación distribuido y puede resultar complicado si no se es sistemático encontrando el problema.

Si algo va mal primero debería revisar la salida de error estándar del programa que falla. Para que esta salida esté disponible desde la herramienta de administración se debe definir la propiedad `Ice.StdErr` y darle una ruta a un fichero tal que `${application.distrib}/server-err.txt`. Una vez que el programa se ejecute podrá acceder a su contenido por medio del menú contextual del nodo servidor en la opción *Retrieve stderr*. Del mismo modo se puede activar la salida estándar con la propiedad `Ice.StdOut`. Estas mismas propiedades pueden definirse en la configuración de los nodos por medio de los ficheros que aparecen en la sección 3p.2.2. Esto nos dará información extra muy útil para localizar y resolver posibles problemas.

3p.9.1. Evitando problemas con `IcePatch2`

Recuerda que si cambias y recompilas un programa debes volverlos a desplegar (ver § 3p.6.1). Sin embargo, es bastante frecuente olvidar reiniciar el propio servicio `IcePatch2`. Si el servicio no se reinicia no detectará los cambios.

Una buena solución para este problema es configurar siempre la propiedad `Ice.ServerIdleTime` en la instancia del servidor `IcePatch2` asignándole un valor pequeño.

3p.9.2. Descripción de la aplicación

Recuerda que la aplicación queda definida en un fichero XML. Toda la configuración que se realiza en la definición de la aplicación en `icegridgui` queda almacenada en un fichero bastante legible/editable manualmente. Para nuestra aplicación ese fichero (obviando las plantillas de los servicios comunes) se muestra en el listado 3p.9.

LISTADO 3P.9: Descripción de la aplicación `PrinterApp` en XML
`icegrid/printerapp-py.xml`

```
<node name="node1">
  <server-instance template="IcePatch2" directory="/tmp/ice-hello/py/dist">
    <properties>
      <property name="Ice.ServerIdleTime" value="5"/>
    </properties>
  </server-instance>
  <server id="PrinterServer1" activation="manual" exe="./Server.py" pwd="${application.distrib}">
    <properties>
      <property name="Ice.StdOut" value="${application.distrib}/server-out.txt"/>
    </properties>
    <adapter name="PrinterAdapter" endpoints="default" id="${server}.PrinterAdapter">
      <object identity="printer1" type="::Example::Printer"/>
    </adapter>
  </server>
</node>
<node name="node2">
  <server id="PrinterServer2" activation="manual" exe="./Server-UUID.py" pwd="${application.distrib}">
```

3p.10. Receta

A continuación se listan a modo de resumen los pasos más importantes para realizar un despliegue cliente/servidor sencillo como el que se ha descrito:

- Crear un fichero de configuración para el `Registry` y para cada uno de los nodos del grid (§3p.2.2).
- Arrancar un `icegridnode` por cada nodo con su configuración correspondiente (§3p.2.3).
- Crear la aplicación distribuida (§3p.3).
- Crear un nodo en la aplicación para cada nodo ejecutado.
- Compilar los programas, copiar los binarios a otro directorio y ejecutar `ice-patch2calc`.
- Crear un servidor de `IcePatch2` (§3p.4) y configurarlo para desplegar el directorio del paso anterior.
- Configurar ese servidor de `IcePatch2` como el `IcePatch2` por defecto de la aplicación distribuida (figura 3p.14).

- Crear los servidores IceGrid para ambos programas (3p.5) en sus respectivos nodos.
- Desplegar la aplicación (§ 3p.6.1).
- Arrancar los servidores (§ 3p.6.2).
- Ejecutar el cliente (§ 3p.6.3).

3p.11. Ejercicios

- E 3p.01** Invoca el servidor desplegado con IceGrid ejecutando el cliente desde otro servidor IceGrid.
- E 3p.02** Modifica el código del cliente para que invoque a los dos servidores.
- E 3p.03** El servidor del node2 tiene una identidad aleatoria. Averigua cómo fijar la identidad del objeto por medio de una propiedad. Cambia el programa que ejecutan ambos servidores para ejecutar la nueva versión (ambos nodos el mismo programa) e indica por medio de una propiedad que sus identidades son respectivamente «printer1» y «printer2».
- E 3p.04** Define nodos adicionales, crea una plantilla¹ para el servidor Printer-Server. Arranca nodos adicionales y, usando la plantilla, instancia varias impresoras (objetos *Printer*) en ellos. Invoca esas impresoras con el cliente original desde la consola. Consulta el ejemplo `demopy/IceGrid/simple` de la distribución de ICE².
- E 3p.05** Explora las posibilidades de `icegridadmin`³, la herramienta de administración en línea de comandos, para obtener información y modificar la configuración de la aplicación. ¿Qué comando debes introducir para obtener la lista de nodos? ¿Y la lista de servidores?
- E 3p.06** Repite el tutorial de este capítulo usando un computador diferente para cada nodo. Esos computadores pueden ser virtuales (se aconseja `VirtualBox`).

¹<http://doc.zeroc.com/display/Ice/IceGrid+Templates>

²<http://www.zeroc.com/download/Ice/3.5/Ice-3.5.0-demos.tar.gz>

³<http://doc.zeroc.com/display/Ice/icegridadmin+Command+Line+Tool>

Propagación de eventos

[C++]

Uno de los servicios más sencillos y útiles de ICE es IceStorm. Se trata de un servicio de propagación de eventos, pero a diferencia de otros servicios similares en otros middlewares, como NotificationService de CORBA, DDS o JMS (Java Message Service), IceStorm propaga invocaciones a métodos en lugar de objetos o estructuras de datos. Es, a todos los efectos, una implementación distribuida del patrón de diseño publicación/suscripción, más conocido como «observador» [GHJV94].

Para utilizar el servicio de eventos se requiere un proxy a un *TopicManager*. Este objeto permite listar, crear y destruir canales (*topics*). Los canales también son objetos, que residen en el mismo servidor en el que se encuentre el *TopicManager*.

Para recibir eventos procedentes de un canal es necesario suscribir un objeto, es decir, darle al canal la referencia (el proxy) del objeto. Para enviar invocaciones a un canal se necesita el proxy al *publicador del canal*. El publicador es un objeto especial que no tiene interfaz concreta. Se puede invocar cualquier método sobre él. El canal enviará esa invocación a todos sus suscriptores. Sin embargo, si el suscriptor no implementa la interfaz que corresponde con invocación, elevará una excepción y el canal lo des-suscribirá inmediatamente. Por esa razón se deben crear canales diferentes para cada interfaz de la que se quiera propagar eventos.

Los canales solo pueden propagar invocaciones oneway o datagram, es decir, que no tengan valores de retorno, dado que sería complejo tratar las respuestas de todos los suscriptores hacia un mismo origen. El canal es un intermediario que desacopla completamente a publicadores y suscriptores. Todos conocen el canal, pero no se conocen entre sí.

Veamos su funcionamiento con un ejemplo. Se crea un canal de eventos para la interfaz *Example::Printer* del capítulo anterior. Se empieza por codificar el suscriptor, que es en esencia un servidor. El código completo para el suscriptor se muestra el en siguiente listado:

LISTADO 4C.1: Suscriptor C++
[cpp-icestorm/subscriber.cpp](#)

```
#include <Ice/Application.h>
#include <IceStorm/IceStorm.h>
```

```

#include <IceUtil/UUID.h>
#include "Printer.h"

using namespace std;
using namespace Ice;
using namespace IceStorm;
using namespace Example;

class PrinterI : public Printer {
public:
    void write(const string& message, const Current& current) {
        cout << "Event received: " << message << endl;
    }
};

class Subscriber : public Application {
    TopicManagerPrx get_topic_manager() {
        string key = "IceStorm.TopicManager.Proxy";
        ObjectPrx proxy = communicator()->propertyToProxy(key);
        if (!proxy) {
            cerr << "property " << key << " not set." << endl;
            return 0;
        }

        cout << "Using IceStorm in: '" << key << "' " << endl;
        return TopicManagerPrx::checkedCast(proxy);
    }

public:
    virtual int run(int argc, char* argv[]) {
        TopicManagerPrx topic_mgr = get_topic_manager();
        if (!topic_mgr) {
            cerr << appName() << ": invalid proxy" << endl;
            return EXIT_FAILURE;
        }

        ObjectPtr servant = new PrinterI;
        ObjectAdapterPtr adapter = \
            communicator()->createObjectAdapter("PrinterAdapter");
        ObjectPrx subscriber = adapter->addWithUUID(servant);

        string topic_name = "PrinterTopic";
        TopicPrx topic;
        try {
            topic = topic_mgr->retrieve(topic_name);
        } catch(const NoSuchTopic& e) {
            topic = topic_mgr->create(topic_name);
        }

        topic->subscribeAndGetPublisher(QoS(), subscriber);
        cout << "Waiting events... " << subscriber << endl;

        adapter->activate();
        shutdownOnInterrupt();
        communicator()->waitForShutdown();

        topic->unsubscribe(subscriber);

        return EXIT_SUCCESS;
    }
};

int main(int argc, char* argv[]) {
    Subscriber app;
    return app.main(argc, argv);
}

```


Lo primero es conseguir el proxy al *TopicManager* (**línea 35**), que se obtiene a partir de una propiedad en el fichero de configuración (método privado `get_topic_manager()` del servidor).

A continuación se crea un sirviente y un adaptador de objetos, y se añade el sirviente al adaptador (**líneas 41–44**).

Después se obtiene una referencia al canal `PrinterTopic` (**línea 49**), que debería existir. Si no existe se produce una excepción (*NoSuchTopic*), que es capturada y su manejador crea el canal (**línea 51**). En cualquier caso, después de obtener el canal, se suscribe el objeto (**línea 54**). Por último, se activa el adaptador y el servidor queda a la espera de recibir eventos.

Si el suscriptor es como un servidor, el publicador es como un cliente. Solo se muestra el método `run()` puesto que el resto del código es prácticamente idéntico al suscriptor:

LISTADO 4C.2: Publicador C++
`cpp-icestorm/publisher.cpp`

```
virtual int run(int argc, char*[]) {
    TopicManagerPrx topic_mgr = get_topic_manager();
    if (!topic_mgr) {
        cerr << appName() << ": invalid proxy" << endl;
        return EXIT_FAILURE;
    }

    string topic_name = "PrinterTopic";
    TopicPrx topic;
    try {
        topic = topic_mgr->retrieve(topic_name);
    } catch (const NoSuchTopic& e) {
        cerr << appName()
            << ": no such topic found, creating" << endl;
        topic = topic_mgr->create(topic_name);
    }

    assert(topic);

    ObjectPrx publisher = topic->getPublisher();
    PrinterPrx printer = PrinterPrx::uncheckedCast(publisher);

    cout << "publishing 10 'Hello World' events" << endl;
    for(char i='0'; i <= '9'; ++i)
        printer->write(string("Hello World ") + i + "!");

    return EXIT_SUCCESS;
}
```

Lo más interesante es la obtención del publicador del canal a partir de la referencia al canal (**línea 20**) y el *downcast* para poder invocar sobre él el método de la interfaz *Example::Printer* (**línea 21**). Nótese que este molde usa la modalidad `uncheckedCast()` dado que la comprobación fallaría puesto que el publicador no implementa realmente ninguna interfaz.

Por último se utiliza el proxy printer para enviar diez eventos con la cadena "Hello World!".

4c.1. Arranque del servicio

IceStorm está implementado como un servicio ICE. Los servicios son librerías dinámicas que deben ser lanzadas con el servidor de aplicaciones, que se llama IceBox. Vemos la configuración de IceBox en el listado 4c.3.

LISTADO 4C.3: Configuración de IceBox para lanzar IceStorm
[cpp-icestorm/icebox.config](#)

```
IceBox.Service.IceStorm=IceStormService:createIceStorm --Ice.Config=icestorm.config
```

Sólo se definen dos propiedades: la carga del servicio IceStorm y los endpoints del gestor remoto de IceBox. La configuración de IceStorm a la que se hace referencia (`icestorm.config`) se muestra en el listado 4c.4.

LISTADO 4C.4: Configuración de IceStorm
[cpp-icestorm/icestorm.config](#)

```
IceStormAdmin.TopicManager.Default=IceStorm/TopicManager:tcp -p 10000
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 2000
IceStorm.Flush.Timeout=2000
Freeze.DbEnv.IceStorm.DbHome=db
```

Las **líneas 1–3** especifican los endpoints para los adaptadores del *TopicManager*, el objeto de administración, y los publicadores de los canales. La **línea 5** indica el nombre del directorio donde se almacenará la configuración del servicio (que es persistente de forma automática).

Una vez configurado se puede lanzar el servicio y probar el ejemplo. Lo primero es arrancar IceBox (en *background*):

```
$ icebox --Ice.Config=icebox.config &
```

A continuación se puede usar la herramienta de administración de IceStorm en línea de comando para crear el canal, asumiendo que no existe:

```
$ icestormadmin --Ice.Config=icestorm.config -e "create PrinterTopic"
```

La configuración del suscriptor es:

LISTADO 4C.5: Configuración del suscriptor
[cpp-icestorm/subscriber.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
PrinterAdapter.Endpoints=tcp
```

Y con eso ya se puede arrancar el suscriptor:

```
$ ./subscriber --Ice.Config=subscriber.config
```

La configuración para el publicador es:

LISTADO 4C.6: Configuración de publicador
[cpp-icestorm/publisher.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
```

Y por último en una consola diferente se ejecuta el publicador:

```
$ ./publisher --Ice.Config=publisher.config
```

Hecho lo cual aparecerán los eventos al llegar al suscriptor:

```
0: Event received: Hello World!
1: Event received: Hello World!
2: Event received: Hello World!
3: Event received: Hello World!
4: Event received: Hello World!
5: Event received: Hello World!
6: Event received: Hello World!
7: Event received: Hello World!
8: Event received: Hello World!
9: Event received: Hello World!
```

Es de destacar el hecho de que los eventos estén modelados como invocaciones. De ese modo es posible eliminar temporalmente el canal de eventos e invocar directamente a un solo suscriptor durante las fases iniciales de modelado del protocolo y desarrollo de prototipos.

4c.2. Federación de canales

La federación de canales de eventos representa un mecanismo que, apropiadamente usado, puede aportar grandes ventajas en cuanto a escalabilidad y tolerancia a fallos a un determinado entorno. Generalmente los servicios de eventos asociados a las plataformas distribuidas contemplan este mecanismo para asociar canales de eventos y permitir escalar los entornos de forma que un evento generado en un canal de eventos en particular es transmitido a sus canales federados.

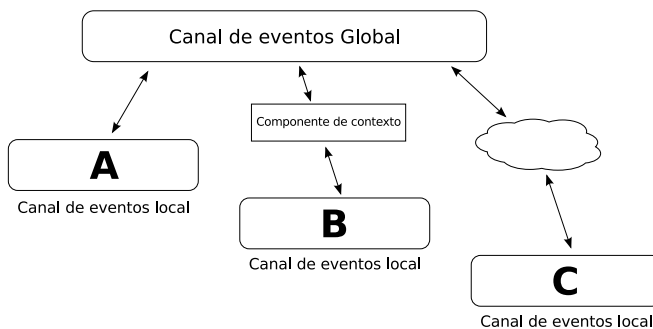


FIGURA 4C.1: Esquema de federación de canales

Se contemplan tres formas de federación posibles:

Federación directa

La federación directa es la más básica y generalmente está soportada por el servicio de eventos del middleware. Este tipo de federación debe ser usada entre canales de eventos del mismo tipo (es decir, mensajes con la misma estructura) y con el objetivo general de agrupar eventos por funcionalidad en un entorno y configurar entornos tolerantes a fallos.

En la figura 4c.1, el canal de eventos A se federa al canal de eventos Global por medio del servicio básico. Si el canal de eventos A y el canal de eventos Global están soportados por distintas instancias del servicio de eventos, una caída del servicio de eventos que da soporte al canal A no afecta al canal de eventos Global y viceversa.

Federación indirecta

Se realiza a través de un componente de contexto. Al poner un elemento intermedio entre los dos canales de eventos se permite la implementación de políticas complejas en el paso de eventos de un canal a otro. En la figura 4c.1 el canal de eventos B está conectado a través de un componente de contexto (que es un objeto distribuido) al canal de eventos Global.

Un ejemplo de aplicación de este tipo de federación podría ser la gestión de los sensores de presencia en un entorno laboral (ej: fábricas, oficinas, plantas, etc.). En horario diurno cuando los trabajadores se encuentran realizando su jornada laboral se puede indicar a los componentes de contexto que no retransmitan los eventos de presencia ya que son continuos y están asociados al ir y venir de operarios por la instalación. En horario nocturno esta política cambia y se indica a dichos componentes que retransmitan dichos eventos ya que los mismos eventos de presencia son considerados como eventos de intrusión. Es necesario resaltar que los propios servicios de eventos permiten establecer políticas básicas por lo que este escenario está contemplado para la implementación de políticas relativamente complejas difíciles de abordar con el mecanismo de políticas básicas. Este escenario permite federar canales de eventos de distinto tipo y filtrar la información que debe ser manejada por instancias superiores.

Federación remota

Se considera una federación remota (canal de eventos C en la figura 4c.1) cuando los eventos retransmitidos deben atravesar secciones de red no pertenecientes a la organización (ej: a través de Internet) o entornos poco seguros (ej: la vía pública). Esta federación se puede realizar a su vez de forma directa o de forma indirecta (ej: a través de un componente de contexto localizado en el entorno al que pertenece el canal de eventos C). Es necesario resaltar que en estos escenarios se debe establecer una conexión segura (SSL, VPN, Autenticación, etc.) para mantener la confidencialidad de las comunicaciones.

Los mecanismos de federación de canales de eventos deben ser utilizados para segmentar la funcionalidad del entorno atendiendo a razones de despliegue físicas (entornos autónomos y estancos) o de funcionalidad lógica (jerarquizando el paso de mensajes).

4c.2.1. Propagación entre canales de eventos federados

La política de propagación de mensajes entre canales de eventos federados es un aspecto crítico para que una arquitectura de eventos sea eficiente y no sobrecargue la red.

Uno de los aspectos mas importantes a la hora de propagar mensajes entre canales de eventos federados es evitar establecer bucles ya sea de forma directa o indirecta que originen mensajes que pasan de un canal a otro de forma recurrente.

Para evitar este tipo de bucles se define el coste de un mensaje. El coste asociado a un mensaje es un valor entero que nos determinará la política de propagación que debemos seguir con ese mensaje.

Para los distintos tipos de federaciones, definimos la siguiente política:

- Los mensajes nunca son propagados mas allá de un solo enlace.
- Los enlaces deben tener un coste asociado, aquellos canales de eventos que tienen otros canales federados, deben propagar un mensaje a dichos canales si el coste que lleva asociado el mensaje es igual o inferior al coste asociado al enlace.
- Si un enlace establece un coste 0, entonces todos los mensajes serán enviados a través de dicho enlace sin importar el coste del mensaje.
- Si un mensaje tiene un coste 0 será enviado a todos los canales de eventos federados sin importar el coste del mensaje.
- Los enlaces entre canales de eventos con un coste 0 deben ser utilizados para difundir de forma automática todos los mensajes de un canal a otro de forma directa (e.j para modelar escenarios mas extensos).
- La federación directa, tal y como podemos ver en la siguiente figura, es útil para modularizar escenarios y agrupar dispositivos que generan el mismo tipo de mensajes en función de su localización física, agrupación funcional, etc.

Propagación de eventos

[Java]

Uno de los servicios más sencillos y útiles de ICE es IceStorm. Se trata de un servicio de propagación de eventos, pero a diferencia de otros servicios similares en otros middlewares, como NotificationService de CORBA, DDS o JMS, IceStorm propaga invocaciones a métodos en lugar de objetos o estructuras de datos. Es, a todos los efectos, una implementación distribuida del patrón de diseño publicación/suscripción, más conocido como «observador» [GHJV94].

Para utilizar el servicio de eventos se requiere un proxy a un *TopicManager*. Este objeto permite listar, crear y destruir canales (*topics*). Los canales también son objetos, que residen en el mismo servidor en el que se encuentre el *TopicManager*.

Para recibir eventos procedentes de un canal es necesario suscribir un objeto, es decir, darle al canal la referencia (el proxy) del objeto. Para enviar invocaciones a un canal se necesita el proxy al *publicador del canal*. El publicador es un objeto especial que no tiene interfaz concreta. Se puede invocar cualquier método sobre él. El canal enviará esa invocación a todos sus suscriptores. Sin embargo, si el suscriptor no implementa la interfaz que corresponde con invocación, elevará una excepción y el canal lo des-suscribirá inmediatamente. Por esa razón se deben crear canales diferentes para cada interfaz de la que se quiera propagar eventos.

Los canales solo pueden propagar invocaciones oneway o datagram, es decir, que no tengan valores de retorno, dado que sería complejo tratar las respuestas de todos los suscriptores hacia un mismo origen. El canal es un intermediario que desacopla completamente a publicadores y suscriptores. Todos conocen el canal, pero no se conocen entre sí.

Veamos su funcionamiento con un ejemplo. Se crea un canal de eventos para la interfaz *Example.Printer* del capítulo anterior. Se empieza por codificar el suscriptor, que es en esencia un servidor. El código completo para el suscriptor se muestra el en siguiente listado:

LISTADO 4J.1: Suscriptor Java
`java-icestorm/Subscriber.java`

```
import java.util.HashMap;  
import Ice.*;
```

```
import IceStorm.*;
import Example.*;

public class Subscriber extends Application {
    public class PrinterI extends _PrinterDisp {
        public void write(String message, Ice.Current current) {
            System.out.println(String.format("Event received: %s", message));
        }
    }

    public int run(String[] args) {
        String key = "IceStorm.TopicManager.Proxy";
        ObjectPrx prx = communicator().propertyToProxy(key);
        TopicManagerPrx manager = TopicManagerPrxHelper.checkedCast(prx);

        if (manager == null) {
            System.err.println("invalid proxy");
            return 1;
        }

        PrinterI servant = new PrinterI();
        ObjectAdapter adapter =
            communicator().createObjectAdapter("PrinterAdapter");
        ObjectPrx subscriber = adapter.addWithUUID(servant);

        String topic_name = "PrinterTopic";
        TopicPrx topic;
        try {
            topic = manager.retrieve(topic_name);
        } catch (NoSuchTopic e) {
            try {
                topic = manager.create(topic_name);
            } catch (TopicExists ex) {
                System.err.println("Temporary failure, try again.");
                return 1;
            }
        }

        try {
            topic.subscribeAndGetPublisher(null, subscriber);
        } catch (AlreadySubscribed e) {
            e.printStackTrace();
            return 1;
        } catch (BadQoS e) {
            e.printStackTrace();
            return 1;
        }

        System.out.println("Waiting events... ");

        adapter.activate();
        shutdownOnInterrupt();
        communicator().waitForShutdown();

        topic.unsubscribe(subscriber);

        return 0;
    }

    public static void main(String[] args) {
        Subscriber app = new Subscriber();
        int status = app.main("Subscriber", args);
        System.exit(status);
    }
}
```


Lo primero es conseguir el proxy al *TopicManager* (**línea 16**), que se obtiene a partir de una propiedad en el fichero de configuración (método privado `get_topic_manager()` del servidor).

A continuación se crea un sirviente y un adaptador de objetos, y se añade el sirviente al adaptador (**líneas 23–26**).

Después se obtiene una referencia al canal `PrinterTopic` (**línea 31**), que debería existir. Si no existe se produce una excepción (*NoSuchTopic*), que es capturada y su manejador crea el canal (**línea 34**). En cualquier caso, después de obtener el canal, se suscribe el objeto (**línea 42**). Por último, se activa el adaptador y el servidor queda a la espera de recibir eventos.

Si el suscriptor es como un servidor, el publicador es como un cliente. Solo se muestra el método `run()` puesto que el resto del código es prácticamente idéntico al suscriptor:

LISTADO 4J.2: Publicador Java
`java-icestorm/Publisher.java`

```
try {
    topic = manager.retrieve(topicName);
} catch(NoSuchTopic e) {
    try {
        topic = manager.create(topicName);
    } catch(TopicExists ex) {
        System.err.println("Temporary failure, try again.");
        return 1;
    }
}

ObjectPrx publisher = topic.getPublisher();
PrinterPrx printer = PrinterPrxHelper.uncheckedCast(publisher);

System.out.println("publishing 10 'Hello World' events");
for(int i=0; i < 10; i++)
    printer.write(String.format("Hello World %s!", i));

return 0;
}

public static void main(String[] args) {
    Publisher app = new Publisher();
    int status = app.main("Publisher", args);
    System.exit(status);
}
```

Lo más interesante es la obtención del publicador del canal a partir de la referencia al canal (**línea 12**) y el *downcast* para poder invocar sobre él el método de la interfaz *Example::Printer* (**línea 13**). Nótese que este molde usa la modalidad `uncheckedCast()` dado que la comprobación fallaría puesto que el publicador no implementa realmente ninguna interfaz.

Por último se utiliza el proxy printer para enviar diez eventos con la cadena "Hello World!".

4j.1. Arranque del servicio

IceStorm está implementado como un servicio ICE. Los servicios son librerías dinámicas que deben ser lanzadas con el servidor de aplicaciones, que se llama IceBox. Vemos la configuración de IceBox en el listado 4j.3.

LISTADO 4J.3: Configuración de IceBox para lanzar IceStorm
[cpp-icestorm/icebox.config](#)

```
IceBox.Service.IceStorm=IceStormService:createIceStorm --Ice.Config=icestorm.config
```

Sólo se definen dos propiedades: la carga del servicio IceStorm y los endpoints del gestor remoto de IceBox. La configuración de IceStorm a la que se hace referencia (`icestorm.config`) se muestra en el listado 4j.4.

LISTADO 4J.4: Configuración de IceStorm
[cpp-icestorm/icestorm.config](#)

```
IceStormAdmin.TopicManager.Default=IceStorm/TopicManager:tcp -p 10000
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 2000
IceStorm.Flush.Timeout=2000
Freeze.DbEnv.IceStorm.DbHome=db
```

Las **líneas 1–3** especifican los endpoints para los adaptadores del *TopicManager*, el objeto de administración, y los publicadores de los canales. La **línea 5** indica el nombre del directorio donde se almacenará la configuración del servicio (que es persistente de forma automática).

Una vez configurado se puede lanzar el servicio y probar el ejemplo. Lo primero es arrancar IceBox (en *background*):

```
$ icebox --Ice.Config=icebox.config &
```

A continuación se puede usar la herramienta de administración de IceStorm en línea de comando para crear el canal, asumiendo que no existe:

```
$ icestormadmin --Ice.Config=icestorm.config -e "create PrinterTopic"
```

La configuración del suscriptor es:

LISTADO 4J.5: Configuración del suscriptor
[cpp-icestorm/subscriber.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
PrinterAdapter.Endpoints=tcp
```

Y con eso ya se puede arrancar el subscritor:

```
$ java -classpath /usr/share/java/Ice.jar:. Subscriber --Ice.Config=subscriber.config
```

La configuración para el publicador es:

LISTADO 4J.6: Configuración de publicador
[cpp-icestorm/publisher.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
```

Y por último en una consola diferente se ejecuta el publicador:

```
$ java -classpath /usr/share/java/Ice.jar:. Publisher --Ice.Config=publisher.config
```

Hecho lo cual aparecerán los eventos al llegar al suscriptor:

```
0: Event received: Hello World!
1: Event received: Hello World!
2: Event received: Hello World!
3: Event received: Hello World!
4: Event received: Hello World!
5: Event received: Hello World!
6: Event received: Hello World!
7: Event received: Hello World!
8: Event received: Hello World!
9: Event received: Hello World!
```

Es de destacar el hecho de que los eventos estén modelados como invocaciones. De ese modo es posible eliminar temporalmente el canal de eventos e invocar directamente a un solo suscriptor durante las fases iniciales de modelado del protocolo y desarrollo de prototipos.

4j.2. Federación de canales

La federación de canales de eventos representa un mecanismo que, apropiadamente usado, puede aportar grandes ventajas en cuanto a escalabilidad y tolerancia a fallos a un determinado entorno. Generalmente los servicios de eventos asociados a las plataformas distribuidas contemplan este mecanismo para asociar canales de eventos y permitir escalar los entornos de forma que un evento generado en un canal de eventos en particular es transmitido a sus canales federados.

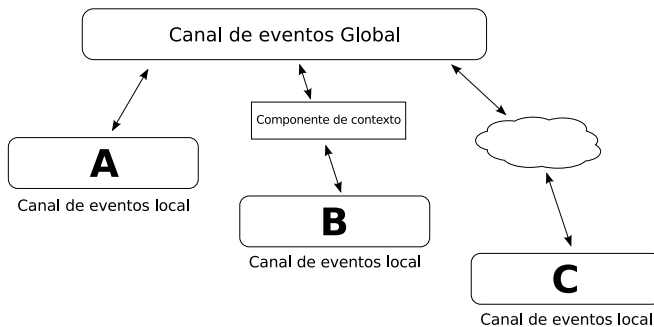


FIGURA 4J.1: Esquema de federación de canales

Se contemplan tres formas de federación posibles:

Federación directa

La federación directa es la más básica y generalmente está soportada por el servicio de eventos del middleware. Este tipo de federación debe ser usada entre canales de eventos del mismo tipo (es decir, mensajes con la misma estructura) y con el objetivo general de agrupar eventos por funcionalidad en un entorno y configurar entornos tolerantes a fallos.

En la figura 4j.1, el canal de eventos A se federa al canal de eventos Global por medio del servicio básico. Si el canal de eventos A y el canal de eventos Global están soportados por distintas instancias del servicio de eventos, una caída del servicio de eventos que da soporte al canal A no afecta al canal de eventos Global y viceversa.

Federación indirecta

Se realiza a través de un componente de contexto. Al poner un elemento intermedio entre los dos canales de eventos se permite la implementación de políticas complejas en el paso de eventos de un canal a otro. En la figura 4j.1 el canal de eventos B está conectado a través de un componente de contexto (que es un objeto distribuido) al canal de eventos Global.

Un ejemplo de aplicación de este tipo de federación podría ser la gestión de los sensores de presencia en un entorno laboral (ej: fábricas, oficinas, plantas, etc.). En horario diurno cuando los trabajadores se encuentran realizando su jornada laboral se puede indicar a los componentes de contexto que no retransmitan los eventos de presencia ya que son continuos y están asociados al ir y venir de operarios por la instalación. En horario nocturno esta política cambia y se indica a dichos componentes que retransmitan dichos eventos ya que los mismos eventos de presencia son considerados como eventos de intrusión. Es necesario resaltar que los propios servicios de eventos permiten establecer políticas básicas por lo que este escenario está contemplado para la implementación de políticas relativamente complejas difíciles de abordar con el mecanismo de políticas básicas. Este escenario permite federar canales de eventos de distinto tipo y filtrar la información que debe ser manejada por instancias superiores.

Federación remota

Se considera una federación remota (canal de eventos C en la figura 4j.1) cuando los eventos retransmitidos deben atravesar secciones de red no pertenecientes a la organización (ej: a través de Internet) o entornos poco seguros (ej: la vía pública). Esta federación se puede realizar a su vez de forma directa o de forma indirecta (ej: a través de un componente de contexto localizado en el entorno al que pertenece el canal de eventos C). Es necesario resaltar que en estos escenarios se debe establecer una conexión segura (SSL, VPN, Autenticación, etc.) para mantener la confidencialidad de las comunicaciones.

Los mecanismos de federación de canales de eventos deben ser utilizados para segmentar la funcionalidad del entorno atendiendo a razones de despliegue físicas (entornos autónomos y estancos) o de funcionalidad lógica (jerarquizando el paso de mensajes).

4j.2.1. Propagación entre canales de eventos federados

La política de propagación de mensajes entre canales de eventos federados es un aspecto crítico para que una arquitectura de eventos sea eficiente y no sobrecargue la red.

Uno de los aspectos mas importantes a la hora de propagar mensajes entre canales de eventos federados es evitar establecer bucles ya sea de forma directa o indirecta que originen mensajes que pasan de un canal a otro de forma recurrente.

Para evitar este tipo de bucles se define el coste de un mensaje. El coste asociado a un mensaje es un valor entero que nos determinará la política de propagación que debemos seguir con ese mensaje.

Para los distintos tipos de federaciones, definimos la siguiente política:

- Los mensajes nunca son propagados mas allá de un solo enlace.
- Los enlaces deben tener un coste asociado, aquellos canales de eventos que tienen otros canales federados, deben propagar un mensaje a dichos canales si el coste que lleva asociado el mensaje es igual o inferior al coste asociado al enlace.
- Si un enlace establece un coste 0, entonces todos los mensajes serán enviados a través de dicho enlace sin importar el coste del mensaje.
- Si un mensaje tiene un coste 0 será enviado a todos los canales de eventos federados sin importar el coste del mensaje.
- Los enlaces entre canales de eventos con un coste 0 deben ser utilizados para difundir de forma automática todos los mensajes de un canal a otro de forma directa (e.j para modelar escenarios mas extensos).
- La federación directa, tal y como podemos ver en la siguiente figura, es útil para modularizar escenarios y agrupar dispositivos que generan el mismo tipo de mensajes en función de su localización física, agrupación funcional, etc.

Propagación de eventos

[Python]

Uno de los servicios más sencillos y útiles de ICE es IceStorm. Se trata de un servicio de propagación de eventos, pero a diferencia de otros servicios similares en otros middlewares, como NotificationService de CORBA, DDS o JMS, IceStorm propaga invocaciones a métodos en lugar de objetos o estructuras de datos. Es, a todos los efectos, una implementación distribuida del patrón de diseño publicación/suscripción, más conocido como «observador» [GHJV94].

Para utilizar el servicio de eventos se requiere un proxy a un *TopicManager*. Este objeto permite listar, crear y destruir canales (*topics*). Los canales también son objetos, que residen en el mismo servidor en el que se encuentre el *TopicManager*.

Para recibir eventos procedentes de un canal es necesario suscribir un objeto, es decir, darle al canal la referencia (el proxy) del objeto. Para enviar invocaciones a un canal se necesita el proxy al *publicador del canal*. El publicador es un objeto especial que no tiene interfaz concreta. Se puede invocar cualquier método sobre él. El canal enviará esa invocación a todos sus suscriptores. Sin embargo, si el suscriptor no implementa la interfaz que corresponde con invocación, elevará una excepción y el canal lo des-suscribirá inmediatamente. Por esa razón se deben crear canales diferentes para cada interfaz de la que se quiera propagar eventos.

Los canales solo pueden propagar invocaciones oneway o datagram, es decir, que no tengan valores de retorno, dado que sería complejo tratar las respuestas de todos los suscriptores hacia un mismo origen. El canal es un intermediario que desacopla completamente a publicadores y suscriptores. Todos conocen el canal, pero no se conocen entre sí.

Veamos su funcionamiento con un ejemplo. Se crea un canal de eventos para la interfaz del capítulo anterior. Se empieza por codificar el suscriptor, que es en esencia un servidor. El código completo para el suscriptor se muestra en el siguiente listado:

LISTADO 4P.1: Suscriptor Python
[py-icestorm/subscriber.py](#)

```
#!/usr/bin/python -u
# -*- coding: utf-8 -*-
```

```
import sys
import Ice, IceStorm
Ice.loadSlice('Printer.ice')
import Example

class PrinterI(Example.Printer):
    def write(self, message, current=None):
        print("Event received: {0}".format(message))
        sys.stdout.flush()

class Subscriber(Ice.Application):
    def get_topic_manager(self):
        key = 'IceStorm.TopicManager.Proxy'
        proxy = self.communicator().propertyToProxy(key)
        if proxy is None:
            print "property", key, "not set"
            return None

        print("Using IceStorm in: '%s'" % key)
        return IceStorm.TopicManagerPrx.checkedCast(proxy)

    def run(self, argv):
        topic_mgr = self.get_topic_manager()
        if not topic_mgr:
            print ': invalid proxy'
            return 2

        ic = self.communicator()
        servant = PrinterI()
        adapter = ic.createObjectAdapter("PrinterAdapter")
        subscriber = adapter.addWithUUID(servant)

        topic_name = "PrinterTopic"
        qos = {}
        try:
            topic = topic_mgr.retrieve(topic_name)
        except IceStorm.NoSuchTopic:
            topic = topic_mgr.create(topic_name)

        topic.subscribe(qos, subscriber)
        print 'Waiting events...', subscriber

        adapter.activate()
        self.shutdownOnInterrupt()
        ic.waitForShutdown()

        topic.unsubscribe(subscriber)

    return 0

sys.exit(Subscriber().main(sys.argv))
```

Lo primero es conseguir el proxy al *TopicManager* (línea 28), que se obtiene a partir de una propiedad en el fichero de configuración (método privado `get_topic_manager()` del servidor).

A continuación se crea un sirviente y un adaptador de objetos, y se añade el sirviente al adaptador (líneas 34–36).

Después se obtiene una referencia al canal `PrinterTopic` (línea 41), que debería existir. Si no existe se produce una excepción (*NoSuchTopic*), que es capturada y su manejador crea el canal (línea 43). En cualquier caso, después de obtener

el canal, se suscribe el objeto (**línea 45**). Por último, se activa el adaptador y el servidor queda a la espera de recibir eventos.

Si el suscriptor es como un servidor, el publicador es como un cliente. Solo se muestra el método `run()` puesto que el resto del código es prácticamente idéntico al suscriptor:

LISTADO 4P.2: Publicador Python
`py-icestorm/publisher.py`

```
def run(self, argv):
    topic_mgr = self.get_topic_manager()
    if not topic_mgr:
        print ': invalid proxy'
        return 2

    topic_name = "PrinterTopic"
    try:
        topic = topic_mgr.retrieve(topic_name)
    except IceStorm.NoSuchTopic:
        print "no such topic found, creating"
        topic = topic_mgr.create(topic_name)

    publisher = topic.getPublisher()
    printer = Example.PrinterPrx.uncheckedCast(publisher)

    print "publishing 10 'Hello World' events"
    for i in range(10):
        printer.write("Hello World %!" % i)
```

Lo más interesante es la obtención del publicador del canal a partir de la referencia al canal (**línea 15**) y el *downcast* para poder invocar sobre él el método de la interfaz `Example::Printer` (**línea 16**). Nótese que este molde usa la modalidad `uncheckedCast()` dado que la comprobación fallaría puesto que el publicador no implementa realmente ninguna interfaz.

Por último se utiliza el proxy printer para enviar diez eventos con la cadena "Hello World!".

4p.1. Arranque del servicio

IceStorm está implementado como un servicio ICE. Los servicios son librerías dinámicas que deben ser lanzadas con el servidor de aplicaciones, que se llama IceBox. Vemos la configuración de IceBox en el listado 4p.3.

LISTADO 4P.3: Configuración de IceBox para lanzar IceStorm
`cpp-icestorm/icebox.config`

```
IceBox.Service.IceStorm=IceStormService:createIceStorm --Ice.Config=icestorm.config
```

Sólo se definen dos propiedades: la carga del servicio IceStorm y los endpoints del gestor remoto de IceBox. La configuración de IceStorm a la que se hace referencia (`icestorm.config`) se muestra en el listado 4p.4.

LISTADO 4P.4: Configuración de IceStorm
`cpp-icestorm/icestorm.config`

```
IceStormAdmin.TopicManager.Default=IceStorm/TopicManager:tcp -p 10000
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 2000
IceStorm.Flush.Timeout=2000
Freeze.DbEnv.IceStorm.DbHome=db
```

Las **líneas 1–3** especifican los endpoints para los adaptadores del *TopicManager*, el objeto de administración, y los publicadores de los canales. La **línea 5** indica el nombre del directorio donde se almacenará la configuración del servicio (que es persistente de forma automática).

Una vez configurado se puede lanzar el servicio y probar el ejemplo. Lo primero es arrancar IceBox (en *background*):

```
$ icebox --Ice.Config=icebox.config &
```

A continuación se puede usar la herramienta de administración de IceStorm en línea de comando para crear el canal, asumiendo que no existe:

```
$ icestormadmin --Ice.Config=icestorm.config -e "create PrinterTopic"
```

La configuración del suscriptor es:

LISTADO 4P.5: Configuración del suscriptor
[cpp-icestorm/subscriber.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
PrinterAdapter.Endpoints=tcp
```

Y con eso ya se puede arrancar el suscriptor:

```
$ ./subscriber.py --Ice.Config=subscriber.config
```

La configuración para el publicador es:

LISTADO 4P.6: Configuración de publicador
[cpp-icestorm/publisher.config](#)

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
```

Y por último en una consola diferente se ejecuta el publicador:

```
$ ./publisher.py --Ice.Config=publisher.config
```

Hecho lo cual aparecerán los eventos al llegar al suscriptor:

```
0: Event received: Hello World!
1: Event received: Hello World!
2: Event received: Hello World!
3: Event received: Hello World!
4: Event received: Hello World!
5: Event received: Hello World!
6: Event received: Hello World!
7: Event received: Hello World!
8: Event received: Hello World!
9: Event received: Hello World!
```

Es de destacar el hecho de que los eventos estén modelados como invocaciones. De ese modo es posible eliminar temporalmente el canal de eventos e invocar directamente a un solo suscriptor durante las fases iniciales de modelado del protocolo y desarrollo de prototipos.

4p.2. Federación de canales

La federación de canales de eventos representa un mecanismo que, apropiadamente usado, puede aportar grandes ventajas en cuanto a escalabilidad y tolerancia a fallos a un determinado entorno. Generalmente los servicios de eventos asociados a las plataformas distribuidas contemplan este mecanismo para asociar canales de eventos y permitir escalar los entornos de forma que un evento generado en un canal de eventos en particular es transmitido a sus canales federados.

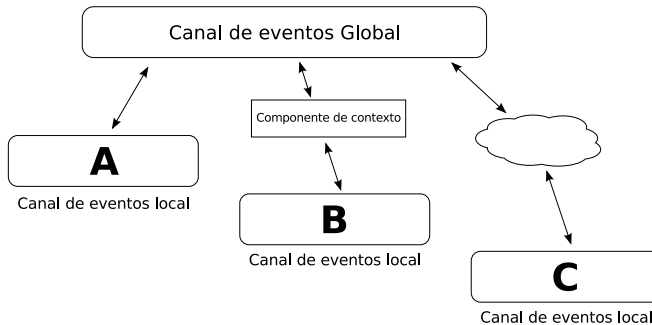


FIGURA 4P.1: Esquema de federación de canales

Se contemplan tres formas de federación posibles:

Federación directa

La federación directa es la más básica y generalmente está soportada por el servicio de eventos del middleware. Este tipo de federación debe ser usada entre canales de eventos del mismo tipo (es decir, mensajes con la misma estructura) y con el objetivo general de agrupar eventos por funcionalidad en un entorno y configurar entornos tolerantes a fallos.

En la figura 4p.1, el canal de eventos A se federa al canal de eventos Global por medio del servicio básico. Si el canal de eventos A y el canal de eventos Global están soportados por distintas instancias del servicio de eventos, una caída del servicio de eventos que da soporte al canal A no afecta al canal de eventos Global y viceversa.

Federación indirecta

Se realiza a través de un componente de contexto. Al poner un elemento intermedio entre los dos canales de eventos se permite la implementación de políticas complejas en el paso de eventos de un canal a otro. En la figura 4p.1 el canal de eventos B está conectado a través de un componente de contexto (que es un objeto distribuido) al canal de eventos Global.

Un ejemplo de aplicación de este tipo de federación podría ser la gestión de los sensores de presencia en un entorno laboral (ej: fábricas, oficinas, plantas, etc.). En horario diurno cuando los trabajadores se encuentran realizando su jornada laboral se puede indicar a los componentes de contexto que no retransmitan los eventos de presencia ya que son continuos y están asociados al ir y venir de operarios por la instalación. En horario nocturno esta política

cambia y se indica a dichos componentes que retransmitan dichos eventos ya que los mismos eventos de presencia son considerados como eventos de intrusión. Es necesario resaltar que los propios servicios de eventos permiten establecer políticas básicas por lo que este escenario está contemplado para la implementación de políticas relativamente complejas difíciles de abordar con el mecanismo de políticas básicas. Este escenario permite federar canales de eventos de distinto tipo y filtrar la información que debe ser manejada por instancias superiores.

Federación remota

Se considera una federación remota (canal de eventos C en la figura 4p.1) cuando los eventos retransmitidos deben atravesar secciones de red no pertenecientes a la organización (ej: a través de Internet) o entornos poco seguros (ej: la vía pública). Esta federación se puede realizar a su vez de forma directa o de forma indirecta (ej: a través de un componente de contexto localizado en el entorno al que pertenece el canal de eventos C). Es necesario resaltar que en estos escenarios se debe establecer una conexión segura (SSL, VPN, Autenticación, etc.) para mantener la confidencialidad de las comunicaciones.

Los mecanismos de federación de canales de eventos deben ser utilizados para segmentar la funcionalidad del entorno atendiendo a razones de despliegue físicas (entornos autónomos y estancos) o de funcionalidad lógica (jerarquizando el paso de mensajes).

4p.2.1. Propagación entre canales de eventos federados

La política de propagación de mensajes entre canales de eventos federados es un aspecto crítico para que una arquitectura de eventos sea eficiente y no sobrecargue la red.

Uno de los aspectos mas importantes a la hora de propagar mensajes entre canales de eventos federados es evitar establecer bucles ya sea de forma directa o indirecta que originen mensajes que pasan de un canal a otro de forma recurrente.

Para evitar este tipo de bucles se define el coste de un mensaje. El coste asociado a un mensaje es un valor entero que nos determinará la política de propagación que debemos seguir con ese mensaje.

Para los distintos tipos de federaciones, definimos la siguiente política:

- Los mensajes nunca son propagados mas allá de un solo enlace.
- Los enlaces deben tener un coste asociado, aquellos canales de eventos que tienen otros canales federados, deben propagar un mensaje a dichos canales si el coste que lleva asociado el mensaje es igual o inferior al coste asociado al enlace.
- Si un enlace establece un coste 0, entonces todos los mensajes serán enviados a través de dicho enlace sin importar el coste del mensaje.

- Si un mensaje tiene un coste 0 será enviado a todos los canales de eventos federados sin importar el coste del mensaje.
- Los enlaces entre canales de eventos con un coste 0 deben ser utilizados para difundir de forma automática todos los mensajes de un canal a otro de forma directa (e.j para modelar escenarios mas extensos).
- La federación directa, tal y como podemos ver en la siguiente figura, es útil para modularizar escenarios y agrupar dispositivos que generan el mismo tipo de mensajes en función de su localización física, agrupación funcional, etc.

Invocación y despachado asíncrono

[C++]

En este capítulo veremos dos técnicas (invocación asíncrona y despachado asíncrono) muy convenientes para reducir el impacto que las comunicaciones pueden suponer para el rendimiento de la aplicación distribuida.

Cuando un cliente invoca un método remoto la ejecución se bloquea hasta que el método ha terminado, de igual forma que ocurre en una invocación a método local convencional. La diferencia es que ésta espera puede durar entre 10 y 100 veces más en una invocación remota que en una local, aparte por supuesto del trabajo real que implique el método. En muchas situaciones el cliente podría realizar alguna otra tarea mientras espera a que la ejecución del método haya concluido. Las técnicas para lograr esto suelen denominarse «programación asíncrona».

Dichas técnicas ayudan a conseguir un sistema más eficiente y escalable, aprovechando mejor los recursos disponibles.

5c.1. Invocación síncrona de métodos

Por defecto, el modelo de envío de peticiones utilizado por ICE está basado en la invocación síncrona de métodos remotos: la invocación de la operación se comporta como una llamada a un método local convencional, es decir, el hilo del cliente se bloquea durante la ejecución del método y se vuelve a activar cuando la llamada se ha completado, y todos los resultados están disponibles. La misma situación se da a pesar de que el método no devuelva ningún resultado (como las invocaciones locales).

Éste es el caso de los programas de los capítulos anteriores. Es fácil comprobar que funciona de este modo sin más que añadir una pausa de unos pocos segundos en la implementación del método en el sirviente (tal como proponía el ejercicio **E2c.11**). La ejecución del cliente dura poco más que la pausa que pongamos en el sirviente.

5c.2. Invocación asíncrona de métodos

Hay muchas situaciones en las que el cliente puede realizar otras tareas mientras espera a que el objeto remoto realice el trabajo y devuelva los resultados (valor de retorno). O simplemente porque no existen o no necesita dichos resultados. Esto es especialmente relevante en programas con interfaz gráfica. En estos casos se puede conseguir una importante mejora de rendimiento utilizando invocación asíncrona.

ICE proporciona varios mecanismos para implementar la invocación asíncrona:

- Proxies asíncronos.
- Objetos *callback*.
- Comprobación activa (polling).

Los dos primeros son los más interesantes y son los que veremos a continuación.

Una cuestión que resulta particularmente interesante es el hecho de que el servidor no está involucrado en forma alguna en los mecanismos AMI (Asynchronous Method Invocation), atañe únicamente a la parte cliente. De hecho, desde el servidor no es posible saber si el cliente está haciendo una invocación asíncrona o síncrona.

5c.2.1. Proxies asíncronos

El cliente invoca un método especial con el prefijo `begin_` antes del nombre del método especificado en el fichero SLICE. Inmediatamente obtiene un manejador (`Ice::AsyncResult`) para la invocación, que puede no haber comenzado siquiera. El cliente puede realizar otras operaciones y después, utilizar el manejador (por medio de otro método especial con el prefijo `end_`) para conseguir los valores de retorno. Aunque ese método remoto concreto no devuelva resultados puede ser interesante realizar dicha acción si se necesita comprobar que la operación ha terminado correctamente en el servidor.

En este caso, utilizamos un fichero SLICE (ver listado 5c.1) que describe la operación `factorial()`, es decir, en este caso si hay un valor de retorno:

LISTADO 5C.1: Especificación SLICE para cálculo del factorial
([cpp-ami/factorial.ice](#))

```
module Example {
  interface Math {
    long factorial(int value);
  };
};
```

El listado 5c.2 muestra el cliente completo utilizando un proxy asíncrono.

LISTADO 5C.2: Cliente AMI con proxy asíncrono
([Client-end.cpp](#))


```

#include <Ice/Ice.h>
#include "factorial.h"

using namespace std;
using namespace Ice;

class Client: public Ice::Application {
    int run(int argc, char* argv[]) {
        ObjectPrx proxy = communicator()->stringToProxy(argv[1]);
        Example::MathPrx math = Example::MathPrx::checkedCast(proxy);

        Ice::AsyncResultPtr async_result = math->begin_factorial(atoi(argv[2]));
        cout << "that was an async call" << endl;

        cout << math->end_factorial(async_result) << endl;
        return 0;
    }
};

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "usage: " << argv[0] << "<server> <value>" << endl;
        return 1;
    }

    Client app;
    return app.main(argc, argv);
}

```

La parte interesante de este listado comienza en la **línea 12**, en la que el cliente invoca el método `begin_factorial()` del proxy `math` pasando el segundo argumento de línea de comandos, convenientemente convertido a entero con `atoi()`. Se obtiene un objeto de tipo *AsyncResultPtr* llamado `async_result`.

Después se imprime un mensaje (**línea 13**) que demuestra que el cliente puede realizar otras operaciones mientras la invocación se está realizando. Por último, se invoca el método `end_factorial()` del mismo proxy (**línea 15**), pasando el objeto `async_result` y se obtiene el resultado (del cálculo del factorial) como valor de retorno.

Si en el momento de ejecutar este método la invocación no se hubiera completado aún, el cliente quedaría bloqueado en espera de su finalización.

El diagrama de secuencia de la figura 5c.1 muestra todo el proceso.

Tal como se ha indicado anteriormente, el servidor se implementa de la forma habitual. Se muestra en el listado 5c.3.

LISTADO 5C.3: Servidor para `Math::factorial()`
(`cpp-ami/Server.cpp`)

```

#include <Ice/Ice.h>
#include "factorialI.h"

using namespace std;
using namespace Ice;

class Server: public Ice::Application {
    int run(int argc, char* argv[]) {
        Example::MathPtr servant = new Example::MathI();

        ObjectAdapterPtr adapter = \
            communicator()->createObjectAdapter("MathAdapter");
    }
};

```

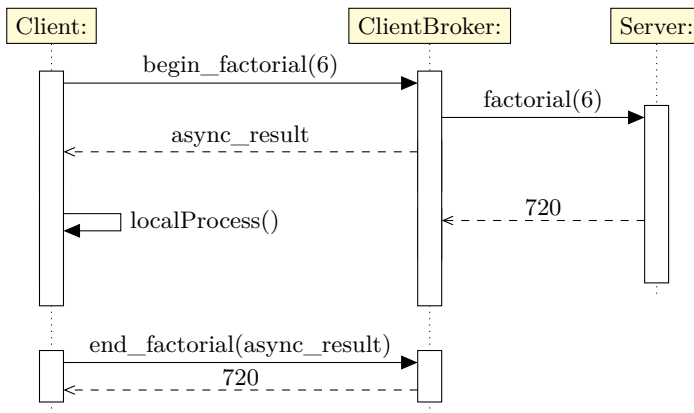


FIGURA 5C.1: Invocación asíncrona (proxy asíncrono)

```

ObjectPrx math = adapter->add(
    servant, communicator()->stringToIdentity("math1"));

cout << communicator()->proxyToString(math) << endl;

adapter->activate();
shutdownOnInterrupt();
communicator()->waitForShutdown();
return 0;
}
};

int main(int argc, char* argv[]) {
    Server app;
    return app.main(argc, argv);
}

```

5c.2.2. Utilizando un objeto *callback*

En este caso, para realizar la invocación remota, el cliente utiliza un método del proxy con el prefijo *begin* que acepta un parámetro adicional. Se trata de un objeto de retrollamada (*callback object*). Esta invocación retorna inmediatamente y el cliente puede seguir ejecutando otras operaciones.

Cuando la invocación remota se ha completado, el núcleo de ejecución de la parte del cliente invoca el método indicado en el objeto *callback* pasado inicialmente, proporcionando los resultados de la invocación. En caso de error, invoca otro método en el que proporciona información sobre la excepción asociada.

En el listado 5c.4 se muestra el código completo para el cliente que crea y proporciona un *callback* según la técnica indicada.

LISTADO 5C.4: Cliente AMI que recibe la respuesta mediante un *callback*
([cpp-ami/Client-callback.cpp](#))

```

#include <Ice/Ice.h>
#include <factorial.h>

using namespace std;

```

```

using namespace Ice;
using namespace Example;

class FactorialCB : public IceUtil::Shared {
public:
    void response(const Ice::Long retval) {
        cout << "Callback: Value is " << retval << endl;
    }

    void failure(const Exception& ex) {
        cout << "Exception is: " << ex << endl;
    }
};

class Client: public Ice::Application {
public:
    int run(int argc, char* argv[]) {

        ObjectPrx proxy = communicator()->stringToProxy(argv[1]);
        MathPrx math = MathPrx::checkedCast(proxy);

        Callback_Math_factorialPtr factorial_cb =
            newCallback_Math_factorial(new FactorialCB,
                                       &FactorialCB::response,
                                       &FactorialCB::failure);

        math->begin_factorial(atoi(argv[2]), factorial_cb);
        return 0;
    }
};

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "usage: " << argv[0] << "<server> <value>" << endl;
        return 1;
    }

    Client app;
    return app.main(argc, argv);
}

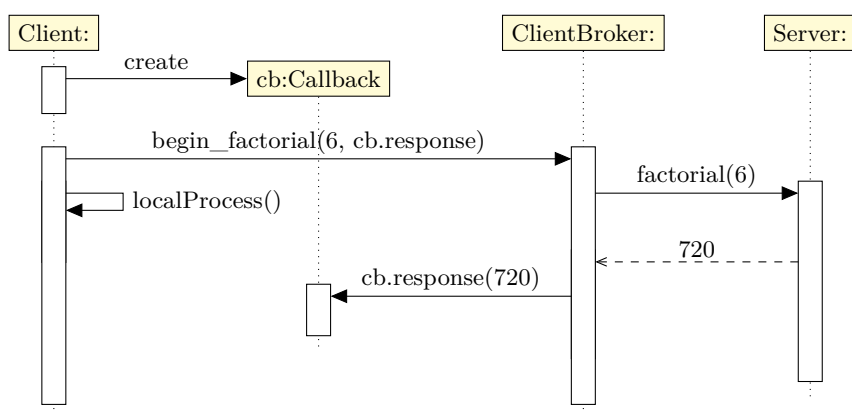
```

Las **líneas 8–17** corresponden con la definición de la clase *callback* `FactorialCB`. El método `response()` será invocado por el núcleo de ejecución cuando el método remoto haya terminado y el resultado esté de vuelta. También se define un método `failure()` que será invocado con una excepción en el caso de producirse.

En la **línea 26** se crea la instancia del *callback*, mediante una función específica para este método: `newCallback_Math_factorial()` pasándole una instancia de nuestra clase `FactorialCB` y sendos punteros a los métodos definidos en ella. Esa función y todos los tipos nombrados siguiendo el patrón ‘`callback`’_ {interface}_ {método} son generados automáticamente por el compilador de interfaces en el espacio de nombre *Example*.

Por último en la **línea 31** aparece la invocación al método remoto pasando la instancia del *callback* (`factorial_cb`) como segundo parámetro.

El diagrama de secuencia de la figura 5c.2 representa las interacciones entre los distintos componentes.

FIGURA 5C.2: Invocación asíncrona (objeto *callback*)

5c.3. Despachado asíncrono de métodos

El tratamiento de métodos asíncrono AMD (Asynchronous Method Dispatching) es la contrapartida de AMI en el lado del servidor. En el tratamiento síncrono por defecto, el núcleo de ejecución de la parte servidora ejecuta la operación en el momento en el que recibe. Si esa operación implica utilizar algún recurso adicional, el hilo quedará bloqueado en espera de dicho recurso o de que se completen las operaciones derivadas correspondientes. Eso implica que el hilo de ejecución asociado a la invocación en el servidor sólo se libera cuando la operación se ha completado.

Con AMD se informa al sirviente de la llegada de la invocación. Pero, en lugar de forzar al proceso a atender la petición inmediatamente, el servidor puede almacenar en una cola la especificación de la tarea asociada a esa solicitud y atender la tarea en otro momento. De ese modo se puede liberar el hilo asociado a la invocación.

El servidor puede utilizar otro hilo (que puede ser siempre el mismo si se desea) para extraer las tareas de la cola y procesarla en la forma y tiempo que considere más adecuada en función de los recursos disponibles. Estos hilos, que extraen y procesan tareas de la cola, se les suele denominar *workers* y de hecho puede haber varios sin mayor problema. Cuando los resultados de la tarea están disponibles, el servidor puede recuperar un objeto especial proporcionado por el API del middleware para enviar el mensaje de respuesta al cliente.

AMD es útil, por ejemplo, si un servidor ofrece operaciones que bloquean a los clientes por un periodo largo de tiempo. Por ejemplo, el servidor puede tener un objeto con una operación `get()` que devuelve los datos de una fuente de datos externa y asíncrona, lo cual bloquearía al cliente hasta que los datos estuvieran disponibles.

Con el tratamiento síncrono, cada cliente que espere unos determinados datos estaría vinculado a un hilo de ejecución del servidor. Claramente, este enfoque no es escalable con una docena de clientes. Con AMD, cientos o incluso miles de

clientes podrían bloquearse en la misma invocación, pero desacoplados del modelo de concurrencia que utilice el servidor.

El tratamiento síncrono y asíncrono de métodos es transparente al cliente, es decir, el cliente es incapaz de determinar si un cierto servidor realiza un tratamiento síncrono o asíncrono de las invocaciones que recibe.

Para gestionar las tareas pendientes se utiliza una cola en la que se introducen estructuras de datos que incluyen los argumentos de entrada y la instancia que permite recuperar el manejador de la invocación para enviar la respuesta. Esta cola debe ser *thread-safe* dado que las tareas se introducen desde el hilo del sirviente y se extraen desde un *worker* (un hilo diferente).

Para utilizar AMD es necesario añadir metadatos a la especificación SLICE. El listado 5c.5 muestra el fichero de interfaces que utilizamos en este caso.

LISTADO 5C.5: Especificación SLICE para cálculo del factorial con AMD
([cpp-amd/factorial.ice](#))

```
module Example {
    interface Math {
        ["amd"] long factorial(int value);
    };

    exception RequestCanceledException {};
};
```

En la **línea 3** se puede apreciar la etiqueta ["amd"] que le indica al compilador de interfaces que debe generar soporte en el servidor para la gestión de la respuesta asíncrona tal como se ha descrito. El listado 5c.6 muestra el código del servidor AMD.

LISTADO 5C.6: Servidor AMD
([cpp-amd/Server.cpp](#))

```
#include <Ice/Ice.h>
#include "factorialI.h"
#include "WorkQueue.h"

using namespace std;
using namespace Ice;

class AMDServer : public Application {
private:
    WorkQueuePtr _workQueue;

public:
    virtual int run(int, char*[]) {
        callbackOnInterrupt();

        _workQueue = new WorkQueue();
        Example::MathPtr servant = new Example::MathI(_workQueue);

        ObjectAdapterPtr adapter = \
            communicator()->createObjectAdapter("MathAdapter");
        ObjectPrx math = adapter->add(
            servant, communicator()->stringToIdentity("math1"));

        cout << communicator()->proxyToString(math) << endl;

        _workQueue->start();
        adapter->activate();
        communicator()->waitForShutdown();
    }
};
```

```

        _workQueue->getThreadControl().join();

    return EXIT_SUCCESS;
}

virtual void interruptCallback(int) {
    _workQueue->destroy();
    communicator()->shutdown();
}
};

int main(int argc, char* argv[]) {
    AMDServer app;
    return app.main(argc, argv);
}

```

Las diferencias respecto a un servidor convencional están en la creación de la cola de tareas `_workQueue` (línea 16), que se pasa en el constructor del sirviente (línea 17), el arranque de la cola (línea 26) y el método `interruptCallback()`, que se encarga de la limpieza: destrucción de la cola y el comunicador. Este método es invocado automáticamente por la aplicación cuando el proceso reciba la *señal de interrupción* ('SIGINT' en un sistema POSIX) debido a que anteriormente lo fijamos como «manejador» con `callbackOnInterrupt()` (línea 14).

No se incluye aquí el código de la implementación de la cola de tareas. En todo caso puedes ver su implementación en el repositorio de ejemplos, concretamente en la ruta `cpp-amd/WorkerQueue.cpp`.

En este caso el sirviente es extremadamente simple, puesto que lo único que hace la implementación del método (línea 9) es añadir el callback de la invocación (que se recibe como parámetro en `cb`) y el parámetro (`value`) a la cola de tareas (línea 11). Nótese que en este caso, el método se llama `factorial_async()`, para distinguirlo de la versión síncrona, que no tiene el sufijo `_async`.

LISTADO 5c.7: Sirviente para AMD
(`cpp-amd/factorialI.cpp`)

```

#include "factorialI.h"
#include "WorkQueue.h"

using namespace Example;

MathI::MathI(const WorkQueuePtr& workQueue) :
    _workQueue(workQueue) { }

void MathI::factorial_async(const Example::AMD_Math_factorialPtr& cb,
                           int value, const Ice::Current&) {
    _workQueue->add(cb, value);
}

```

Como en los casos anteriores, el proceso se resume en el diagrama de secuencia 5c.3.

5c.4. Mecanismos desacoplados

Es importante recalcar que los mecanismos AMI y AMD están completamente desacoplados y que la decisión de utilizarlos involucra únicamente al programador

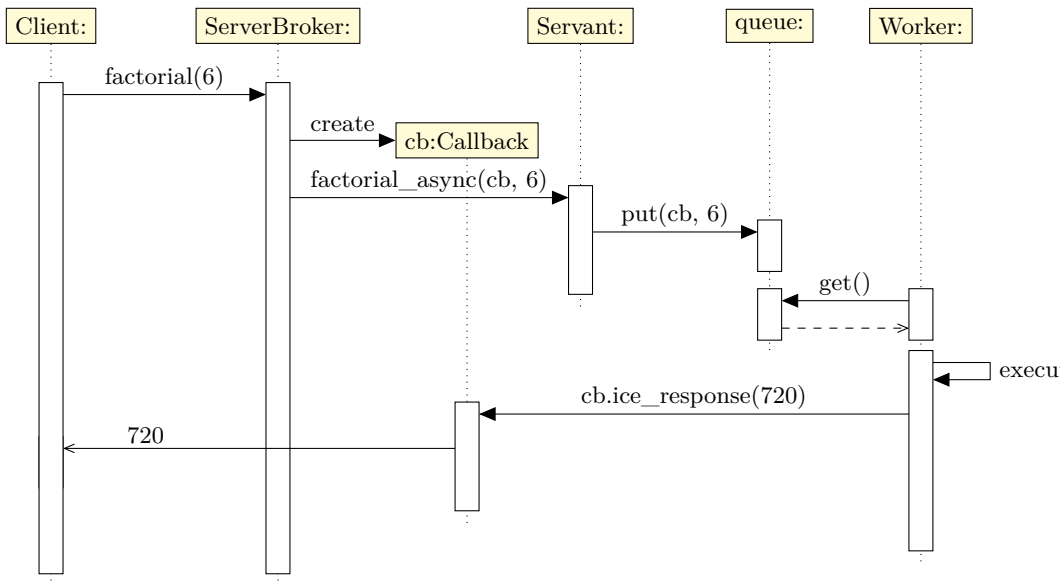


FIGURA 5C.3: Despachado asíncrono

de cada parte. Es perfectamente posible implementar una o incluso varias técnicas AMI a la vez en el mismo cliente, hacer invocaciones AMI a otros objetos desde un sirviente¹ o en un mismo servidor, utilizar AMD para el despachado y AMI para la invocación a otros objetos.

5c.5. Ejercicios

- E 5c.01** Escribe un cliente AMI y un servidor para demostrar que efectivamente el cliente puede hacer otras operaciones mientras espera a que el servidor termine la invocación.
- E 5c.02** Escribe un cliente AMI que invoca 10 objetos de forma asíncrona y después recoge los resultados. La invocación al siguiente objeto se realizará antes de recibir los resultados de la invocación anterior.
- E 5c.03** Con la configuración por defecto un cliente AMI no puede realizar varias invocaciones asíncronas a un mismo objeto. ¿Por qué? ¿Qué cambio se debe hacer para lograrlo?
- E 5c.04** Escribe un nuevo servidor de *Example.Math* que redirige las invocaciones al servidor original (de forma similar al ejercicio **E 2c.13**). Este nuevo servidor debe hacer despachado asíncrono (AMD) para atender a los clientes e invocación asíncrona (AMI) para invocar al servidor original.

¹Esta situación requiere configuración específica en el servidor puesto que la ejecución del sirviente se realiza en un hilo, y la invocación AMI necesita un hilo adicional.

- E 5c.05** La `workqueue` implementada en el ejemplo de AMD tiene un único *worker*. Escribe una `workQueue` con una cantidad configurable de *workers* atendiendo la misma cola.

Invocación y despacho asíncrono

[Java]

En este capítulo veremos dos técnicas (invocación asíncrona y despacho asíncrono) muy convenientes para reducir el impacto que las comunicaciones pueden suponer para el rendimiento de la aplicación distribuida.

Cuando un cliente invoca un método remoto la ejecución se bloquea hasta que el método ha terminado, de igual forma que ocurre en una invocación a método local convencional. La diferencia es que ésta espera puede durar entre 10 y 100 veces más en una invocación remota que en una local, aparte por supuesto del trabajo real que implique el método. En muchas situaciones el cliente podría realizar alguna otra tarea mientras espera a que la ejecución del método haya concluido. Las técnicas para lograr esto suelen denominarse «programación asíncrona».

Dichas técnicas ayudan a conseguir un sistema más eficiente y escalable, aprovechando mejor los recursos disponibles.

5j.1. Invocación síncrona de métodos

Por defecto, el modelo de envío de peticiones utilizado por ICE está basado en la invocación síncrona de métodos remotos: la invocación de la operación se comporta como una llamada a un método local convencional, es decir, el hilo del cliente se bloquea durante la ejecución del método y se vuelve a activar cuando la llamada se ha completado, y todos los resultados están disponibles. La misma situación se da a pesar de que el método no devuelva ningún resultado (como las invocaciones locales).

Éste es el caso de los programas de los capítulos anteriores. Es fácil comprobar que funciona de este modo sin más que añadir una pausa de unos pocos segundos en la implementación del método en el sirviente (tal como proponía el ejercicio **E 2j.11**). La ejecución del cliente dura poco más que la pausa que pongamos en el sirviente.

5j.2. Invocación asíncrona de métodos

Hay muchas situaciones en las que el cliente puede realizar otras tareas mientras espera a que el objeto remoto realice el trabajo y devuelva los resultados (valor de retorno). O simplemente porque no existen o no necesita dichos resultados. Esto es especialmente relevante en programas con interfaz gráfica. En estos casos se puede conseguir una importante mejora de rendimiento utilizando invocación asíncrona.

ICE proporciona varios mecanismos para implementar la invocación asíncrona:

- Proxies asíncronos.
- Objetos *callback*.
- Comprobación activa (polling).

Los dos primeros son los más interesantes y son los que veremos a continuación.

Una cuestión que resulta particularmente interesante es el hecho de que el servidor no está involucrado en forma alguna en los mecanismos AMI, atañe únicamente a la parte cliente. De hecho, desde el servidor no es posible saber si el cliente está haciendo una invocación asíncrona o síncrona.

5j.2.1. Proxies asíncronos

El cliente invoca un método especial con el prefijo `begin_` antes del nombre del método especificado en el fichero SLICE. Inmediatamente obtiene un manejador (*Ice::AsyncResult*) para la invocación, que puede no haber comenzado siquiera. El cliente puede realizar otras operaciones y después, utilizar el manejador (por medio de otro método especial con el prefijo `end_`) para conseguir los valores de retorno. Aunque ese método remoto concreto no devuelva resultados puede ser interesante realizar dicha acción si se necesita comprobar que la operación ha terminado correctamente en el servidor.

En este caso, utilizamos un fichero SLICE (ver listado 5j.1) que describe la operación `factorial()`, es decir, en este caso si hay un valor de retorno:

LISTADO 5J.1: Especificación SLICE para cálculo del factorial
([java-ami/factorial.ice](#))

```
module Example {  
  interface Math {  
    long factorial(int value);  
  };  
};
```

El listado 5j.2 muestra el cliente completo utilizando un proxy asíncrono.

LISTADO 5J.2: Cliente AMI con proxy asíncrono
([java-ami/Client_end.java](#))

```
public class Client_end extends Ice.Application {
```

```

public int run(String[] args) {
    Ice.ObjectPrx proxy = communicator().stringToProxy(args[0]);
    Example.MathPrx math = Example.MathPrxHelper.checkedCast(proxy);

    int value = Integer.parseInt(args[1]);
    Ice.AsyncResult async_result = math.begin_factorial(value);
    System.out.println("that was an async call");

    System.out.println(math.end_factorial(async_result));
    return 0;
}

static public void main(String[] args) {
    if (args.length != 2) {
        System.err.println(appName() + ": usage: <server> <value>");
        return;
    }

    Client_end app = new Client_end();
    app.main("Client", args);
}
}

```

La parte interesante de este listado comienza en la **línea 7**, en la que el cliente invoca el método `begin_factorial()` del proxy `math` pasando el segundo argumento de línea de comandos, convenientemente convertido a entero con `Integer.parseInt()`. Se obtiene un objeto de tipo *AsyncResultPtr* llamado `async_result`.

Después se imprime un mensaje (**línea 8**) que demuestra que el cliente puede realizar otras operaciones mientras la invocación se está realizando. Por último, se invoca el método `end_factorial()` del mismo proxy (**línea 10**), pasando el objeto `async_result` y se obtiene el resultado (del cálculo del factorial) como valor de retorno.

Si en el momento de ejecutar este método la invocación no se hubiera completado aún, el cliente quedaría bloqueado en espera de su finalización.

El diagrama de secuencia de la figura 5j.1 muestra todo el proceso.

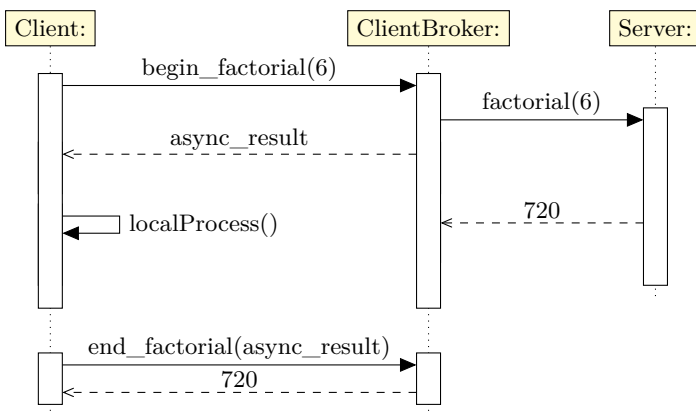


FIGURA 5J.1: Invocación asincrónica (proxy asíncrono)

Tal como se ha indicado anteriormente, el servidor se implementa de la forma habitual. Se muestra en el listado 5j.3.

LISTADO 5J.3: Servidor para `Math.factorial()`
([java-ami/Server.java](#))

```
import Ice.*;

public class Server extends Ice.Application {
    public int run(String[] args) {
        ObjectAdapter adapter = communicator().createObjectAdapter("MathAdapter");
        ObjectPrx math = adapter.add(new MathI(), Util.stringToIdentity("math1"));

        System.out.println(communicator().proxyToString(math));

        adapter.activate();
        shutdownOnInterrupt();
        communicator().waitForShutdown();
        return 0;
    }

    static public void main(String[] args) {
        Server app = new Server();
        app.main("Server", args);
    }
}
```

5j.2.2. Utilizando un objeto *callback*

En este caso, para realizar la invocación remota, el cliente utiliza un método del proxy con el prefijo `begin` que acepta un parámetro adicional. Se trata de un objeto de retrollamada (*callback object*). Esta invocación retorna inmediatamente y el cliente puede seguir ejecutando otras operaciones.

Cuando la invocación remota se ha completado, el núcleo de ejecución de la parte del cliente invoca el método indicado en el objeto *callback* pasado inicialmente, proporcionando los resultados de la invocación. En caso de error, invoca otro método en el que proporciona información sobre la excepción asociada.

En el listado 5j.4 se muestra el código completo para el cliente que crea y proporciona un *callback* según la técnica indicada.

LISTADO 5J.4: Cliente AMI que recibe la respuesta mediante un *callback*
([java-ami/Client_callback.java](#))

```
class FactorialCB extends Example.Callback_Math_factorial {
    public void response(long value) {
        System.out.println("Callback: Value is: " + value);
    }

    public void exception(Ice.LocalException ex) {
        System.err.println("Exception is: " + ex);
    }
}

public class Client_callback extends Ice.Application {
    public int run(String[] args) {
        Ice.ObjectPrx proxy = communicator().stringToProxy(args[0]);
        Example.MathPrx math = Example.MathPrxHelper.checkedCast(proxy);

        FactorialCB factorial_cb = new FactorialCB();
        math.begin_factorial(Integer.parseInt(args[1]), factorial_cb);
        return 0;
    }

    static public void main(String[] args) {
        if (args.length != 2) {

```

```

    System.err.println(appName() + ": usage: <server> <value>");
    return;
}

Client_callback app = new Client_callback();
app.main("Client", args);
}
}

```

Las **líneas 1–9** corresponden con la definición de la clase *callback* **FactorialCB**. El método `response()` será invocado por el núcleo de ejecución cuando el método remoto haya terminado y el resultado esté de vuelta. También se define un método `failure()` que será invocado con una excepción en el caso de producirse.

En la **línea 17** se crea la instancia del *callback* y en la **línea 17** se invoca el método remoto pasando la instancia del *callback* (`factorial_cb`) como segundo parámetro.

El diagrama de secuencia de la figura 5j.2 representa las interacciones entre los distintos componentes.

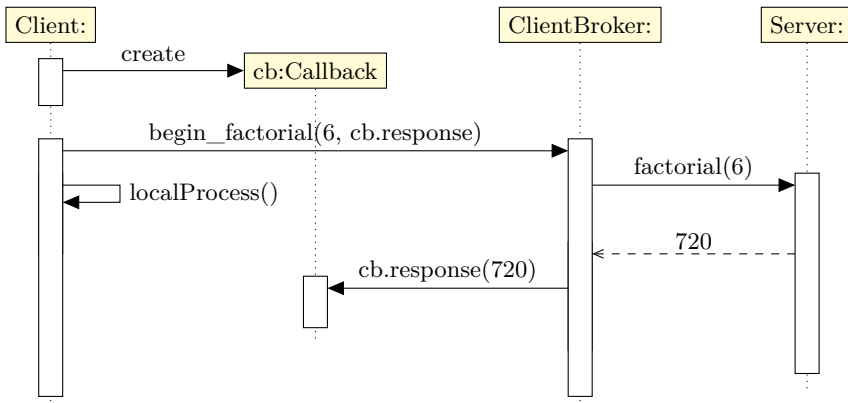


FIGURA 5J.2: Invocación asíncrona (objeto *callback*)

5j.3. Despachado asíncrono de métodos

El tratamiento de métodos asíncrono AMD es la contrapartida de AMI en el lado del servidor. En el tratamiento síncrono por defecto, el núcleo de ejecución de la parte servidora ejecuta la operación en el momento en el que recibe. Si esa operación implica utilizar algún recurso adicional, el hilo quedará bloqueado en espera de dicho recurso o de que se completen las operaciones derivadas correspondientes. Eso implica que el hilo de ejecución asociado a la invocación en el servidor sólo se libera cuando la operación se ha completado.

Con AMD se informa al sirviente de la llegada de la invocación. Pero, en lugar de forzar al proceso a atender la petición inmediatamente, el servidor puede almacenar en una cola la especificación de la tarea asociada a esa solicitud y atender la tarea en otro momento. De ese modo se puede liberar el hilo asociado a la invocación.

El servidor puede utilizar otro hilo (que puede ser siempre el mismo si se desea) para extraer las tareas de la cola y procesarla en la forma y tiempo que considere más adecuada en función de los recursos disponibles. Estos hilos, que extraen y procesan tareas de la cola, se les suele denominar *workers* y de hecho puede haber varios sin mayor problema. Cuando los resultados de la tarea están disponibles, el servidor puede recuperar un objeto especial proporcionado por el API del middleware para enviar el mensaje de respuesta al cliente.

AMD es útil, por ejemplo, si un servidor ofrece operaciones que bloquean a los clientes por un periodo largo de tiempo. Por ejemplo, el servidor puede tener un objeto con una operación `get()` que devuelve los datos de una fuente de datos externa y asíncrona, lo cual bloquearía al cliente hasta que los datos estuvieran disponibles.

Con el tratamiento síncrono, cada cliente que espere unos determinados datos estaría vinculado a un hilo de ejecución del servidor. Claramente, este enfoque no es escalable con una docena de clientes. Con AMD, cientos o incluso miles de clientes podrían bloquearse en la misma invocación, pero desacoplados del modelo de concurrencia que utilice el servidor.

El tratamiento síncrono y asíncrono de métodos es transparente al cliente, es decir, el cliente es incapaz de determinar si un cierto servidor realiza un tratamiento síncrono o asíncrono de las invocaciones que recibe.

Para gestionar las tareas pendientes se utiliza una cola en la que se introducen estructuras de datos que incluyen los argumentos de entrada y la instancia que permite recuperar el manejador de la invocación para enviar la respuesta. Esta cola debe ser *thread-safe* dado que las tareas se introducen desde el hilo del sirviente y se extraen desde un *worker* (un hilo diferente).

Para utilizar AMD es necesario añadir metadatos a la especificación SLICE. El listado 5j.5 muestra el fichero de interfaces que utilizamos en este caso.

LISTADO 5J.5: Especificación SLICE para cálculo del factorial con AMD
([java-amd/factorial.ice](#))

```
module Example {
  interface Math {
    ["amd"] long factorial(int value);
  };

  exception RequestCanceledException {};
};
```

En la **línea 3** se puede apreciar la etiqueta `["amd"]` que le indica al compilador de interfaces que debe generar soporte en el servidor para la gestión de la respuesta asíncrona tal como se ha descrito. El listado 5j.6 muestra el código del servidor AMD.

LISTADO 5J.6: Servidor AMD
([java-amd/Server.java](#))

```
import Ice.*;

public class Server extends Ice.Application {
  class ShutdownHook extends Thread {
```

```

    public void run() {
        _workQueue._destroy();
        communicator().shutdown();
    }
}

public int run(String[] args) {
    setInterruptHook(new ShutdownHook());

    _workQueue = new WorkQueue();
    MathI servant = new MathI(_workQueue);

    ObjectAdapter adapter =
        communicator().createObjectAdapter("MathAdapter");
    ObjectPrx math = adapter.add(
        servant, Util.stringToIdentity("math1"));

    System.out.println(communicator().proxyToString(math));

    _workQueue.start();
    adapter.activate();
    communicator().waitForShutdown();

    try {
        _workQueue.join();
    } catch (java.lang.InterruptedException ex) {}

    return 0;
}

static public void main(String[] args) {
    Server app = new Server();
    app.main("Server", args);
}

private WorkQueue _workQueue;
}

```

Las diferencias respecto a un servidor convencional están en la creación de la cola de tareas `_workQueue` (línea 14), que se pasa en el constructor del sirviente (línea 15), el arranque de la cola (línea 24) y la clase que se encarga de la destrucción de la cola y el comunicador (`ShutdownHook()`). El método `run()` de dicha clase es invocado automáticamente por la aplicación cuando el proceso reciba la *señal de interrupción* (`'SIGINT'` en un sistema POSIX) debido a que anteriormente lo fijamos con la llamada `setInterruptHook()` (línea 12).

No se incluye aquí el código de la implementación de la cola de tareas. En todo caso puedes ver su implementación en el repositorio de ejemplos, concretamente en la ruta `java-amd/WorkerQueue.java`.

En este caso el sirviente es extremadamente simple, puesto que lo único que hace la implementación del método (línea 6) es añadir el callback de la invocación (que se recibe como parámetro en `cb`) y el parámetro (`value`) a la cola de tareas (línea 8). Nótese que en este caso, el método se llama `factorial_async()`, para distinguirlo de la versión síncrona, que no tiene el sufijo `_async`.

LISTADO 5J.7: Sirviente para AMD
([java-amd/MathI.java](#))

```

public final class MathI extends Example._MathDisp {
    public MathI(WorkQueue workQueue) {
        _workQueue = workQueue;
    }
}

```

```

public void factorial_async(Example.AMD_Math_factorial cb,
                           int value, Ice.Current current) {
    _workQueue.add(cb, value);
}

private WorkQueue _workQueue;
}

```

Como en los casos anteriores, el proceso se resume en el diagrama de secuencia 5j.3.

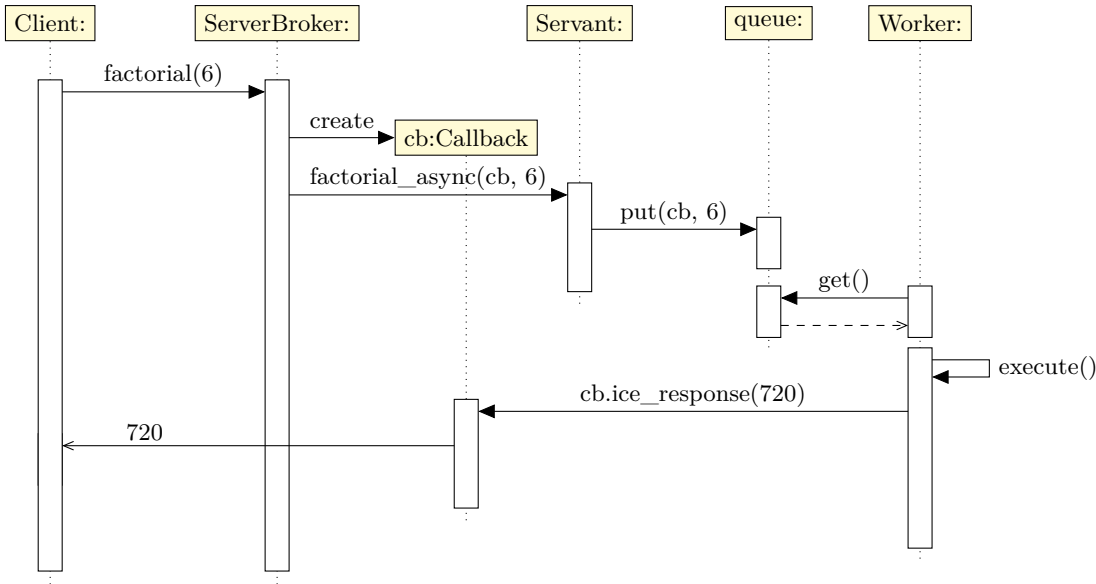


FIGURA 5j.3: Despachado asíncrono

5j.4. Mecanismos desacoplados

Es importante recalcar que los mecanismos AMI y AMD están completamente desacoplados y que la decisión de utilizarlos involucra únicamente al programador de cada parte. Es perfectamente posible implementar una o incluso varias técnicas AMI a la vez en el mismo cliente, hacer invocaciones AMI a otros objetos desde un sirviente¹ o en un mismo servidor, utilizar AMD para el despacho y AMI para la invocación a otros objetos.

5j.5. Ejercicios

E 5j.01 Escribe un cliente AMI y un servidor para demostrar que efectivamente el cliente puede hacer otras operaciones mientras espera a que el servidor termine la invocación.

¹Esta situación requiere configuración específica en el servidor puesto que la ejecución del sirviente se realiza en un hilo, y la invocación AMI necesita un hilo adicional.

- E 5j.02** Escribe un cliente AMI que invoca 10 objetos de forma asíncrona y después recoge los resultados. La invocación al siguiente objeto se realizará antes de recibir los resultados de la invocación anterior.
- E 5j.03** Con la configuración por defecto un cliente AMI no puede realizar varias invocaciones asíncronas a un mismo objeto. ¿Por qué? ¿Qué cambio se debe hacer para lograrlo?
- E 5j.04** Escribe un nuevo servidor de *Example.Math* que redirige las invocaciones al servidor original (de forma similar al ejercicio **E 2j.13**). Este nuevo servidor debe hacer despachado asíncrono (AMD) para atender a los clientes e invocación asíncrona (AMI) para invocar al servidor original.
- E 5j.05** La *workQueue* implementada en el ejemplo de AMD tiene un único *worker*. Escribe una *workQueue* con una cantidad configurable de *workers* atendiendo la misma cola.

Invocación y despachado asíncrono

[Python]

En este capítulo veremos dos técnicas (invocación asíncrona y despachado asíncrono) muy convenientes para reducir el impacto que las comunicaciones pueden suponer para el rendimiento de la aplicación distribuida.

Cuando un cliente invoca un método remoto la ejecución se bloquea hasta que el método ha terminado, de igual forma que ocurre en una invocación a método local convencional. La diferencia es que ésta espera puede durar entre 10 y 100 veces más en una invocación remota que en una local, aparte por supuesto del trabajo real que implique el método. En muchas situaciones el cliente podría realizar alguna otra tarea mientras espera a que la ejecución del método haya concluido. Las técnicas para lograr esto suelen denominarse «programación asíncrona».

Dichas técnicas ayudan a conseguir un sistema más eficiente y escalable, aprovechando mejor los recursos disponibles.

5p.1. Invocación síncrona de métodos

Por defecto, el modelo de envío de peticiones utilizado por ICE está basado en la invocación síncrona de métodos remotos: la invocación de la operación se comporta como una llamada a un método local convencional, es decir, el hilo del cliente se bloquea durante la ejecución del método y se vuelve a activar cuando la llamada se ha completado, y todos los resultados están disponibles. La misma situación se da a pesar de que el método no devuelva ningún resultado (como las invocaciones locales).

Éste es el caso de los programas de los capítulos anteriores. Es fácil comprobar que funciona de este modo sin más que añadir una pausa de unos pocos segundos en la implementación del método en el sirviente (tal como proponía el ejercicio **E2p.11**). La ejecución del cliente dura poco más que la pausa que pongamos en el sirviente.

5p.2. Invocación asíncrona de métodos

Hay muchas situaciones en las que el cliente puede realizar otras tareas mientras espera a que el objeto remoto realice el trabajo y devuelva los resultados (valor de retorno). O simplemente porque no existen o no necesita dichos resultados. Esto es especialmente relevante en programas con interfaz gráfica. En estos casos se puede conseguir una importante mejora de rendimiento utilizando invocación asíncrona.

ICE proporciona varios mecanismos para implementar la invocación asíncrona:

- Proxies asíncronos.
- Objetos *callback*.
- Comprobación activa (polling).

Los dos primeros son los más interesantes y son los que veremos a continuación.

Una cuestión que resulta particularmente interesante es el hecho de que el servidor no está involucrado en forma alguna en los mecanismos AMI, atañe únicamente a la parte cliente. De hecho, desde el servidor no es posible saber si el cliente está haciendo una invocación asíncrona o síncrona.

5p.2.1. Proxies asíncronos

El cliente invoca un método especial con el prefijo `begin_` antes del nombre del método especificado en el fichero SLICE. Inmediatamente obtiene un manejador (*Ice::AsyncResult*) para la invocación, que puede no haber comenzado siquiera. El cliente puede realizar otras operaciones y después, utilizar el manejador (por medio de otro método especial con el prefijo `end_`) para conseguir los valores de retorno. Aunque ese método remoto concreto no devuelva resultados puede ser interesante realizar dicha acción si se necesita comprobar que la operación ha terminado correctamente en el servidor.

En este caso, utilizamos un fichero SLICE (ver listado 5p.1) que describe la operación `factorial()`, es decir, en este caso si hay un valor de retorno:

LISTADO 5P.1: Especificación SLICE para cálculo del factorial
([py-ami/factorial.ice](#))

```
module Example {
  interface Math {
    long factorial(int value);
  };
};
```

El listado 5p.2 muestra el cliente completo utilizando un proxy asíncrono.

LISTADO 5P.2: Cliente AMI con proxy asíncrono
([py-ami/Client-end.py](#))

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

```
# -*- coding: utf-8 -*-
"usage: {} <server> <value>"

import sys
import Ice
Ice.loadSlice('./factorial.ice')
import Example

class Client(Ice.Application):
    def run(self, argv):
        proxy = self.communicator().stringToProxy(argv[1])
        math = Example.MathPrx.checkedCast(proxy)

        if not math:
            raise RuntimeError("Invalid proxy")

        async_result = math.begin_factorial(int(argv[2]))
        print 'that was an async call'

        print math.end_factorial(async_result)

        return 0

if len(sys.argv) != 3:
    print __doc__.format(__file__)
    sys.exit(1)

app = Client()
sys.exit(app.main(sys.argv))
```

La parte interesante de este listado comienza en la **línea 19**, en la que el cliente invoca el método `begin_factorial()` del proxy `math` pasando el segundo argumento de línea de comandos, convenientemente convertido a entero con `int()`. Se obtiene un objeto de tipo *AsyncResultPtr* llamado `async_result`.

Después se imprime un mensaje (**línea 20**) que demuestra que el cliente puede realizar otras operaciones mientras la invocación se está realizando. Por último, se invoca el método `end_factorial()` del mismo proxy (**línea 22**), pasando el objeto `async_result` y se obtiene el resultado (del cálculo del factorial) como valor de retorno.

Si en el momento de ejecutar este método la invocación no se hubiera completado aún, el cliente quedaría bloqueado en espera de su finalización.

El diagrama de secuencia de la figura 5p.1 muestra todo el proceso.

Tal como se ha indicado anteriormente, el servidor se implementa de la forma habitual. Se muestra en el listado 5p.3.

LISTADO 5P.3: Servidor para `Math.factorial()`
(`py-ami/Server.py`)

```
#!/usr/bin/python -u
# -*- coding: utf-8 -*-

import sys
import Ice

Ice.loadSlice('./factorial.ice')
import Example

def factorial(n):
```

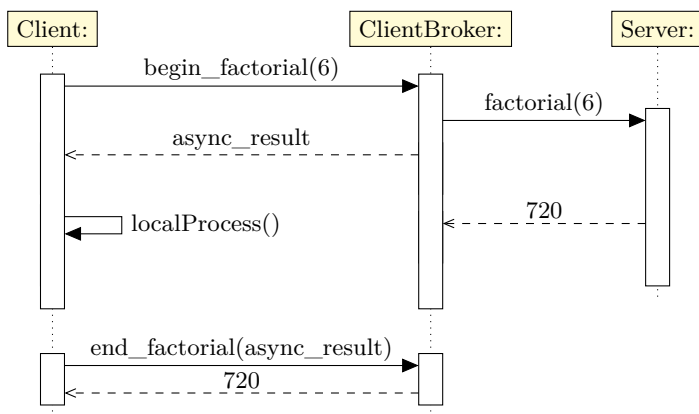


FIGURA 5P.1: Invocación asíncrona (proxy asíncrono)

```

if n == 0:
    return 1

return n * factorial(n - 1)

class MathI(Example.Math):
    def factorial(self, n, current=None):
        return factorial(n)

class Server(Ice.Application):
    def run(self, argv):
        broker = self.communicator()

        adapter = broker.createObjectAdapter("MathAdapter")
        math = adapter.add(MathI(), broker.stringToIdentity("math1"))

        print math

        adapter.activate()
        self.shutdownOnInterrupt()
        broker.waitForShutdown()

    return 0

sys.exit(Server().main(sys.argv))

```

5p.2.2. Utilizando un objeto *callback*

En este caso, para realizar la invocación remota, el cliente utiliza un método del proxy con el prefijo `begin` que acepta un parámetro adicional. Se trata de un objeto de retollamada (*callback object*). Esta invocación retorna inmediatamente y el cliente puede seguir ejecutando otras operaciones.

Cuando la invocación remota se ha completado, el núcleo de ejecución de la parte del cliente invoca el método indicado en el objeto *callback* pasado inicialmente, proporcionando los resultados de la invocación. En caso de error, invoca otro método en el que proporciona información sobre la excepción asociada.

En el listado 5p.4 se muestra el código completo para el cliente que crea y proporciona un *callback* según la técnica indicada.

LISTADO 5P.4: Cliente AMI que recibe la respuesta mediante un *callback* (*py-ami/Client_callback.py*)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys

import Ice
Ice.loadSlice('./factorial.ice')
import Example

class FactorialCB(object):
    def response(self, retval):
        print("Callback: Value is {}".format(retval))

    def failure(self, ex):
        print("Exception is: {}".format(ex))

class Client(Ice.Application):
    def run(self, argv):
        proxy = self.communicator().stringToProxy(argv[1])
        math = Example.MathPrx.checkedCast(proxy)

        if not math:
            raise RuntimeError("Invalid proxy")

        factorial_cb = FactorialCB()

        math.begin_factorial(int(argv[2]),
                             factorial_cb.response, factorial_cb.failure)
        print 'that was an async call'

        return 0

if len(sys.argv) != 3:
    print __doc__.format(__file__)
    sys.exit(1)

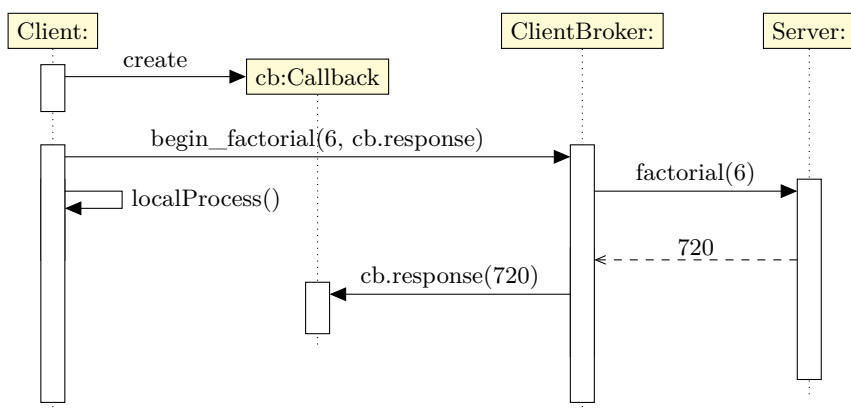
app = Client()
sys.exit(app.main(sys.argv))
```

Las líneas 11–16 corresponden con la definición de la clase *callback* `FactorialCB`. El método `response()` será invocado por el núcleo de ejecución cuando el método remoto haya terminado y el resultado esté de vuelta. También se define un método `failure()` que será invocado con una excepción en el caso de producirse.

El diagrama de secuencia de la figura 5p.2 representa las interacciones entre los distintos componentes.

5p.3. Despachado asíncrono de métodos

El tratamiento de métodos asíncrono AMD es la contrapartida de AMI en el lado del servidor. En el tratamiento síncrono por defecto, el núcleo de ejecución de la parte servidora ejecuta la operación en el momento en el que recibe. Si esa operación implica utilizar algún recurso adicional, el hilo quedará bloqueado en espera de dicho recurso o de que se completen las operaciones derivadas correspondientes. Eso implica que el hilo de ejecución asociado a la invocación en el servidor sólo se libera cuando la operación se ha completado.

FIGURA 5P.2: Invocación asíncrona (objeto *callback*)

Con AMD se informa al sirviente de la llegada de la invocación. Pero, en lugar de forzar al proceso a atender la petición inmediatamente, el servidor puede almacenar en una cola la especificación de la tarea asociada a esa solicitud y atender la tarea en otro momento. De ese modo se puede liberar el hilo asociado a la invocación.

El servidor puede utilizar otro hilo (que puede ser siempre el mismo si se desea) para extraer las tareas de la cola y procesarla en la forma y tiempo que considere más adecuada en función de los recursos disponibles. Estos hilos, que extraen y procesan tareas de la cola, se les suele denominar *workers* y de hecho puede haber varios sin mayor problema. Cuando los resultados de la tarea están disponibles, el servidor puede recuperar un objeto especial proporcionado por el API del middleware para enviar el mensaje de respuesta al cliente.

AMD es útil, por ejemplo, si un servidor ofrece operaciones que bloquean a los clientes por un periodo largo de tiempo. Por ejemplo, el servidor puede tener un objeto con una operación `get()` que devuelve los datos de una fuente de datos externa y asíncrona, lo cual bloquearía al cliente hasta que los datos estuvieran disponibles.

Con el tratamiento síncrono, cada cliente que espere unos determinados datos estaría vinculado a un hilo de ejecución del servidor. Claramente, este enfoque no es escalable con una docena de clientes. Con AMD, cientos o incluso miles de clientes podrían bloquearse en la misma invocación, pero desacoplados del modelo de concurrencia que utilice el servidor.

El tratamiento síncrono y asíncrono de métodos es transparente al cliente, es decir, el cliente es incapaz de determinar si un cierto servidor realiza un tratamiento síncrono o asíncrono de las invocaciones que recibe.

Para gestionar las tareas pendientes se utiliza una cola en la que se introducen estructuras de datos que incluyen los argumentos de entrada y la instancia que permite recuperar el manejador de la invocación para enviar la respuesta. Esta cola debe ser *thread-safe* dado que las tareas se introducen desde el hilo del sirviente y se extraen desde un *worker* (un hilo diferente).

Para utilizar AMD es necesario añadir metadatos a la especificación SLICE. El listado 5p.5 muestra el fichero de interfaces que utilizamos en este caso.

LISTADO 5P.5: Especificación SLICE para cálculo del factorial con AMD
([py-amd/factorial.ice](#))

```
module Example {
  interface Math {
    ["amd"] long factorial(int value);
  };

  exception RequestCanceledException {};
};
```

En la **línea 3** se puede apreciar la etiqueta `["amd"]` que le indica al compilador de interfaces que debe generar soporte en el servidor para la gestión de la respuesta asíncrona tal como se ha descrito. El listado 5p.6 muestra el código del servidor AMD.

LISTADO 5P.6: Servidor AMD
([py-amd/Server.py](#))

```
#!/usr/bin/python -u
# -*- mode:python; coding:utf-8; tab-width:4 -*-

import sys

import Ice
Ice.loadSlice('factorial.ice')
import Example

from work_queue import WorkQueue

class MathI(Example.Math):
    def __init__(self, work_queue):
        self.work_queue = work_queue

    def factorial_async(self, cb, value, current=None):
        self.work_queue.add(cb, value)

class Server(Ice.Application):
    def run(self, argv):
        work_queue = WorkQueue()
        servant = MathI(work_queue)

        broker = self.communicator()

        adapter = broker.createObjectAdapter("MathAdapter")
        print adapter.add(servant, broker.stringToIdentity("math1"))
        adapter.activate()

        work_queue.start()

        self.shutdownOnInterrupt()
        broker.waitForShutdown()

        work_queue.destroy()
        return 0

sys.exit(Server().main(sys.argv))
```

Las diferencias respecto a un servidor convencional están en la creación de la cola de tareas queue (**línea 23**), que se pasa en el constructor del sirviente (**línea 24**), el

arranque de la cola (**línea 32**) y la destrucción de la cola al terminar el programa (**línea 37**).

No se incluye aquí el código de la implementación de la cola de tareas. En todo caso puedes ver su implementación en el repositorio de ejemplos, concretamente en la ruta `py-amd/work_queue.py`.

En este caso el sirviente es extremadamente simple, puesto que lo único que hace la implementación del método (**línea 17**) es añadir el callback de la invocación (que se recibe como parámetro en `cb`) y el parámetro (`value`) a la cola de tareas (**línea 18**). Nótese que en este caso, el método se llama `factorial_async()`, para distinguirlo de la versión síncrona, que no tiene el sufijo `_async`.

Como en los casos anteriores, el proceso se resume en el diagrama de secuencia 5p.3.

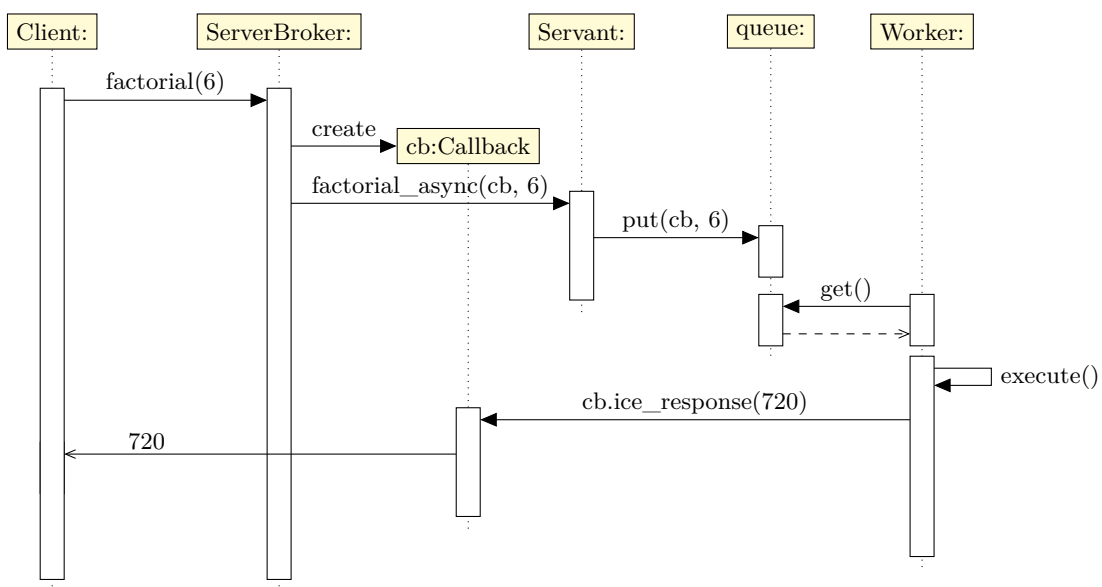


FIGURA 5P.3: Despachado asíncrono

5p.4. Mecanismos desacoplados

Es importante recalcar que los mecanismos AMI y AMD están completamente desacoplados y que la decisión de utilizarlos involucra únicamente al programador de cada parte. Es perfectamente posible implementar una o incluso varias técnicas AMI a la vez en el mismo cliente, hacer invocaciones AMI a otros objetos desde un sirviente¹ o en un mismo servidor, utilizar AMD para el despacho y AMI para la invocación a otros objetos.

¹Esta situación requiere configuración específica en el servidor puesto que la ejecución del sirviente se realiza en un hilo, y la invocación AMI necesita un hilo adicional.

5p.5. Ejercicios

- E 5p.01** Escribe un cliente AMI y un servidor para demostrar que efectivamente el cliente puede hacer otras operaciones mientras espera a que el servidor termine la invocación.
- E 5p.02** Escribe un cliente AMI que invoca 10 objetos de forma asíncrona y después recoge los resultados. La invocación al siguiente objeto se realizará antes de recibir los resultados de la invocación anterior.
- E 5p.03** Con la configuración por defecto un cliente AMI no puede realizar varias invocaciones asíncronas a un mismo objeto. ¿Por qué? ¿Qué cambio se debe hacer para lograrlo?
- E 5p.04** Escribe un nuevo servidor de *Example.Math* que redirige las invocaciones al servidor original (de forma similar al ejercicio **E 2p.13**). Este nuevo servidor debe hacer despachado asíncrono (AMD) para atender a los clientes e invocación asíncrona (AMI) para invocar al servidor original.
- E 5p.05** La *workQueue* implementada en el ejemplo de AMD tiene un único *worker*. Escribe una *workQueue* con una cantidad configurable de *workers* atendiendo la misma cola.

En cualquier middleware orientado a objetos como ICE, los objetos obviamente son los protagonistas. En este capítulo verá algunas soluciones típicas para pueden cubrir algunas necesidades básicas en la gestión y manipulación de objetos. Los denominados *servicios comunes* (los que ofrece el fabricante del middleware) cubren precisamente necesidades de este tipo.

En este capítulo no verá como funcionan esos *servicios comunes*. En su lugar, verá cómo implementar versiones minimalistas alternativas, que si bien pueden ser insuficientes para una aplicación en producción, le servirán para entender cuál es la finalidad y funcionamiento de sus equivalente más sofisticados. Algunos de estos servicios se asemejan (o directamente corresponden) con los patrones de diseño clásicos [GHJV94], mientras que en otros son simplemente soluciones de conveniencia práctica.

6p.1. Contenedor de objetos

Un contenedor (o directorio) de objetos es conceptualmente un servicio tremendamente simple. Se trata de un objeto distribuido que almacena referencias a otros objetos cualesquiera. Una interfaz sencilla para este servicio podría ser la siguiente:

LISTADO 6P.1: Especificación SLICE para el contenedor de objetos

[py-container/container.ice](#)

```
module Services {
    exception AlreadyExists { string key; };
    exception NoSuchKey { string key; };

    dictionary<string, Object*> ObjectPrxDict;

    interface Container {
        void link(string key, Object* proxy) throws AlreadyExists;
        void unlink(string key) throws NoSuchKey;
        ObjectPrxDict list();
    };
};
```

La interfaz *Container* tiene tres métodos muy simples:

link Añade al directorio una referencia a un objeto arbitrario mediante una clave.

unlink

Elimina una referencia a un objeto enlazado en el contenedor dada su clave.

list Devuelve un diccionario (una tabla asociativa) de clave-objeto.

Es fácil ver que la implementación del servicio en sí es en esencia una tabla asociativa. Veamos la implementación del sirviente en el listado 6p.2.

LISTADO 6P.2: Sirviente del servicio contenedor de objetos

[py-container/server.py](#)

```

class ContainerI(Services.Container):
    def __init__(self):
        self.proxies = dict()

    def link(self, key, proxy, current=None):
        if key in self.proxies:
            raise Services.AlreadyExists(key)

        print("link: {0} -> {1}".format(key, proxy))
        self.proxies[key] = proxy

    def unlink(self, key, current=None):
        if not key in self.proxies:
            raise Services.NoSuchKey(key)

        print("unlink: {0}".format(key))
        del self.proxies[key]

    def list(self, current=None):
        return self.proxies

```

Veamos algunos detalles destacables. Como se puede apreciar, el sirviente efectivamente almacena las referencias en un diccionario (**línea 3**). En la implementación del método `link()` se comprueba si la clave ya existe (**línea 6**), caso en el que se eleva la excepción remota `AlreadyExists`; esto significa que no es posible sobre-escribir componentes del contenedor con la misma clave. La contraparte de esta comprobación y su excepción correspondiente se tratan en el método `unlink()` para verificar que el objeto que se desea eliminar existe realmente en el contenedor (**línea 13**). Por último, el método `list()` simplemente devuelve una copia del atributo diccionario, que será serializado por ICE al tipo correspondiente declarado en el fichero `Slice: ObjectPrxDict`.

Como el tipo de los proxies en el contenedor es `Object*`, este contenedor puede almacenar **cualquier tipo de objeto Ice** y será el cliente del servicio (como suele ser habitual) el encargado de realizar el molde (*cast*) al tipo concreto que espera. Este servicio es muy útil cuando una aplicación debe manejar una cantidad grande o arbitraria de objetos. También resulta interesante el hecho de que un contenedor como éste puede enlazar otros contenedores, de modo que se podría crear fácilmente una jerarquía de objetos con un nivel de anidación arbitrario.

Es importante destacar que el contenedor únicamente contiene referencias, es decir, los objetos deben existir en algún otro servidor dentro del grid, y no les afecta en absoluto que sus referencias sean añadidas o eliminadas de éste u otros contenedores.

Como el lector habrá deducido, este tipo de servicio ya existe en ICE. Efectivamente gracias a IceGrid y en particular al servicio *Locator* es posible asociar

nombres a objetos y obtener fácilmente sus referencias; es lo que conocemos como «objetos bien conocidos». Sin embargo, este tipo de servicio «contenedor» es más adecuado para objetos más efímeros y irrelevantes para la aplicación, y de hecho la clave que se utilice para referenciarlos dentro del contenedor no tiene porqué tener ningún importancia significativa en el contexto de la aplicación distribuida. Si se obvian las claves, el contenedor puede verse como un *conjunto* de objetos sin un orden específico.

La implementación que aquí se propone tiene algunas limitaciones e inconvenientes relevantes.

Volátil

Las referencias que almacena el contenedor no son persistentes. Si el servidor que lo aloja se reinicia o falla se perderá toda la información. Esta limitación puede resolverse fácilmente haciendo uso del servicio Freeze de ICE u otro sistema de persistencia similar. Recuerde que el objetivo principal de los ejemplos de este capítulo es la simplicidad y no la completitud.

E 6p.01 Modifique el sirviente `ContainerI` para añadir soporte de persistencia de modo que, al reiniciarse el servidor, el contenedor tenga los mismos proxies.

Escalabilidad limitada

El servicio no escala satisfactoriamente. Si el contenedor almacena referencias a cientos o miles de objetos, el método `list()` tratará de devolver toda la información en un solo mensaje, y eso puede no ser posible y mucho menos eficiente incluso con relativamente pocos objetos. Este problema normalmente se solventa creando «iteradores» o bien «paginadores».

El *iterador* es un objeto especial con estado ¹; permite obtener un subconjunto acotado del contenido y después haciendo uso de un método del propio iterador, obtener bloques sucesivos hasta agotar los datos (en este caso los proxies) alojados en el contenedor. Un iterador remoto de este tipo plantea a su vez otros problemas puesto que es complejo para el servidor qué iteradores están en uso. Es sencillo imaginar un ataque DoS (Denial of Service) contra un servicio que ofrece iteradores. Un ejemplo de este tipo de servicio podría ser el *CosNaming::NamingContext* de CORBA.

El *paginador* persigue el mismo objetivo, pero no tiene estado. A partir de un *tamaño de página* determinado, el cliente puede solicitar en cualquier momento la cantidad de *páginas* que tiene el contenedor y pedir los datos del contenedor que corresponden una página arbitraria. Esta solución conlleva definir una política de obsolescencia ya que los datos añadidos o eliminados del contenedor deben reflejarse en el resultado de la paginación, es decir, las solicitudes de los clientes deben manejar posibles inconsistencias.

¹otro patrón de diseño: el *iterator*

Sin validaciones

No se realiza ningún tipo de validación sobre los objetos referenciados. Los proxies suministrados al container podrían corresponder a objetos inexistentes o inalcanzables. Esta es una cuestión que puede ser objeto de debate. Por ejemplo, que los objetos no sean accesibles para el contenedor no implica necesariamente que no lo sean para el cliente que los añadió o para aquel que los vaya a obtener. En general, es razonable que este tipo de comprobaciones las realicen los clientes y no el servicio.

6p.2. Factoría de objetos

En los ejemplos que han aparecido hasta ahora en este documento, los únicos objetos distribuidos que se han manejado corresponden a sirvientes creados por el programador del servicio, típicamente en el programa principal. Es decir, una vez arrancado un servidor, no se crea ni se destruye ningún objeto distribuido.

Sin embargo, en muchas aplicaciones puede surgir la necesidad o la conveniencia de crear nuevos objetos distribuidos cuando la aplicación ya está en marcha. Eso no es ningún inconveniente para el propio servicio. Es posible crear nuevos sirvientes y registrarlos a un adaptador en cualquier momento, incluso desde el cuerpo de otro sirviente.

Pero vayamos un poco más lejos: ¿Podría un cliente crear un nuevo objeto distribuido? En principio no, puesto que, por definición, eso le convertiría en servidor. Bien, redefinamos la pregunta: ¿Podría un cliente crear un nuevo objeto en un servidor remoto? Obviamente sí. Es el mismo caso del párrafo anterior, pero a iniciativa del cliente.

Una factoría de objetos (ya sea concreta o abstracta ²) es un objeto que crea otros objetos. En nuestro caso (objetos distribuidos) esto permite al cliente indicar los argumentos del constructor del sirviente, algo que obviamente no puede hacer directamente. El listado 6p.3 muestra la interfaz Slice (llamada *PrinterFactory*) para una factoría concreta de objetos *Factory*.

LISTADO 6P.3: Interfaz Slice para la factoría de objetos *PrinterFactory*
 py-factory/PrinterFactory.ice

```
#include <Printer.ice>

module Example {
  interface PrinterFactory {
    Printer* make(string name);
  };
};
```

El argumento `name` del método `make()` será el nombre del servicio de impresión a crear y como se puede ver, el valor de retorno es de tipo `Printer*`, es decir, *proxy* a *Printer*.

²Ver patrón de diseño [GHJV94]

Veamos una implementación sencilla de esta interfaz. El listado 6p.4 muestra sirvientes para las interfaces *Printer* (que ya vimos en el capítulo 2p) y la nueva interfaz *PrinterFactory*.

LISTADO 6P.4: Sirvientes de la factoría de objetos
[py-factory/Server.py](#)

```
class PrinterI(Example.Printer):
    def __init__(self, name):
        self.name = name

    def write(self, message, current=None):
        print("{0}: {1}".format(self.name, message))
        sys.stdout.flush()

class PrinterFactoryI(Example.PrinterFactory):
    def make(self, name, current=None):
        servant = PrinterI(name)
        proxy = current.adapter.addWithUUID(servant)
        return Example.PrinterPrx.checkedCast(proxy)
```

E 6p.02 Modifique el método `make()` del listado 6p.4 de modo que el parámetro `name` se utilice como identidad para el nuevo objeto *Printer*, en lugar de utilizar un UUID.

El servidor para el servicio factoría no tiene nada especial (ver listado 6p.5), simplemente crea una instancia de *PrinterFactoryI* e imprime el proxy, de modo muy similar al servidor de *hola mundo* (ver § 2p.2).

LISTADO 6P.5: Servidor para la factoría de objetos
[py-factory/Server.py](#)

```
class Server(Ice.Application):
    def run(self, argv):
        broker = self.communicator()
        servant = PrinterFactoryI()

        adapter = broker.createObjectAdapter("PrinterFactoryAdapter")
        proxy = adapter.add(servant,
                           broker.stringToIdentity("printerFactory1"))

        print(proxy)
        sys.stdout.flush()

        adapter.activate()
        self.shutdownOnInterrupt()
        broker.waitForShutdown()

    return 0

server = Server()
sys.exit(server.main(sys.argv))
```

Por último, el cliente que utiliza la factoría se muestra en el listado 6p.6.

LISTADO 6P.6: Cliente de la factoría de objetos
[py-factory/Client.py](#)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
import Ice
Ice.loadSlice('-I. --all PrinterFactory.ice')
```

```

import Example

class Client(ice.Application):
    def run(self, argv):
        proxy = self.communicator().stringToProxy(argv[1])
        factory = Example.PrinterFactoryPrx.checkedCast(proxy)

        if not factory:
            raise RuntimeError('Invalid proxy')

        printer = factory.make("printer1")
        printer.write('Hello World!')

    return 0

sys.exit(Client().main(sys.argv))

```

Destrucción de objetos

Éste es una cuestión clásica de los sistemas distribuidos. Cuando se le da a los clientes la posibilidad de crear objetos en otra localización aparece un problema no trivial de gestión de recursos. El servidor que realmente aloja el objeto (e instancia posiblemente un sirviente) debería, de un modo u otro, hacerse responsable del ciclo de vida del objeto.

Podríamos pensar en dar al cliente la posibilidad de destruir el objeto cuando no lo necesita, pero el servidor no tiene ninguna garantía de que efectivamente ocurrirá. Incluso un cliente correcto y bienintencionado podría fallar repentinamente o el computador en el que se ejecuta, quedando el objeto «huérfano». Además, el cliente responsable de la creación del objeto podría pasar referencias de ese objeto a otros elementos del sistema de modo que no es sencillo asegurar que realmente no hay ningún potencial usuario del objeto. Este problema se conoce como «recolección de basura distribuida»

- E 6p.03** Realice las modificaciones necesarias, tanto en la declaración Slice como en la implementación del servidor, que permitan al cliente destruir objetos bajo demanda. Aparte de los antes mencionados ¿qué otros problemas acarrea ofrecer esta funcionalidad?
- E 6p.04** Realice las modificaciones necesarias para que el servidor destruya el objeto (e invalide la referencia del cliente) si deja de recibir invocaciones pasada cierta cantidad de tiempo. ¿Qué problemas puede acarrear esta funcionalidad?

Referencias

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, edición 1, Noviembre 1994. url: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201633612>.
- [HS11] M. Henning y M. Spruiell. *Distributed Programming with Ice*. ZeroC Inc., 2011. Revision 3.4.2.