Computación Distribuida con ZeroC Ice

Universidad de Castilla-La Mancha

Escuela Superior de Informática
de Ciudad Real

David Villa
Francisco Moya
Óscar Aceña

2025/12/12

Escuela Superior de Informática

e-mail	esi@uclm.es
Teléfono	926 29 53 00
Web	http://www.esi.uclm.es

Índice general

Listado de acrónimos

AMI	Asynchronous Method Invocation
AMD	Asynchronous Method Dispatching
API	Application Program Interface
ARCO	Arquitectura y Redes de Computadores
ASM	Active Servant Map
CORBA	Common Object Request Broker Architecture
DDNS	Dynamic DNS
DDS	Data Distribution Service
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DoS	Denial of Service
DTO	Data Transfer Object
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
PHP	Personal Home Page
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

Este documento es una introducción muy práctica al desarrollo de aplicaciones distribuidas mediante middlewares orientados a objeto. En concreto se utiliza el middleware Internet Communications Engine (Ice) 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 Common Object Request Broker Architecture (CORBA), 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.

A pesar de estar centrado en una tecnología concreta como es ZeroC Ice, prácticamente la totalidad de conceptos, servicios y mecanismos abordados son perfectamente extrapolables a otras tecnologías presentes en el desarrollo de aplicaciones en red y en los sistemas distribuidos pasados, presentes y probablemente futuros.

Tanto la documentación como el código ha sido desarrollado o adaptado por los miembros del grupo ARCO[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 importantes proyectos de investigación y desarrollo tanto públicos como privados a lo largo de casi dos décadas.

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://github.com/uclm-esi/hello.ice`

Aunque es posible descargar estos ficheros individualmente o como un archivo comprimido, se aconseja utilizar el sistema de control de versiones git[2].

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

[1] `http://arco.esi.uclm.es`
[2] `https://git-scm.com/`

https://github.com/uclm-esi/hello.ice/issues

Es importante que el lector disponga en su computador de una copia actualizada de este repositorio ya que es necesario para seguir correctamente muchas de las explicaciones y ejemplos. En los listados de código y de ejecución de comandos se incluyen rutas que corresponden a los directorios de este repositorio. Por ejemplo, al mostrar la ejecución del siguiente comando, se debe entender que se realiza desde el subdirectorio `./cpp` del repositorio de ejemplos:

```
py$ ./server.py
```

Sobre este documento

Los fuentes de este documento (en LaTeX) también se encuentran en un repositorio mercurial `https://github.com/uclm-esi/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.

https://github.com/uclm-esi/ice-book/issues

Capítulo 1

Introducción

Un middleware de comunicaciones es un sofisticado sistema de Inter-Process Communication (IPC) 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 sintaxis y semántica para los mensajes, o dicho de otro modo, permiten crear protocolos nuevos a la medida de cada aplicación.

Se puede entender un middleware como un software de conectividad que hace posible que aplicaciones distribuidas pueden 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: Remote Procedure Call (RPC), CORBA, Enterprise JavaBeans (EJB), Java Remote Method Invocation (RMI), Data Distribution Service (DDS), .Net Remoting, Thrift, 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 (excepto los relacionados con tecnología web), 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

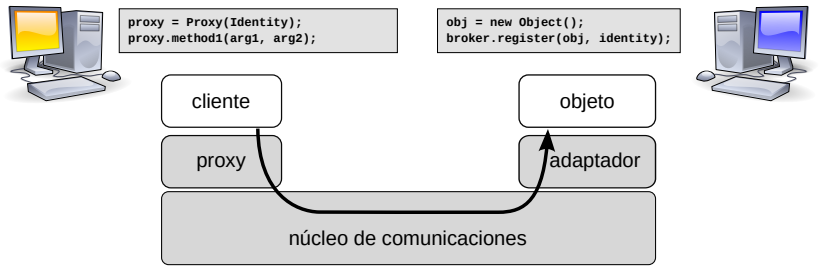


Figura 1.1: Invocación a método remoto

claro que una invocación remota puede resultar hasta 100 veces más lenta que una invocación local convencional.

Llamamos **nodo** a cualquier elemento de cómputo de propósito general que tiene la capacidad de ejecutar código de la aplicación distribuida. El nodo típico es un computador convencional, pero puede ser muchas otras cosas tales como un servidor *enracable*, una Raspberry Pi o cualquier otro computador empotrado, o incluso una máquina virtual o un contenedor Docker

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 remotas. 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. El *proxy* es un objeto en el código del cliente que ofrece exactamente la misma interfaz que el objeto remoto que únicamente actúa como intermediario.

El servidor, por otro lado, utiliza una clase base (una por cada interfaz especificada) encargada de traducir los mensajes entrantes desde 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.

objeto distribuido

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

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 retrans-

misiones, conectividad, asigna puertos, gestiona el ciclo de vida de las conexiones, identifica los objetos, proporciona soporte para invocación y despachado 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.

1.1. ZeroC Ice

Ice (Internet Communications Engine) es un middleware de comunicaciones orientado a objetos desarrollado por la empresa ZeroC Inc[1]. 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, Personal Home Page (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 las decisiones arquitecturales 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 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] http://www.zeroc.com

1.2. Especificación de interfaces

Cuando nos planteamos una interacción con un objeto remoto, lo primero es definir el «contrato» o interfaz, 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 protocolo (estándar o específico) y programar las rutinas de 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 programador únicamente tiene que rellenar ese esqueleto con la implementación concreta.

El lenguaje de especificación de interfaces de Ice se llama Specification Language for Ice (Slice) y proporciona compiladores de interfaces (*compiladores*) 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 Interface Definition Language (IDL) o eXternal Data Representation (XDR)—, es puramente declarativo. Es decir, no permite especificar lógica ni funcionalidad, únicamente interfaces.

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 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#, .NET, Python, y, para el lado del cliente, PHP y Ruby, JavaScript y MatLab.

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

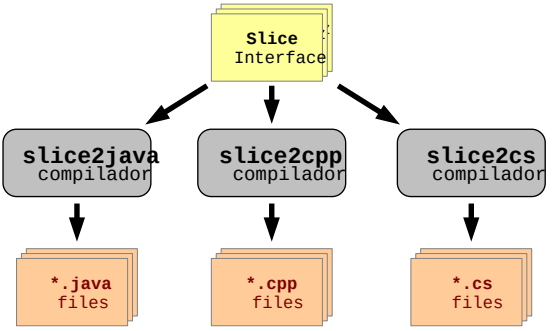


Figura 1.2: Ice proporciona compiladores para cada uno de los lenguajes soportados

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 tiene 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 diferente 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. Además, actúa como el representante local de un objeto (posiblemente remoto), de forma que cuando un cliente invoca una operación en el proxy, el núcleo de comunicaciones:

1. Localiza el 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 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.

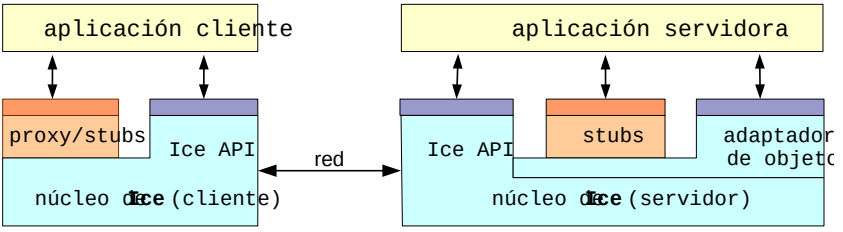


Figura 1.3: Componentes básicos del middleware

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.

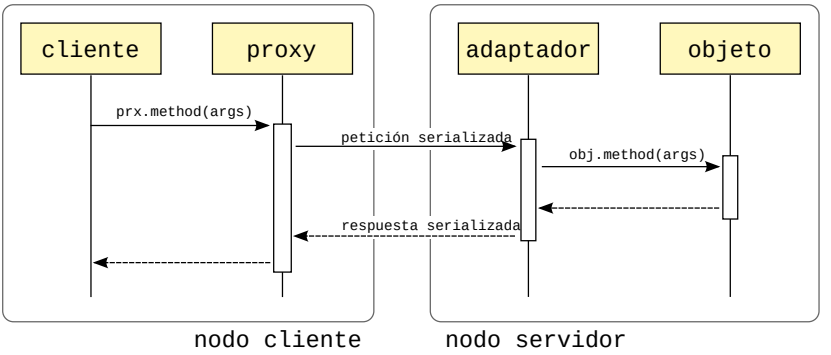


Figura 1.4: Diagrama de secuencia de una invocación a un objeto remoto

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. Modo 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. 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 anunciamiento o búsqueda de servicios. Al no existir mensajes de respuesta se consigue un importante ahorro de ancho de banda.

1.5.4. 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

1.6.1. 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.

1.6.2. 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 mediantes 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 de CORBA, Ice proporciona un protocolo de aplicación para la codificación de las invocaciones a 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 cara 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 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 2p

«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
py/printer.ice

```
module Example {
  interface Printer {
    void write(string message);
  };
};
```

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 *compilador* `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  def main(ic):
2      servant = PrinterI()
3      adapter = ic.createObjectAdapter("PrinterAdapter")
4      proxy = adapter.add(servant, ic.stringToIdentity("printer1"))
5
6      print(proxy)
7
8      adapter.activate()
9      ic.waitForShutdown()
10
11 if __name__ == "__main__":
12     try:
13         with Ice.initialize(sys.argv) as communicator:
14             sys.exit(main(communicator))
15     except KeyboardInterrupt:
16         pass
```

En la **línea 2** se crea el sirviente (una instancia de la clase `PrinterI`). En la **línea 9** se crea un adaptador de objetos, que es el componente encargado de multiplexar

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

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 4** 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 8** 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 9**) bloquea el hilo principal hasta que el comunicador sea terminado.

Con esto tenemos el servidor completo. Pero existe un problema, el cliente necesita un modo de encontrar 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.

Existen formas de conseguir una referencia a un objeto como una representación textual[2] del proxy del objeto. El proxy al nuevo objeto lo obtenemos como resultado de invocar el método `add()` del adaptador. A su vez, el cliente puede generar un objeto proxy a partir de esa representación textual que puede utilizar para contactar con el objeto en el servidor.

Este proxy textual es lo que en sistemas distribuidos se conoce como una **referencia interoperable**. En el caso de Ice tiene un aspecto similar a:

```
printer1 -t:tcp -h 192.168.0.12 -p 9090
```

que corresponde con el formato:

```
object-identity options:endpoints
```

A su vez los *endpoints* siguen el formato:

```
protocol -h host -p port
```

En su forma programática, el proxy funciona como una especie «puntero remoto» para acceder al objeto.

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

[2] llamado *stringified proxy* en la terminología de Ice

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 sys
2  import Ice
3  Ice.loadSlice('printer.ice')
4  import Example
5
6  def main(ic):
7     proxy = ic.stringToProxy(sys.argv[1])
8      printer = Example.PrinterPrx.checkedCast(proxy)
9
10     if not printer:
11         raise RuntimeError('Invalid proxy')
12
13    printer.write('Hello World!')
14
15 if __name__ == "__main__":
16     with Ice.initialize(sys.argv) as communicator:
17         main(communicator)
```

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 7**). 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

Una vez conseguido el proxy del tipo correcto (objeto `printer`) se puede invocar el método remoto `write()` (**línea 13**) 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:

```
py$ python3 server.py
```

[3]Consiste 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.

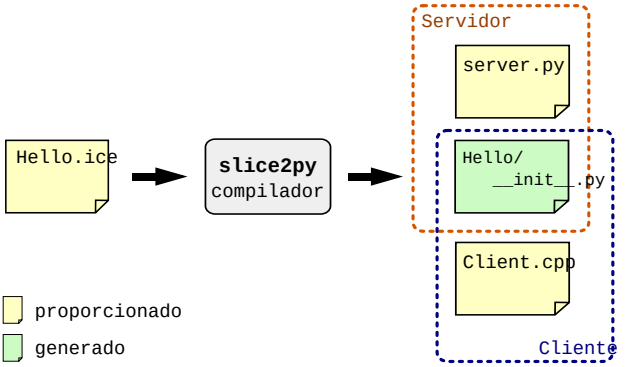


Figura 2p.1: Esquema de compilación de cliente y servidor en Python

```
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)
  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 7070
```

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:

```
py$ python3 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 distribuido[4]. 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.

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:

```
py$ 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.

[4] http://doc.zeroc.com/display/Ice/Proxy+and+Endpoint+Syntax

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 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

Capítulo 3

Transparencia de localización

El direccionamiento es un mecanismo esencial en las redes de computadores y por tanto en cualquier aplicación distribuida Es simple: no se puede usar un servicio que no se puede encontrar.

En el capítulo «Hola mundo distribuido» hemos visto cómo desarrollar y ejecutar una aplicación distribuida básica. El servidor imprime por pantalla la **referencia remota** para el objeto que aloja como un *proxy textual*[1] como el siguiente:

```
printer1 -t:tcp -h 192.168.0.12 -p 9090 -t 60000
```

Esto corresponde con lo que Ice denomina *proxy directo*, es decir, una referencia remota que incluya toda la información necesaria para contactar con el servidor (TCP en este caso).

El cliente la utiliza para crear un objeto proxy (local) con el que invocar al objeto remoto. Sin embargo, desde el punto de vista de la gestión, resulta poco práctico para el cliente tener que conocer los detalles concretos de direccionamiento de los objetos.

Desde el punto de vista de la aplicación distribuida el proxy directo plantea varios problemas:

- Cambiar la dirección IP y el puerto invalida la referencia que se proporcionó al cliente. Las direcciones IP de un nodo pueden cambiar cuando se utiliza asignación dinámica con DHCP, que además es lo más común. El puerto también sería deseable que pudiera cambiar para lograr más flexibilidad, aunque no es tan importante.
- Incluso asumiendo que los nodos mantendrán su dirección IP y los servidores mantendrán su puerto, el proxy directo impediría mover el servidor a otro nodo.
- No sería posible añadir medios adicionales (*endpoints*) para contactar con el servidor.

[1] *stringified proxy*

En una aplicación de medio o gran tamaño se utilizan decenas o centenares de servidores y miles de objetos. Es obvio que hace falta un mecanismo más potente que pueda escalar sin problema.

Como en el caso de los servicios de Internet clásicos, la solución es utilizar *nombres* en lugar de direcciones, y proporcionar un sistema que puede traducir esos *nombres* a *direcciones* eventualmente *en el momento de realizar la invocación*. Esto es a lo que llamamos servicio de *binding*. Internet utiliza nombres de dominio y el servicio DNS. Ice utiliza *proxies indirectos* y el servicio `Locator`.

Un proxy indirecto no tiene información sobre la ubicación del servidor (no aparece ningún *endpoint*). En su lugar aparece un nombre simbólico asociado al adaptador que sirve el objeto. Veamos un ejemplo de *proxy indirecto* en formato textual:

```
printer1 -t -e 1.1 @ PrinterAdapter
```

De esta forma Ice proporciona *transparencia de localización*, es decir, el mecanismo que permite que un programador (o usuario) pueda utilizar un objeto remoto sin necesidad de conocer su situación lógica ni física.

Esto posibilita que los servidores (objetos en nuestro caso) puedan migrar a cualquier otro lugar de Internet sin que los usuarios puedan percibir ningún cambio significativo. A su vez, la transparencia de localización habilita otras funcionalidades muy interesantes como redundancia, tolerancia a fallos o balanceo de carga, muy importantes en aplicación de gran escala o que requieren fiabilidad extra.

3.1. `Registry` y `Locator`

Ice dispone de un servidor denominado `registry` que proporciona múltiples servicios como veremos en próximos capítulos. Entre ellos, está el `Registry`, que permite registrar adaptadores de objetos, y el `Locator`, que permite hacer consultas para averiguar los *endpoints* de un adaptador previamente registrado[2].

Para arrancar una instancia del `registry` necesitamos un fichero de configuración, con una sintaxis equivalente a la que vimos en los servidores básicos, es decir, una lista de propiedades definida en forma «clave=valor».

Para una prueba sencilla podemos utilizar el siguiente fichero de configuración:

Listado 3.1: Configuración de un Registry
locator/registry.config

```
1 Ice.Default.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
2
3 IceGrid.Registry.Client.Endpoints=tcp -p 4061
4 IceGrid.Registry.Server.Endpoints=tcp
5 IceGrid.Registry.Internal.Endpoints=tcp
6 IceGrid.Registry.LMDB.Path=/tmp/db/registry
```

[2]Como se puede comprobar la combinación de `Registry` y `Locator` se asemeja mucho a la funcionalidad de un DDNS solo que para objetos en lugar de nodos.

```
7
8  IceGrid.Registry.DynamicRegistration=1
```

La **línea 1** es la referencia al objeto `Locator`. Las **líneas 3–5** definen los *endpoints* en los que escuchan los distintos servicios del servidor. La **línea 6** indica dónde colocar la base de datos que utiliza el programa para guardar su estado, y por último, la **línea 8** activa la propiedad `DynamicRegistration` que permite registrar adaptadores de objetos desconocidos (necesario en este ejemplo).

Debemos crear el directorio referido (en el directorio `/tmp` por ser un ejemplo desechable):

```
$ mkdir -p /tmp/db/registry
```

Y para arrancar el servicio ejecutamos simplemente:

```
locator$ icegridregistry --Ice.Config=registry.config
```

Como es un servidor, el programa queda ocupando la consola indefinidamente.

3.2. Creando proxies indirectos

Por fortuna, los mecanismos que permiten a los servidores Ice utilizar un `Locator` también son transparentes al programador. Lo único que necesitamos para un uso básico es ajustar la configuración. No hay que tocar ni una sola línea de código respecto a nuestro servidor «hola mundo» que creaba proxies directos.

Para nuestro servidor *Printer* simplemente necesitamos un fichero como el siguiente:

Listado 3.2: Configuración del servidor *Printer* para uso de un `Locator`.
locator/server.config

```
1  PrinterAdapter.Endpoints=tcp
2  PrinterAdapter.AdapterId=PrinterAdapter1
3  PrinterAdapter.Locator=IceGrid/Locator -t:tcp -h 127.0.0.1 -p 4061
```

La **línea 1** es equivalente al fichero de configuración que ya usamos en el «hola mundo», salvo que en esta ocasión no incluye el número de puerto, de modo que el *runtime* lo elegirá arbitrariamente.

La **línea 2** es obligatoria y sirve para elegir un identificador único para el adaptador de objetos `PrinterAdapter` dentro del `Locator`. Por último, la **línea 3** fija para nuestro adaptador el proxy del `Locator` que es proporcionado por el programa que ejecutamos en la sección anterior.

Por ejemplo, para lanzar el servidor Python de `Printer` ejecuta el siguiente comando. Por supuesto, debe funcionar exactamente igual con las versiones para otros lenguajes.

```
py$ ./server.py --Ice.Config=../locator/server.config
printer1 -t @ PrinterAdapter1
```

Como se puede comprobar, el mismo servidor (idéntico código) ha generado un proxy indirecto, por el mero hecho de haber configurado un `Locator`.

3.3. Invocando un proxy indirecto

Tampoco es necesario modificar el código del cliente para poder utilizar un proxy indirecto, pero se le debe indicar el proxy del `Locator` para que pueda resolver los proxies indirectos. Para ello sirve el siguiente fichero de configuración:

Listado 3.3: Configuración del cliente para poder resolver proxies indirectos. icegrid/locator.config

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

Para ejecutar el cliente, debemos indicarle el fichero de configuración anterior y el proxy indirecto que imprimió el servidor en su salida.

```
py$ ./client.py --Ice.Config=../locator/locator.config "printer1 -t @ PrinterAdapter1"
```

Al ejecutar este comando, la salida del servidor mostrará el mensaje «Hello World!» corroborando que todo funciona como se esperaba.

3.4. ¿Qué ocurre realmente?

Es decir ¿Cómo puede el mismo cliente (sin cambiar nada en el código) encontrar el servidor y enviar el mensaje de invocación al nodo y puerto correctos?

Lógicamente ocurre porque el *runtime* de Ice modifica su comportamiento al cargar la nueva configuración. Este proceso se denomina *binding* y muy simplificado es algo así:

1. Cuando el servidor crea su adaptador de objetos y comprueba que hay un `Locator` disponible, le envía un mensaje al `Registry` asociado que incluye el `AdapterId` y los *endpoints* en los que va a escuchar el servidor.
2. Como el adaptador de objetos está registrado, cuando se añade el sirviente, el adaptador crea por defecto un proxy indirecto para el objeto.
3. Cuando el cliente trata de utilizar el proxy para enviar un mensaje y comprueba que es indirecto, utiliza el `Locator` disponible para obtener el correspondiente proxy directo.
4. Una vez dispone del proxy directo, puede construir un mensaje de invocación y enviarlo de la forma habitual.

Este proceso es transparente para el programador y ocurre la primera vez que se invoca a un objeto. Las invocaciones sucesivas no lo necesitan porque el *runtime* del cliente cachea los proxies ya resuelto. Si al utilizar un proxy cacheado ocurriera un fallo porque el servidor ya no está en ese lugar, el cliente trataría de conseguir el nuevo proxy directo (si es que lo hay) todo de forma transparente.

3.5. Ventajas

Como ya se ha dicho anteriormente, gracias a los proxies indirectos conseguimos *transparencia de localización*, que en la práctica implica que la referencia que usa el cliente:

- Es la misma aunque el nodo servidor cambie de dirección IP.
- Es la misma aunque el servidor arranque en otro puerto.
- Es la misma aunque una nueva versión del servidor disponga de nuevos *endpoints* y el cliente podrá usarlos.
- Es la misma aunque el servidor pare y arranque en otro nodo.

Para ilustrar este último punto, pruebe el cliente `py/client-loop.py` que repite la misma invocación cada segundo. Para y vuelve a arrancar el servidor mientras el cliente sigue en funcionamiento. Puedes comprobar cómo el cliente puede invocar la nueva instancia tan pronto esté disponible. Y funciona del mismo modo aunque la nueva de instancia del servidor la arranque en un nodo distinto.

Capítulo 4p

Gestión de aplicaciones distribuidas

[Python]

En este capítulo veremos el uso básico de `IceGrid`, un servicio incluido en ZeroC Ice que facilita la gestión y despliegue de aplicaciones distribuidas en grid y proporciona mecanismos de redundancia y tolerando a fallos entre otros. Además se introduce `IcePatch2`, un servicio de despliegue de software. Juntos permiten aplicar procedimientos avanzados sin necesidad de escribir ni una sola línea de código.

4p.1. Introducción

Cuando cualquier sistema crece, inmediatamente aparecen problemas de gestión. Necesitamos saber qué servicio funciona y cuál falla, cuál está activo, cuál ha caído o está inaccesible. Necesitamos mecanismos para distribuir el código de cada servidor en el nodo que le corresponde, arrancarlo o pararlo y diagnosticar problemas que puedan estar ocurriendo.

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 y escalable. Todo ello es posible en buena media gracias a los proxies indirectos que vimos en el capítulo 3.

4p.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.

4p.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, aunque puede estar replicado. 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 sistemas aislados con sus propias aplicaciones distribuidas, incluso aunque usen los mismos nodos.

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 nodos de cómputo 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 nodo de cómputo, 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 configuraciones 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.

4p.2.2. Configuración de IceGrid

Lo primero es configurar un conjunto de nodos 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 computador, 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*. El *Locator* lo proporciona la instancia del Registry y tiene la estructura que se muestra en el siguiente listado:

Listado 4p.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 nodo que ejecuta el Registry. En este ejemplo mínimo, todos los componentes estarán en el mismo computador, por ese motivo, usamos esta dirección.

Recuerda que la dirección IP 127.0.0.1 es una dirección especial que se asigna a la interfaz *loopback*. 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 «hola mundo». Por tanto, se deben proporcionar los *endpoints* para el adaptador de objetos del nodo IceGrid.

Listado 4p.2: Configuración del nodo 1 (continuación)
icegrid/node1.config

```
1 IceGrid.Node.Name=node1
2 IceGrid.Node.Data=/tmp/db/node1
3 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 4p.3: Configuración del Registry (continuación)
icegrid/node1.config

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

IANA

La IANA estandariza los puertos TCP y UDP para servicios comunes.

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**).

Listado 4p.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.LMDB.Path=/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 4p.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 4p.6: Configuración del Registry (continuación)
icegrid/node1.config

```
IceGrid.Registry.DefaultTemplates=/usr/share/ice/templates.xml
```

La ruta del fichero `templates.xml` puede ser diferente en función de la versión de Ice instalada en el sistema.

Como se mencionó 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 4p.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 los clientes que deseen utilizar el `Locator`:

Listado 4p.8: Configuración del Locator
icegrid/locator.config

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

4p.2.3. Arrancando IceGrid

El nodo IceGrid normalmente se ejecuta como un *servicio* del sistema (coloquialmente llamados *demonios*) que arranca automáticamente al conectar cada computador asociado al sistema distribuido y que se ejecuta en *background*. En este caso y con propósitos didácticos vamos a proceder a arrancarlos manualmente y en primer plano. Por tanto, si tu instalación de Ice los ha arrancado como *demonios* debes pararlos antes de continuar.

Abre un 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 *node 1*:

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

Ese terminal queda ocupado con la ejecución del nodo. Déjalo así, en la salida puede imprimir información útil si algo va mal. Cuando acabemos podrás pulsar Control-C para pararlo.

Abre un segundo 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:

```
icegrid$ mkdir -p /tmp/db/node2
icegrid$ 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`, que puedes lanzar en un tercer terminal:

```
$ icegridgui
```

Deberías ver una ventana similar a la de la figura 4p.1.

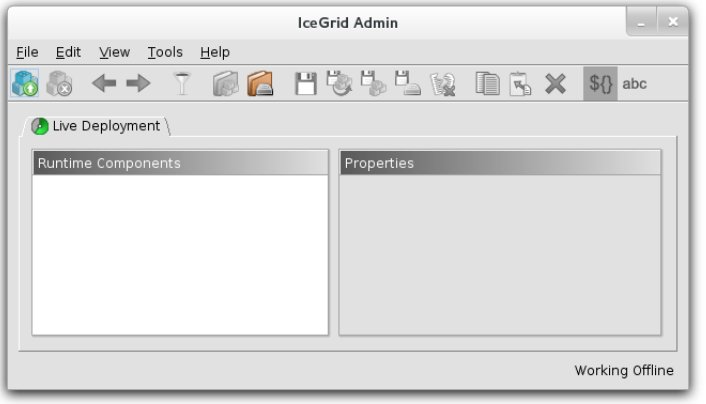


Figura 4p.1: IceGrid Admin: Ventana inicial

4p.3. Creando una aplicación distribuida

A continuación crearemos una nueva aplicación con `icegridgui`. Pero primero necesitamos conectar con el Registry pulsado 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 4p.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 4p.3). Elige *Direct connection*, pulsa Next.
2. Marcar *Connect to a Master Registry*, pulsa Next.
3. Elegir el Registry descubierto automáticamente, pulsa Next.
4. Aceptar los endpoints descubiertos, pulsa Next.
5. Credenciales (figura 4p.4). Escribe «user» y «pass» y pulsa Finish. Por supuesto, en una aplicación en producción, se requeriría una contraseña o una mecanismo de autenticación equivalente. Recuerda que el registro está configurado con un verificador de permisos *dummy* (falso).

Cuando la aplicación ha establecido el acceso al registro debería aparecer una ventana similar a la de la figura 4p.6. 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 Registry al

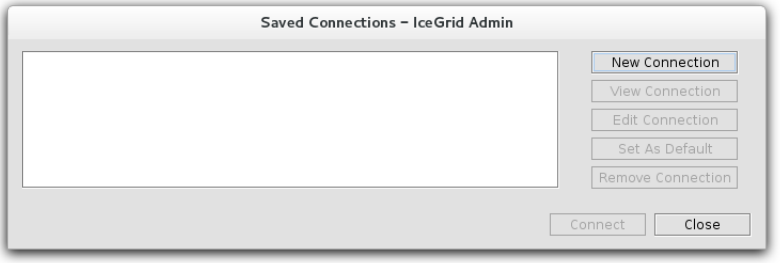


Figura 4p.2: IceGrid Admin: Conexiones guardadas

Figura 4p.3: IceGrid Admin: Nueva conexión

Figura 4p.4: IceGrid Admin: Credenciales

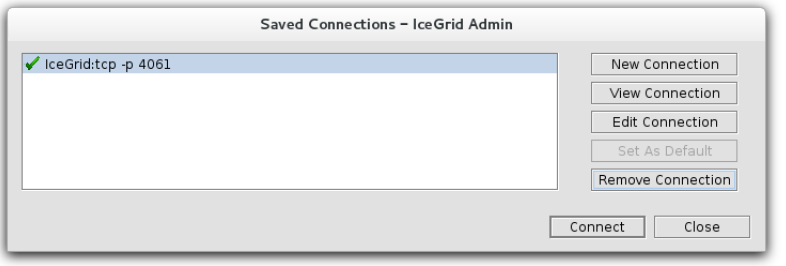


Figura 4p.5: IceGrid Admin: Conexión creada

que has conectado. Puedes pulsar los iconos que representan esos nodos y obtener datos actualizados con información sobre su sistema operativo, su carga, etc.

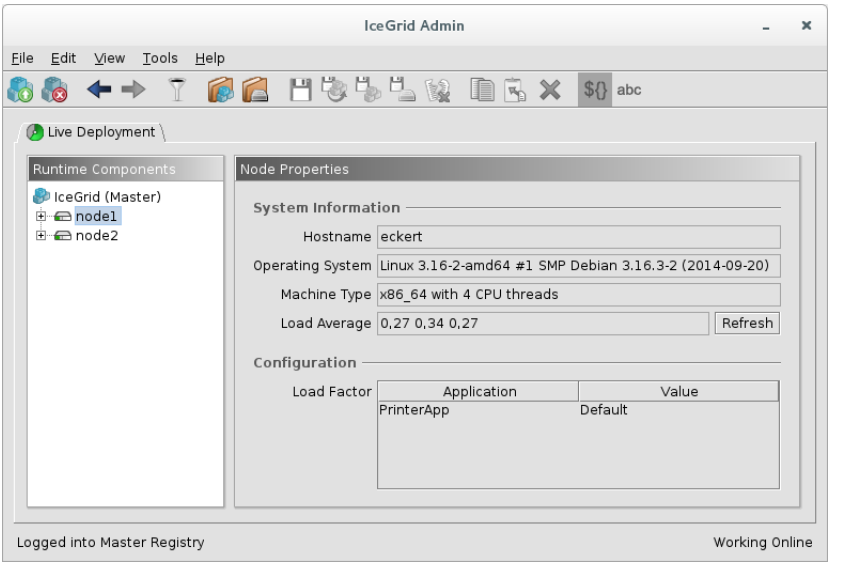


Figura 4p.6: IceGrid Admin: Ventana *Live Deployment*

Para crear una nueva aplicación selecciona la opción de menú *File, New, Application with Default Templates from Registry* (figura 4p.7). Aparecerá una nueva pestaña llamada *NewApplication*. Edita el nombre de la aplicación, escribe «PrinterApp» y pulsa Apply.

Abre el menú contextual (botón derecho del ratón) de la carpeta *Nodes* situado en el panel izquierdo (figura 4p.8). 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 nombres corresponden (respetando mayúsculas y minúsculas).

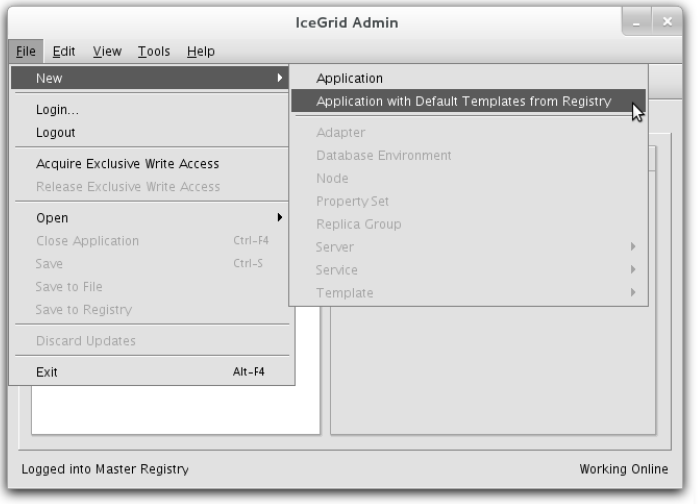


Figura 4p.7: IceGrid Admin: Creación de aplicaciones

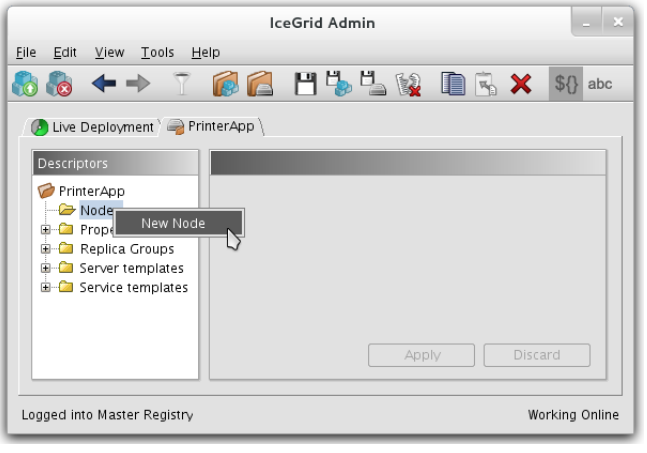


Figura 4p.8: IceGrid Admin: Creación de un nodo

4p.4. Despliegue de aplicaciones con `IcePatch2`

El «despliegue» (en inglés *deployment*) es una actividad presente en el desarrollo e implantación de cualquier sistema software. Se trata de colocar en cada nodo el software necesario para que pueda hacer su parte (servidores, en el caso de Ice).

El despliegue es el proceso de instalación del software de la aplicación en los diferentes nodos que forman el sistema. Pero no vamos a empaquetar los binarios de nuestra aplicación ni tenemos un instalador. Dado que la aplicación puede ejecutarse sobre un grid con muchos nodos, la forma más lógica y eficiente es utilizar la red, por supuesto.

IcePatch2 es un servidor de ficheros que puede coordinar la copia de los ficheros que corresponden a cada nodo. Además, calcula *digest* de los ficheros para evitar copiar ficheros que ya están en el nodo y no han sido modificados, ahorrando así tiempo y ancho de banda. Además IcePatch2 puede tener en cuenta la arquitectura de cada nodo, de modo que se desplieguen a cada nodo solo los binarios compilados para cada arquitectura.

Veamos cómo configurar IcePatch2 para hacer el despliegue de nuestra aplicación. En el menú contextual del nodo `node1` pulsa *New Server from Template* (ver figura 4p.9). Selecciona la plantilla *IcePatch2* en la lista desplegable de la parte superior del diálogo (ver figura 4p.10). En el parámetro *directory* introduce la ruta absoluta del directorio que contiene los binarios de la aplicación «hola mundo» y pulsa Apply.

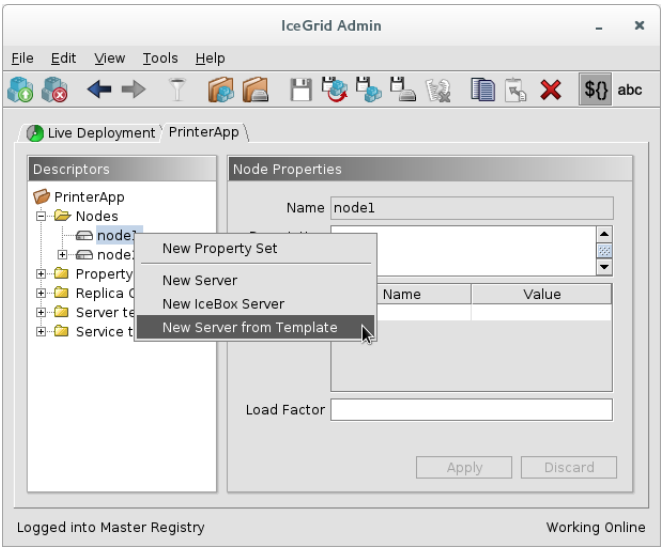


Figura 4p.9: IceGrid Admin: Añadiendo el servidor IcePatch2

Ahora tienes una instancia del servicio IcePatch2 configurada en el nodo `node1`. 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

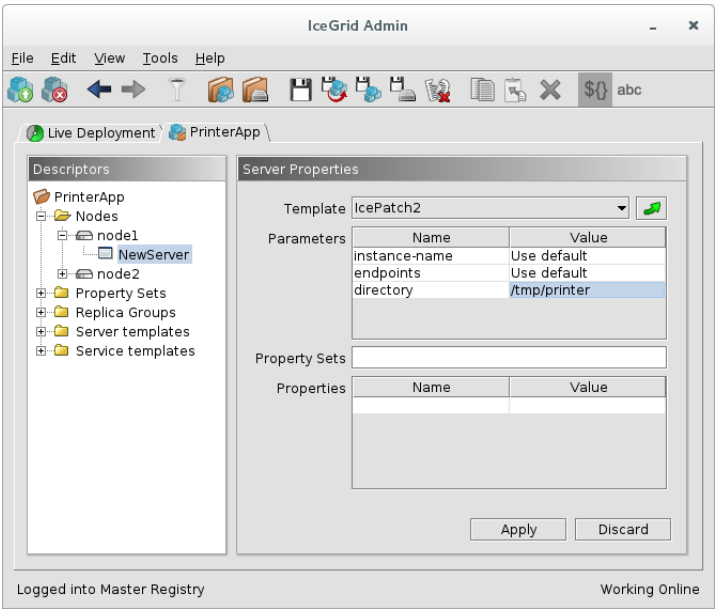


Figura 4p.10: IceGrid Admin: Configuración del servicio IcePatch2

con la instancia recién creada. Pulsa *PrinterApp* arriba a la izquierda y en el desplegable *IcePatch2 Proxy* elije la única que aparece (ver figura 4p.11).

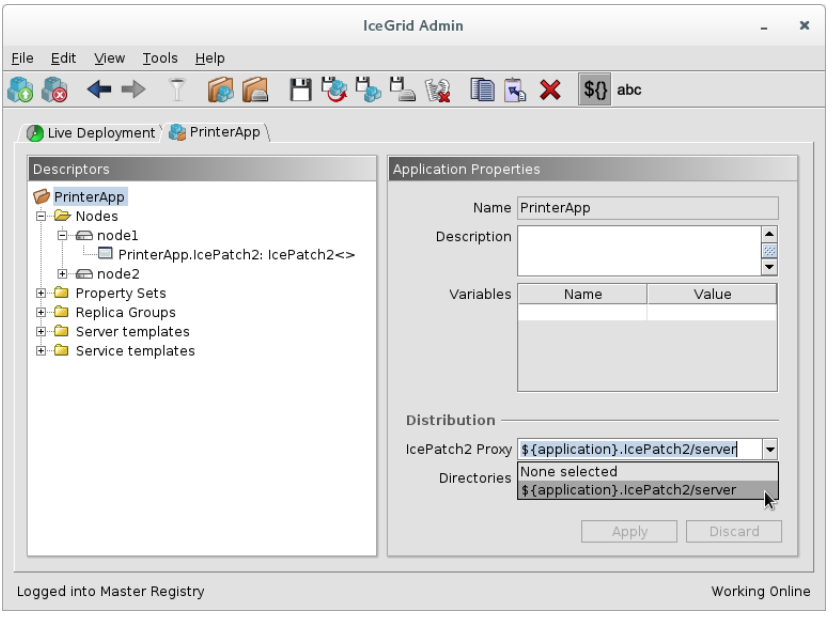


Figura 4p.11: IceGrid Admin: Configuración del proxy IcePatch2 de la aplicación

4p.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`
- En la propiedad *Working Directory* escribe `${application.distrib}` igual que para el cliente.

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 4p.12. 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.

4p.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).

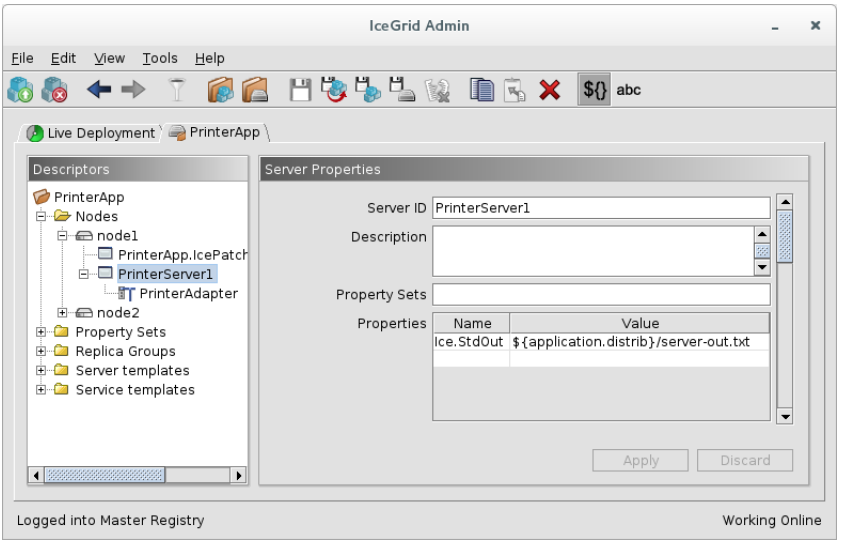


Figura 4p.12: IceGrid Admin: Instancia de la aplicación servidor

4p.6. Ejecutando la aplicación

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

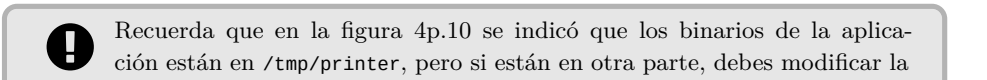
4p.6.1. Despliegue de la aplicación

Es el momento de realizar el despliegue de nuestro código a los 2 nodos usando IcePatch2. Pero antes de eso hay que preparar la «distribución», es decir, los ficheros concretos que queremos desplegar. Eso normalmente implica la obtención de los ficheros binarios luego de un proceso de construcción (compilación principalmente). Pero también necesitamos los ficheros de configuración o cualquier otro que la aplicación necesita durante su ejecución. Lo más sencillo es copiar en un directorio desechable todo lo necesario y así no ensuciar nuestro fichero de código fuente.

Por último, IcePatch2 nos exige una pequeña tarea previa: Calcular los *digest* para cada uno de los ficheros que se van a desplegar. Esto se consigue simplemente ejecutando el comando `icepatch2calc` en el directorio en el que hemos colocado los ficheros y que además debe ser el mismo que indicamos en la configuración del IcePatch2 en `icegridgui`.

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

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



Recuerda que en la figura 4p.10 se indicó que los binarios de la aplicación están en `/tmp/printer`, pero si están en otra parte, debes modificar la propiedad `directory` de IcePatch2 de acuerdo con la situación real.

Ahora selecciona la solapa *Live Deployment* y elije la opción *Tools, Application, Patch Distribution* tal como muestra la figura 4p.13.

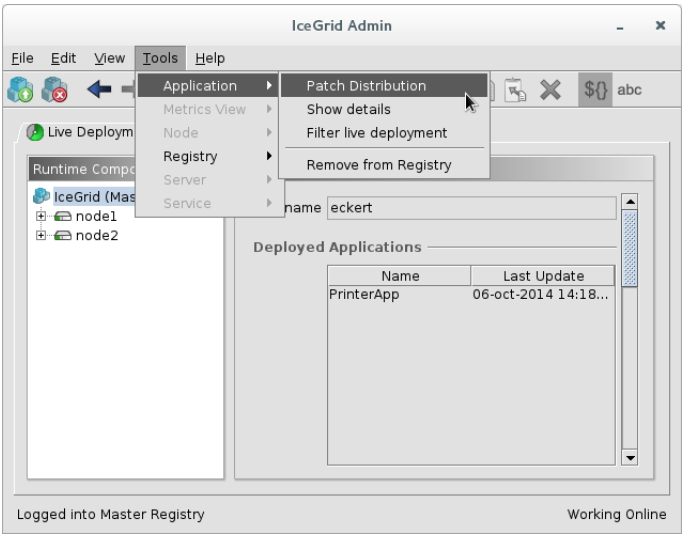


Figura 4p.13: IceGrid Admin: Desplegando la aplicación

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

4p.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 4p.14.

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 un proxy indirecto, dado que cuando un servidor se ejecuta dentro de un nodo IceGrid tiene definido un `Locator` y se comporta de forma similar a como vimos en el capítulo 3.

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

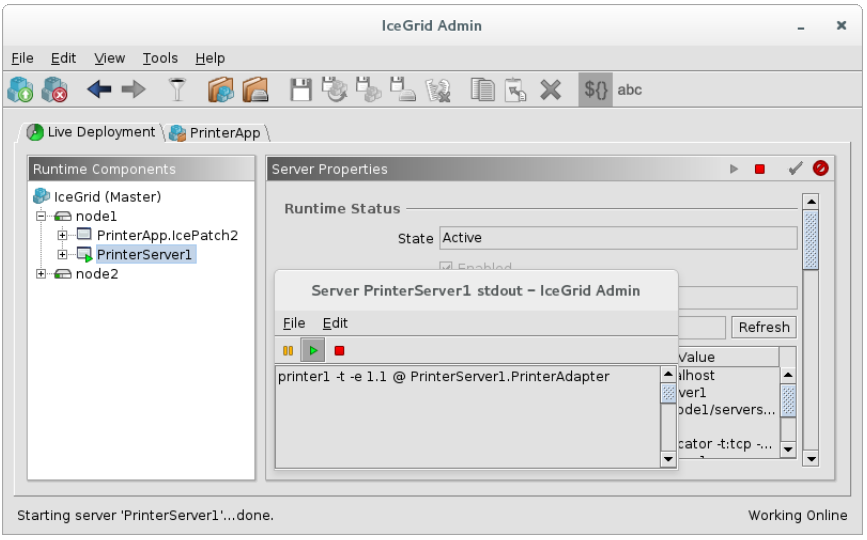


Figura 4p.14: IceGrid Admin: Salida estándar del servidor

Recuerda que 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 4p.15).

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 sin que se repita el identificador del adaptador.

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

4p.6.3. Ejecutando el cliente

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

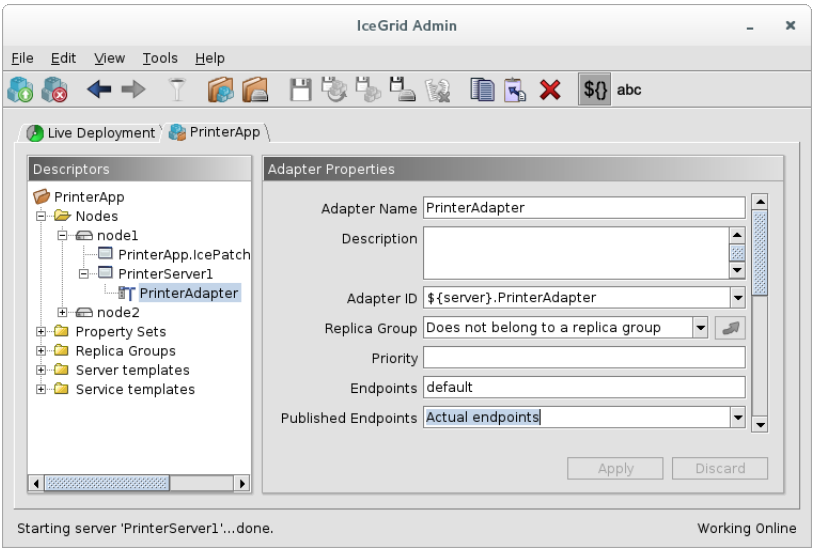


Figura 4p.15: IceGrid Admin: Configuración del adaptador de objetos

```
py$ ./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 4p.16). ¡Funciona!

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.

4p.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 4p.17.

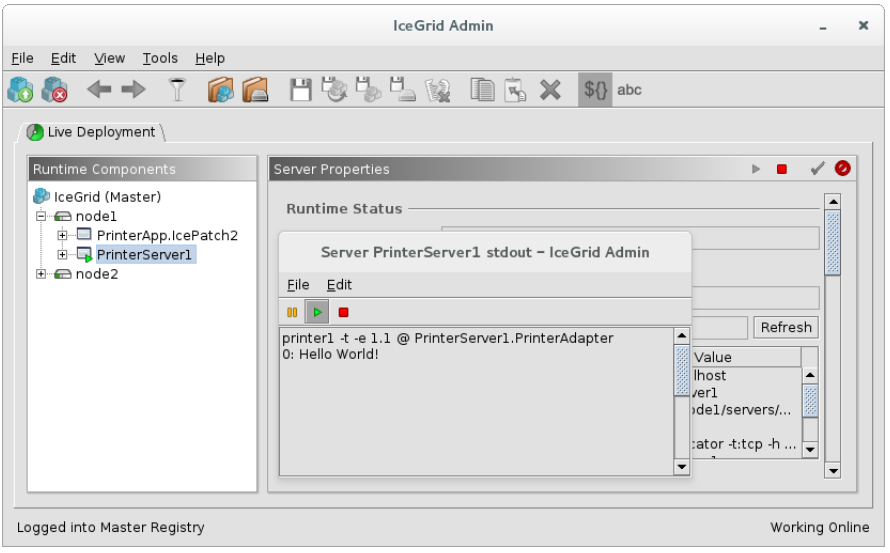


Figura 4p.16: IceGrid Admin: Salida estándar del servidor después de ejecutar el cliente

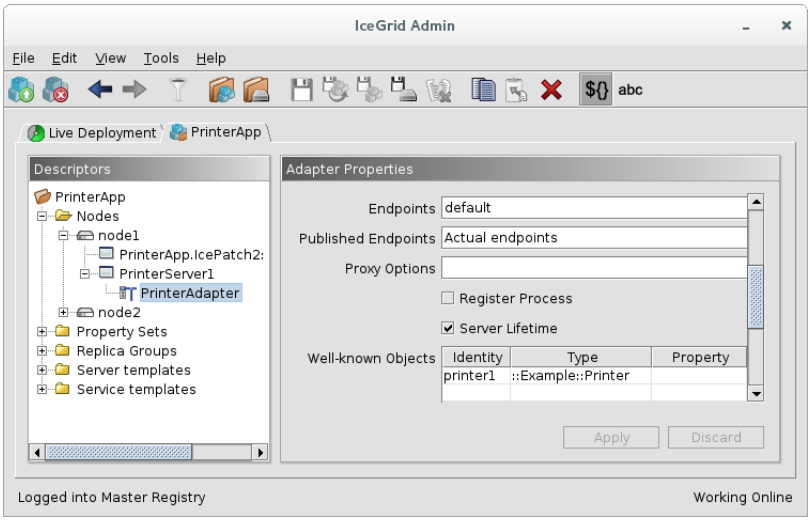


Figura 4p.17: IceGrid Admin: Añadiendo un objeto bien conocido

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

```
py$ ./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` se refiere a un objeto ligado a un adaptador de objetos registrado (aquel que tiene un *Adapter ID*), mientras que la notación `identidad` corresponde con objetos *bien conocidos*.

4p.8. Activación y desactivación implícita

Sería ideal si los servidores solo estuvieran en ejecución cuando realmente son necesarios en lugar de consumir recursos todo el tiempo. 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 Registry 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.

También es bastante conveniente desactivar los servicios ociosos pasado un tiempo de inactividad. Para hacerlo posible 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`'. Pulsa Apply y luego el botón *Save to registry*. Después de 5 segundos de inactividad, el servidor se apagará automáticamente.

4p.9. Depuración

Es fácil cometer algún error durante la configuración de la aplicación distribuida y puede resultar complicado si no se es sistemático al tratar de identificar 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 4p.2.2. Esto nos dará información extra muy útil para localizar y resolver posibles problemas.

4p.9.1. Evitando problemas con IcePatch2

Recuerda que si cambias y recompilas un programa debes desplegar la nueva versión (ver § 4p.6.1). Sin embargo, es bastante frecuente olvidar reiniciar el propio servicio `IcePatch2`. Si el servicio no se reinicia no detectará los cambios.

Una muy buena solución para este problema es configurar la activación implícita y desactivación automática (con unos pocos segundos) del servidor `IcePatch2`.

4p.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 4p.9. Todo lo que hemos hecho en la segunda pestaña de `icegridgui` en realidad solo sirve para editar este fichero XML de una forma más visual.

Listado 4p.9: Descripción de la aplicación PrinterApp en XML
icegrid/printerapp-py.xml

4p.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:

- Crea un fichero de configuración para el `Registry` y para cada uno de los nodos del grid (§ 4p.2.2).
- Arranca un `icegridnode` por cada nodo con su configuración correspondiente (§ 4p.2.3).
- Crea la aplicación distribuida (§ 4p.3).
- Crea un nodo en la aplicación para cada nodo ejecutado.
- Compila los programas, copia los binarios y ficheros necesarios a otro directorio y ejecuta `icepatch2calc`.
- Crea un servidor de `IcePatch2` (§ 4p.4) y configúralo para desplegar el directorio del paso anterior.
- Configura ese servidor como el `IcePatch2` por defecto de la aplicación distribuida (figura 4p.11).

- Crea los servidores IceGrid para ambos programas (4p.5) en sus respectivos nodos.
- Despliega la aplicación (§ 4p.6.1).
- Arranca los servidores (§ 4p.6.2).
- Ejecuta el cliente (§ 4p.6.3).

4p.11. Ejercicios

E 4p.01 Invoca el servidor desplegado con `IceGrid` ejecutando el cliente desde otro servidor `IceGrid`.

E 4p.02 Modifica el código del cliente para que invoque a los dos servidores.

E 4p.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 4p.04 Define nodos adicionales, crea una plantilla[1] 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 las *demos* de Ice[2].

E 4p.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 4p.06 Repite el tutorial de este capítulo usando un computador diferente para cada nodo. Esos computadores pueden ser virtuales (con `VirtualBox` por ejemplo) o contendedores docker.

[1] https://doc.zeroc.com/display/Ice/IceGrid+Templates
[2] https://github.com/zeroc-ice/ice-demos
[3] https://doc.zeroc.com/display/Ice/icegridadmin+Command+Line+Tool

Capítulo 5p

Difusión de eventos

[Python]

Uno de los servicios más sencillos y útiles de Ice es IceStorm. Lo habitual en este tipo de servicios en otros middlewares, como NotificationService de CORBA, DDS o Java Message Service (JMS), es que los eventos que se envían sean objetos o estructuras con datos de estilo Data Transfer Object (DTO). Sin embargo, IceStorm propaga invocaciones a métodos, con lo que es prácticamente una implementación distribuida del patrón de diseño publicación/suscripción, más conocido como «observador» [GHJV94].

Para utilizar el servicio de eventos se requiere un *TopicManager*. Este objeto permite listar, crear y destruir canales (*topics*). Los canales también son objetos, que residen en el mismo servidor del *TopicManager*, dado que es él el que los crea.

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 el en siguiente listado:

Listado 5p.1: Suscriptor Python
py-icestorm/subscriber.py

```
import sys
import Ice
import 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 '{}' not set".format(key))
            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.subscribeAndGetPublisher(qos, subscriber)
        print("Waiting events... '{}'".format(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 24**), 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 30–32**).

Después se obtiene una referencia al canal `PrinterTopic` (**línea 37**), que debería existir. Si no existe se produce una excepción (*NoSuchTopic*), que es capturada y su manejador crea el canal (**línea 39**). En cualquier caso, después de obtener el canal, se suscribe el objeto (**línea 41**). 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 5p.2: Publicador Python
py-icestorm/publisher.py

```
1       def run(self, argv):
2           topic_mgr = self.get_topic_manager()
3           if not topic_mgr:
4               print('Invalid proxy')
5               return 2
6
7           topic_name = "PrinterTopic"
8           try:
9               topic = topic_mgr.retrieve(topic_name)
10          except IceStorm.NoSuchTopic:
11              print("no such topic found, creating")
12              topic = topic_mgr.create(topic_name)
13
14          publisher = topic.getPublisher()
15          printer = Example.PrinterPrx.uncheckedCast(publisher)
16
17          print("publishing 10 'Hello World' events")
18          for i in range(10):
19              printer.write("Hello World %s!" % i)
20
21          return 0
```

Lo más interesante es la obtención del publicador del canal a partir de la referencia al canal (**línea 14**) y el *downcast* para poder invocar sobre él el método de la interfaz *Example::Printer* (**línea 15**). 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!"`.

5p.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 5p.3.

Listado 5p.3: Configuración de IceBox para lanzar IceStorm
py-icestorm/icebox.config

```
IceBox.Service.IceStorm=IceStormService,37: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 5p.4.

Listado 5p.4: Configuración de IceStorm
py-icestorm/icestorm.config

```
1 IceStormAdmin.TopicManager.Default=IceStorm/TopicManager:tcp -p 10000
2 IceStorm.TopicManager.Endpoints=tcp -p 10000
3 IceStorm.Publish.Endpoints=tcp -p 2000
4 IceStorm.Flush.Timeout=2000
5 IceStorm.LMDB.Path=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 5p.5: Configuración del suscriptor
py-icestorm/subscriber.config

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

Y con eso ya se puede arrancar el subscriptor:

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

La configuración para el publicador es:

Listado 5p.6: Configuración de publicador
py-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
```

Al hacer esto, 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.

5p.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.

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.

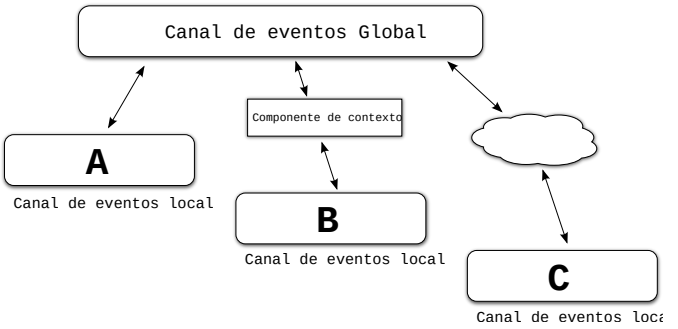


Figura 5p.1: Esquema de federación de canales

En la figura 5p.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 5p.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 5p.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).

5p.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.

Capítulo 6p

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.

6p.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 2p.11**). La ejecución del cliente dura poco más que la pausa que pongamos en el sirviente.

6p.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 Asynchronous Method Invocation (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.

6p.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 6p.1) que describe la operación `factorial()`, es decir, en este caso si hay un valor de retorno:

Listado 6p.1: Especificación Slice para cálculo del factorial (py-ami/factorial.ice)

```
module Example {
  interface Math {
    long factorial(int value);
  };
};
```

El listado 6p.2 muestra el cliente completo utilizando un proxy asíncrono.

Listado 6p.2: Cliente AMI con proxy asíncrono (py-ami/client-end.py)

```
1  import sys
2  import Ice
3  Ice.loadSlice('factorial.ice')
4  import Example
5
6  class Client(Ice.Application):
7      def run(self, argv):
8          proxy = self.communicator().stringToProxy(argv[1])
9          math = Example.MathPrx.checkedCast(proxy)
10
11         if not math:
12             raise RuntimeError("Invalid proxy")
13
14         async_result = math.begin_factorial(int(argv[2]))
15         print('that was an async call')
16
17         print(math.end_factorial(async_result))
18
19         return 0
20
21 if len(sys.argv) != 3:
22     print(f"usage: {__file__} <server> <value>")
23     sys.exit(1)
24
25 app = Client()
26 sys.exit(app.main(sys.argv))
```

La parte interesante de este listado comienza en la **línea 14**, 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 15**) 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 17**), 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 6p.1 muestra todo el proceso.

Tal como se ha indicado anteriormente, el servidor se implementa de la forma habitual. Se muestra en el listado 6p.3.

Listado 6p.3: Servidor para `Math.factorial()` (`py-ami/server.py`)

```
1  import sys
2  import Ice
3
4  Ice.loadSlice('./factorial.ice')
```

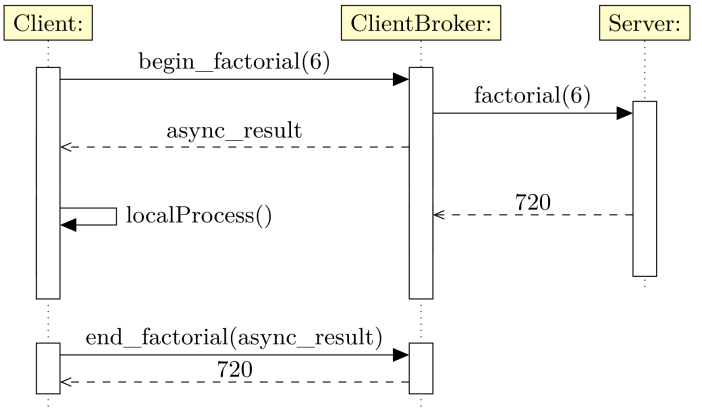


Figura 6p.1: Invocación asíncrona (proxy asíncrono)

```
import Example

def factorial(n):
    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))
```

6p.2.2. Utilizando un *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.

En el listado 6p.4 se muestra el código completo para el cliente que crea y proporciona un *callback* según la técnica indicada.

Listado 6p.4: Cliente AMI que recibe la respuesta mediante un *callback* (py-ami/client-callback.py)

```
1  import sys
2
3  import Ice
4  Ice.loadSlice('./factorial.ice')
5  import Example
6
7  class Client(Ice.Application):
8      def callback(self, future):
9          try:
10             print(f"Callback: value is: {future.result()}")
11         except Exception as ex:
12             print(f"Exception is: {ex}")
13
14     def run(self, argv):
15         proxy = self.communicator().stringToProxy(argv[1])
16         math = Example.MathPrx.checkedCast(proxy)
17         value = int(argv[2])
18
19         if not math:
20             raise RuntimeError("Invalid proxy")
21
22         future = math.factorialAsync(value)
23         future.add_done_callback(self.callback)
24
25         print("That was an async call")
26         return 0
27
28 if len(sys.argv) != 3:
29     print(__doc__.format(__file__))
30     sys.exit(1)
31
32 app = Client()
33 sys.exit(app.main(sys.argv))
34
```

La llamada asíncrona se realiza en la **línea 22** y en la siguiente se le proporciona el *callback* con el método `add_done_callback()`. Cuando la invocación se complete, el *runtime* local invocará el método `callback()` pasando por parámetro un objeto `Future` con el que se puede acceder al resultado, o elevar la excepción.

El diagrama de secuencia de la figura 6p.2 representa las interacciones entre los distintos componentes.

6p.3. Despachado asíncrono de métodos

El tratamiento de métodos asíncrono Asynchronous Method Dispatching (AMD) es la contrapartida de AMI en el lado del servidor. En el tratamiento síncrono

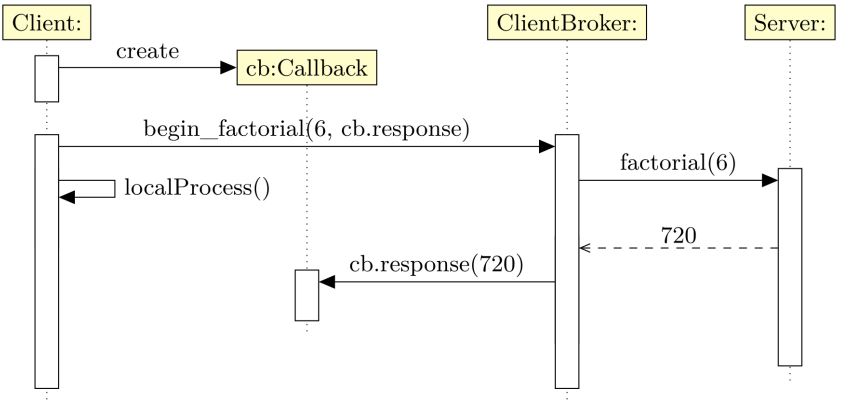


Figura 6p.2: Invocación asíncrona (objeto *callback*)

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 6p.5 muestra el fichero de interfaces que utilizamos en este caso.

Listado 6p.5: Especificación Slice para cálculo del factorial con AMD (py-amd/factorial.ice)

```
1  module Example {
2    interface Math {
3      ["amd"] long factorial(int value);
4    };
5
6    exception RequestCanceledException {};
7  };
```

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 6p.6 muestra el código del servidor AMD.

Listado 6p.6: Servidor AMD (py-amd/server.py)

```
1  import sys
2
3  import Ice
4  Ice.loadSlice('factorial.ice')
5  import Example
6
7  from work_queue import WorkQueue
8
9  class MathI(Example.Math):
10     def __init__(self, work_queue):
11         self.work_queue = work_queue
12
13     def factorial(self, value, current=None):
14         future = Ice.Future()
15         self.work_queue.add(future, value)
16         return future
17
18 class Server(Ice.Application):
19     def run(self, argv):
20         work_queue = WorkQueue()
21         servant = MathI(work_queue)
```

```
22
23          broker = self.communicator()
24
25          adapter = broker.createObjectAdapter("MathAdapter")
26          print(adapter.add(servant, broker.stringToIdentity("math1")))
27          adapter.activate()
28
29          work_queue.start()
30
31          self.shutdownOnInterrupt()
32          broker.waitForShutdown()
33
34          work_queue.destroy()
35          return 0
36
37  sys.exit(Server().main(sys.argv))
```

Las diferencias respecto a un servidor convencional están en la creación de la cola de tareas `queue` (**línea 20**), que se pasa en el constructor del sirviente (**línea 21**), el arranque de la cola (**línea 29**) y la destrucción de la cola al terminar el programa (**línea 34**).

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 13**) es añadir el callback de la invocación (que se recibe como parámetro en `cb`) y el parámetro (`value`) a la cola de tareas (**línea 14**). 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 6p.3.

6p.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 varios técnicas AMI a la vez en el mismo cliente, hacer invocaciones AMI a otros objetos desde un sirviente[1] o en un mismo servidor, utilizar AMD para el despachado y AMI para la invocación a otros objetos.

[1] Esta 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.

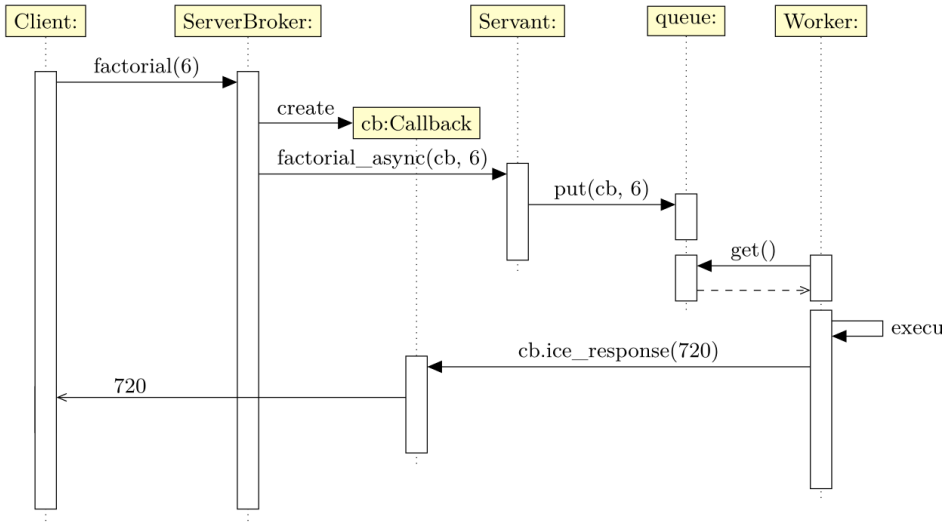


Figura 6p.3: Despachado asíncrono

6p.5. Ejercicios

E 6p.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 6p.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 6p.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 6p.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 6p.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.

Capítulo 7p

Patrones

[Python]

En cualquier middleware orientado a objetos como Ice, los objetos obviamente son los protagonistas. En este capítulos verás algunas soluciones típicas para cubrir algunas necesidades básicas en la gestión y manipulación de objetos. Los denominados *servicios comunes* (los que ofrece el fabricante con el propio middleware) cubren precisamente necesidades de este tipo.

En este capítulo no se explica como funcionan estos *servicios comunes*. En su lugar, veremos cómo implementar versiones minimalistas alternativas, que si bien pueden ser insuficientes para una aplicación en producción, te 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.

7p.1. Contenedor de objetos

Un contenedor (o directorio) de objetos es 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 7p.1: Especificación Slice para el contenedor de objetos
py-container/container.ice

```
1  module Services {
2    exception AlreadyExists { string key; };
3    exception NoSuchKey { string key; };
4
5    dictionary<string, Object*> ObjectPrxDict;
6
7    interface Container {
8      void link(string key, Object* proxy) throws AlreadyExists;
9      void unlink(string key) throws NoSuchKey;
10     Object* get(string key) throws NoSuchKey;
11     ObjectPrxDict list();
12   };
```

```
13 };
```

La interfaz *Container* tiene cuatro 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.

get Obtiene la referencia a un objeto 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 7p.2.

Listado 7p.2: Sirviente del contenedor de objetos
py-container/server.py

```
1  class ContainerI(Services.Container):
2      def __init__(self):
3          self.proxies = dict()
4
5      def link(self, key, proxy, current=None):
6          if key in self.proxies:
7              raise Services.AlreadyExists(key)
8
9          print("link: {0} -> {1}".format(key, proxy))
10         self.proxies[key] = proxy
11
12     def unlink(self, key, current=None):
13         if not key in self.proxies:
14             raise Services.NoSuchKey(key)
15
16         print("unlink: {0}".format(key))
17         del self.proxies[key]
18
19     def get(self, key, current=None):
20         if key not in self.proxies:
21             raise Services.NoSuchKey(key)
22
23         proxy = self.proxies[key]
24         print("get: {0} -> {1}".format(key, proxy))
```

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 sobreescribir 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 atri-

buto diccionario, que será serializado por Ice al tipo correspondiente declarado en el fichero Slice: `ObjectPrxDict`.

E 7p.06 En este ejemplo se ha implementado el contenedor como una tabla asociativa usando una clave arbitraria, pero bien podríamos haber utilizado la propia identidad del objeto, obviando estas claves. Crea una versión del contenedor de acuerdo a esta idea.

E 7p.07 Otra opción podría ser un contenedor que utilice claves en absoluto, es decir, una simple lista, o un conjunto (sin orden ni repeticiones). Implementa también esta versión del contenedor.

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 aplicar el molde (*cast*) al tipo concreto que espera. Este servicio es muy útil cuando una aplicación debe manejar una gran cantidad 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 de la aplicación, y no les afecta en absoluto que sus referencias sean añadidas o eliminadas de éste u otros contenedores.

Como probablemente habrás deducido, Ice ofrece se serie algo similar. 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 o no tan relevantes para la aplicación. Si se obvian las claves, el contenedor puede verse como un *conjunto* de objetos sin un orden específico.

La implementación que proponemos aquí tiene algunas limitaciones e inconvenientes relevantes.

Persistencia

Las referencias que almacena este contenedor no son persistentes. Si el servidor que lo aloja se reinicia o falla se perderá todo la información. Esta limitación podría resolverse con servicio Freeze de Ice u otro sistema de persistencia equivalente. Recuerde que el objetivo principal de los ejemplos de este capítulo es la simplicidad y no la completitud.

E 7p.01 Modifique el sirviente `ContainerI` para añadir soporte de persistencia de modo que, al reiniciarse el servidor, el contenedor tenga los mismos proxies.

Escalabilidad

La interfaz que hemos visto para este servicio no escala adecuadamente. 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 desde luego poco eficiente, incluso con relativamente pocos objetos. Este problema normalmente se solventa creando «iteradores» o «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 determinar qué iteradores están en uso. Es sencillo imaginar un ataque Denial of Service (DoS) contra un servicio que ofrece iteradores.

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.

Sin validación

No se realiza ningún tipo de validación sobre las referencias a objetos. 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 servidor, aunque por supuesto también es una posibilidad a considerar en función de las necesidades de la aplicación.

7p.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 servidor, 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.

[1] otro patrón de diseño: el *iterator*

Pero vayamos un poco más lejos: ¿Podría un cliente crear un nuevo objeto distribuido? En principio no, puesto que, por definición, eso lo convertiría en un 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. Lógicamente en el caso que nos ocupa estamos hablando de una factoría de objetos remotos. Es decir, esto permite a un cliente indicar los argumentos del constructor del sirviente, algo que obviamente no puede hacer él directamente. El listado 7p.3 muestra la interfaz Slice (llamada *PrinterFactory*) para una factoría concreta de objetos *PrinterFactory*.

Listado 7p.3: Interfaz Slice para la factoría de objetos *PrinterFactory*
py-factory/PrinterFactory.ice

```
#include <printer.ice>

module Example {
  exception FactoryError { string reason; };

  interface PrinterFactory {
    Printer* make(string name) throws FactoryError;
  };
};
```

Veamos una implementación sencilla de esta interfaz. El listado 7p.4 muestra el sirviente para la interfaz *PrinterFactory*. En el mismo fichero también se encuentra el sirviente para la clase *Printer*, pero es idéntico al que ya vimos en el capítulo 2p, por lo que no lo repetimos aquí.

Listado 7p.4: Sirviente para la interfaz PrinterFactory
py-factory/server.py

```
class PrinterFactoryI(Example.PrinterFactory):
    def __init__(self):
        self.objects = {}

    def make(self, name, current=None):
        if name in self.objects:
            return self.objects[name]

        print(f"Creating new printer: '{name}'")

        servant = PrinterI(name)
        proxy = current.adapter.addWithUUID(servant)
        printer = self.objects[name] = Example.PrinterPrx.checkedCast(proxy)

        return printer
```

[2] Ver patrón de diseño [GHJV94]

El argumento `name` del método `make()` será un nombre asociado al objeto a crear y como se puede ver, el valor de retorno es de tipo `Printer*`, es decir, *proxy a Printer*.

E 7p.02 El argumento `name` también se podría usar para indicar la identidad del objeto al registrar el sirviente en el adaptador, en lugar generar un UUID como se ve en el ejemplo. Modifica el método `make()` para que se comporte de este modo.

La factoría guarda los proxies de los objetos creados. Si un cliente solicita un objeto con un nombre ya utilizado, la factoría retorna el objeto existente en lugar de crear otro.

El servidor de la factoría no tiene nada especial (ver listado 7p.5); simplemente crea una instancia de *PrinterFactoryI* e imprime el proxy como cualquier servidor Ice básico.

Listado 7p.5: Servidor para la factoría de objetos
py-factory/server.py

```
def run(ic):
    servant = PrinterFactoryI()

    adapter = ic.createObjectAdapter("PrinterFactoryAdapter")
    proxy = adapter.add(servant, ic.stringToIdentity("printerFactory1"))

    print(proxy)

    adapter.activate()
    ic.waitForShutdown()

    return 0

if __name__ == '__main__':
    try:
        with Ice.initialize(sys.argv) as communicator:
            sys.exit(run(communicator))
    except KeyboardInterrupt:
        print("\nShutting down server...")
```

Por último, un cliente que utiliza la factoría para crear e invocar objetos se muestra en el listado 7p.6.

Listado 7p.6: Cliente de la factoría de objetos
py-factory/client.py

```
import sys
import Ice
Ice.loadSlice('-I. --all PrinterFactory.ice')
import Example

def run(ic, args):
    proxy = ic.stringToProxy(args[1])
```

```
    factory = Example.PrinterFactoryPrx.checkedCast(proxy)
    if not factory:
        raise RuntimeError('Invalid proxy')

    printer = factory.make("printer1")
    printer.write('Hello World!')

    return 0

if __name__ == '__main__':
    with Ice.initialize(sys.argv) as communicator:
        sys.exit(run(communicator, sys.argv))
```

Por supuesto, el cliente que solicita la creación del objeto no tiene porqué ser el mismo que lo invoca. El cliente podría enviar el proxy del objeto recién creado por la factoría a un tercer componente (otro servidor) para ser invocado allí.

Contenedor anidado

En la sección 7p.1 hablábamos de la posibilidad de crear un contenedor multinivel, es decir, un contenedor que puedo contener objetos arbitrarios, pero también otros contenedores anidados.

Puede ser una buena idea combinar el contenedor con la factoría, es decir, darle al contenedor la posibilidad de crear otros contenedores que automáticamente añade como contenido propio. El listado 7p.7 muestra una posible interfaz Slice para este contenedor anidado. El método `createContainer()` crea un nuevo contenedor y devuelve su proxy al cliente. El contenedor recién creado se añade automáticamente al contenedor padre.

Listado 7p.7: Interfaz de un contenedor anidado
py-container/nested_container.ice

```
module Services {
  exception AlreadyExists { string key; };
  exception NoSuchKey { string key; };

  dictionary<string, Object*> ObjectPrxDict;

  interface NestedContainer {
    void link(string key, Object* proxy) throws AlreadyExists;
    void unlink(string key) throws NoSuchKey;
    Object* get(string key) throws NoSuchKey;
    ObjectPrxDict list();
    Container* makeContainer(string key);
  };
};
```

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» y no es en absoluto trivial de resolver, pues requiere de mecanismos de captura de estado global.

E 7p.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 7p.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?

7p.3. Factoría IceGrid

IceGrid es a fin de cuentas un gestor de servidores y objetos distribuidos. Es relativamente sencillo utilizar los servicios de IceGrid para crear servidores, no directamente objetos, aunque lógicamente estos servidores van a servir objetos Ice. Por eso, siendo precisos diremos que vamos a crear una factoría de servidores.

IceGrid permite definir plantillas para servidores, que son descripciones en XML de los detalles de un servidor. Veamos una plantilla para servidores de la interfaz *Example.Printer*.

Listado 7p.8: Plantilla XML para servidores *Example.Printer*

```
<server-template id="PrinterTemplate">
  <parameter name="name"/>
  <server id="${name}" activation="on-demand" exe="./printer-server.py" pwd=".">
    <properties>
      <property name="Ice.StdOut" value="./printer.out"/>
      <property name="Ice.StdErr" value="./printer.err"/>
      <property name="Ice.ServerIdleTime" value="5"/>
      <property name="Identity" value="${name}"/>
    </properties>
    <adapter name="PrinterAdapter" endpoints="default"
      id="${server}.PrinterAdapter"/>
  </server>
</server-template>
```

Esta factoría —que puedes encontrar en el directorio `hello.ice/icegrid-factory`— utiliza la misma interfaz del Listado 7p.3. Lo único que cambia un poco es su uso. El argumento de `make()` ahora se utiliza para fijar el nombre del servidor y la identidad del nuevo objeto *Printer*.

No vamos a analizar en detalle aquí la implementación de la factoría, pero podemos resumir el método `make()` de `icegrid-registry/factory.py` como la siguiente secuencia de pasos:

1. Obtiene una referencia al Registry.
2. Crea una sesión para administrar el Registry. Esto requiere usuario y contraseña, aunque en el ejemplo se utilizan credenciales *dummy*.
3. A partir del nombre indicado, busca el nodo IceGrid en el que se va a crear el nuevo servidor (en el ejemplo fijo como `node1`).
4. Comprueba que el nombre de plantilla indicado (en el ejemplo `PrinterTemplate`) está definido en el Registry.
5. Comprueba si ya existe un servidor con el nombre indicado —el argumento de `make()`— y en ese caso obtiene su proxy. Si no existe, entonces crea el servidor y también devuelve el proxy.

El cliente de ejemplo proporcionado (`icegrid-factory/client.py`) es muy similar al del Listado 7p.6. Simplemente invoca 2 veces a `make()` para demostrar que se crean dos servidores (vea `icegrid-factory/client.py`).

Si exploras la aplicación con `icegruidgui` podrás ver los servidores creados dinámicamente por la factoría (ver Figura 7p.1).

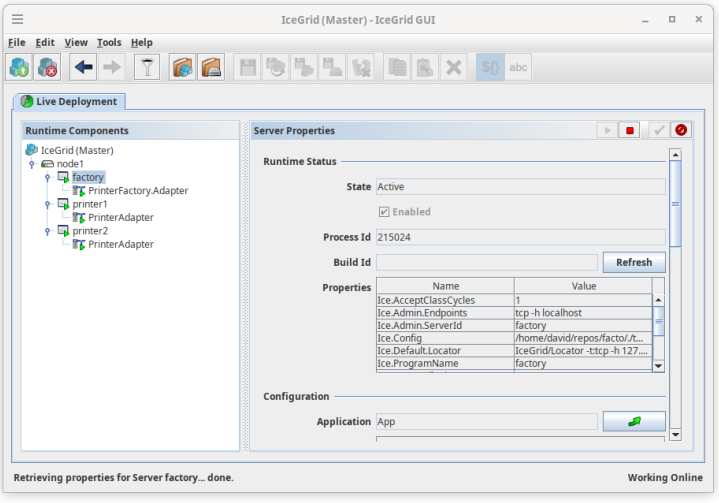


Figura 7p.1: Servidores creados dinámicamente por la factoría

Fíjate que al crear servidores en lugar de objetos. Para la misma cantidad de objetos, la necesidad de recursos de esta factoría es sensiblemente mayor que la de la § 7p.2. Crear procesos es mucho más costoso que crear objetos dentro de un mismo

proceso. Por contra, esta factoría aprovecha las posibilidades que le brinda IceGrid, como la posibilidad de repartir los servidores en distintos nodos o la desactivación automática de servidores (gracias a la propiedad `ServerIdleTime`).

Decimos que esta es una factoría *concreta* porque solo puede crear servidores que contienen objetos *Printer*. Sin embargo, no es difícil implementar una factoría mucho más flexible que permita no solo crear distintos tipos de objetos (basta con utilizar plantillas distintas), también podría aceptar el nombre de la aplicación, el nodo en el que instanciar el servidor y una lista de parámetros con los que instanciar la plantilla. Esta podría ser una posible interfaz para una factoría abstracta y genérica:

Listado 7p.9: Interfaz Slice para la factoría abstracta

```
module Service {
  dictionary<string, string> Parameters;
  exception FactoryError { string reason; };

  interface ServerFactory {
    Object* make(string app, string node, string template, Parameters params)
      throws FactoryError;
  };
};
```

Con este tipo de gestión podemos crear aplicaciones Ice *elásticas* bajo los mismos conceptos que rigen *cloud computing*, es decir, una infraestructura que adapta sus recursos a la demanda, creando y destruyendo servidores en función de la carga, el tráfico o cualquier otro parámetro relevante.

7p.4. Default servant

Una alternativa muy interesante a la creación de objetos es el patrón *default servant* (sirviente por defecto). Esta técnica permite que un único sirviente proporcione la funcionalidad de múltiples objetos —potencialmente una cantidad indefinida— siempre que sean del mismo tipo. Los métodos de cualquier sirviente tienen acceso a la identidad del objeto invocado a través de `current.id` y así el sirviente por defecto la puede utilizar para implementar un comportamiento especifico para cada objeto, por ejemplo, a partir de una configuración disponible en una base de datos u otro tipo de persistencia.

El sirviente por defecto se registra en el adaptador de objetos con el método `addDefaultServant()`. Cuando llega una invocación, el adaptador busca primero la tabla de sirvientes (el ASM) registrados de la forma habitual. Si no lo encuentra, el adaptador utiliza entonces el sirviente por defecto.

En realidad se puede registrar un sirviente por defecto por categoría. La categoría es parte de la identidad de un objeto Ice, por ejemplo, la categoría de la identidad `IceGrid/Locator` es `IceGrid`. También es posible especificar una categoría vacía, en cuyo caso solo hay un sirviente por defecto que se aplica a cualquier identidad.

Vamos un servidor sencillo para la interface *Printer*. El sirviente por defecto debe poder determinar de algún modo qué identidades son válidas y elevar la excepción `ObjectNotExistException` en caso contrario. También es necesario sobrecargar el método `ice_ping()` para que los clientes puedan comprobar la existencia de objetos. Esto es lo que vemos en el Listado 7p.10.

Listado 7p.10: Sirviente por defecto para la interfaz *Printer*
py-default-servant/server.py

```
11 class PrinterI(Example.Printer):
12     def __init__(self, identities):
13         super().__init__()
14         self.valid_identities = set(identities)
15 
16     def ensure_valid_object(self, identity):
17         if identity.name not in self.valid_identities:
18             raise Ice.ObjectNotExistException()
19 
20     def ice_ping(self, current=None):
21         self.ensure_valid_object(current.id)
22 
23     def write(self, message, current=None):
24         self.ensure_valid_object(current.id)
25         print(f"{id2str(current.id)}: {message}")
```

La función `main()` genera 20 identidades, se las pasa como parámetro al sirviente y después crea proxies para cada una de ellas con `createProxy()`. Un cliente convencional puede invocar cualquiera de ellas de la forma habitual. El resto del código del servidor se muestra en el Listado 7p.11.

Listado 7p.11: Servidor para el sirviente por defecto
py-default-servant/server.py

```
28 def main(ic):
29     identities = [f'printer-{uuid4().hex[:8]}' for _ in range(20)]
30     servant = PrinterI(identities)
31 
32     adapter = ic.createObjectAdapter('PrinterAdapter')
33     adapter.addDefaultServant(servant, category='')
34 
35     for identity in identities:
36         print(adapter.createDirectProxy(str2id(identity)))
37 
38     adapter.activate()
39     ic.waitForShutdown()
40     return 0
```

Capítulo 8p

Replicación

En Ice, la replicación implica proporcionar (aparentemente) el mismo adaptador (y sus objetos) en varias localizaciones (endpoints). Aquí el principal objetivo de la replicación es proporcionar redundancia, ejecutando varias instancias del mismo servidor en distintos nodos. Si en una de ellas se produce un error, el servicio sigue estando disponible desde otros lugares.

Utilizar la replicación requiere que las aplicaciones estén diseñadas específicamente. En particular, significa que un cliente observe el mismo comportamiento y obtenga el mismo resultado independientemente de la réplica concreta con la que contacte, ya que el cliente no sabe qué réplica está sirviendo su invocación. Esto es lo que llamamos transparencia de replicación.

El soporte de replicación más básico que proporciona Ice se basa en la posibilidad que tienen los adaptadores de utilizar múltiples endpoints, y funciona incluso con proxies directos y sin necesidad de IceGrid. El núcleo de ejecución de Ice (del cliente) selecciona uno de esos endpoints de manera aleatoria al intentar de conexión inicial. Si la conexión falla, intenta contactar con el siguiente endpoint disponible, es decir, es un mecanismo de *failover*.

Por ejemplo, considera este proxy:

```
1 alarm:tcp -h server1 -p 10001:tcp -h server2 -p 10002
```

Esta referencia indica que el objeto con identidad `alarm` está disponible a través de dos endpoints TCP, uno en el nodo server1 y otro en la nodo server2. Esto podría ocurrir porque el nodo en el que se ejecuta el servidor dispone de dos direcciones IP en la mima red, está conectado a 2 redes diferentes o incluso porque efectivamente el servidor está en ejecución en dos nodos diferentes.

8p.1. Grupo de réplicas

Gracias a IceGrid, Ice soporta métodos de replicación más sofisticados basados en el mecanismo de resolución de los proxies indirectos. Este mecanismo se denomina grupo de réplicas (*replica group*). Un grupo de réplicas tiene un identificador único y aglutina un número arbitrario de adaptadores de objetos. Un adaptador de

objetos solo puede participar en un grupo de réplicas, en cuyo caso decimos que el adaptador está replicado.

Al crear un proxy indirecto, en lugar del identificador de adaptador habitual, podemos utilizar el identificador del grupo de réplicas. En cierto modo, el grupo de réplicas actúa como un adaptador virtual. Solo en el momento de realizar la invocación, el Locator decide cuál (o cuales) de los adaptadores reales formarán parted del proxy directo que devuelve al cliente.

Por ejemplo, un grupo de réplicas identificado por `ToolGroup` puede aparecer en un proxy indirecto en la siguiente forma:

```
tool1 @ ToolGroup
```

Veamos el descriptor de una aplicación IceGrid con un grupo de réplicas empleando el tipo más sencillo: *Random*. Lo puedes encontrar en el directorio `replica-group` del repositorio de ejemplos.

Listado 8p.1: Aplicación IceGrid con un grupo de réplicas (tipo `random`) (`replica-group/random.xml`)

```
<?xml version="1.0" encoding="UTF-8" ?>
<icegrid>
   <application name="App">
      <server-template id="ToolServerTemplate">
         <parameter name="index"/>
         <server id="ToolServer${index}" activation="always" exe="/app/server.py">
            <properties>
               <property name="Ice.StdOut"/>
               <property name="Ice.StdErr"/>
            </properties>
            <adapter name="ToolAdapter" id="${server}.ToolAdapter"
               endpoints="default" replica-group="ToolGroup"/>
         </server>
      </server-template>
      <node name="node1">
         <server-instance template="ToolServerTemplate" index="1"/>
      </node>
      <node name="node2">
         <server-instance template="ToolServerTemplate" index="2"/>
      </node>
      <replica-group id="ToolGroup">
         <load-balancing type="random" n-replicas="1"/>
      </replica-group>
   </application>
</icegrid>
```

El descriptor primero define una plantilla de servidor (*server-template*) de nombre `ToolServerTemplate` para poder crear fácilmente varios servidores idénticos (solo cambia su `id`). Eso es lo que aparece después, dos instancias de esa plantilla, en dos nodos. Por último aparece la parte interesante: la definición del grupo de réplicas `ToolGroup`.

```
<replica-group id="ToolGroup">
    <load-balancing type="random" n-replicas="1"/>
</replica-group>
```

Aquí es donde se define el tipo de balanceo de carga y el número de adaptadores que serán incluidos en el proxy directo creado por el Locator. El tipo `random` significa que el Locator elegirá aleatoriamente entre los adaptadores que forman el grupo. El atributo `n-replicas` indica cuántos adaptadores se incluirán en dicho proxy. Si se quiere conseguir transparencia de replicación, es decir, que el cliente no note que el servidor está replicado, el valor debería ser 1, como en el ejemplo.

Aunque el cliente no sepa que está invocando un replica group, podemos saber a qué servidor concreto está invocando realmente fijando la propiedad `Ice.Trace.Locator=1` (fichero `locator.config`). Aquí puedes ver dos invocaciones consecutivas al mismo proxy indirecto `tool1 @ ToolGroup`, y aparece cómo se traduce a dos proxies directos distintos, cada uno con una IP diferente:

```
replica-group$ ../py-upper/client.py --Ice.Config=locator.config "tool1 @ ToolGroup"
-- 12/12/25 18:13:13.744 Locator: retrieved endpoints for adapter from locator
   adapter = ToolGroup
   endpoints = tcp -h 172.28.0.2 -p 44963 -t 60000
HELLO WORLD!
replica-group$ ../py-upper/client.py --Ice.Config=locator.config "tool1 @ ToolGroup"
-- 12/12/25 18:13:14.817 Locator: retrieved endpoints for adapter from locator
   adapter = ToolGroup
   endpoints = tcp -h 172.28.0.3 -p 40055 -t 60000
HELLO WORLD!
```

El tipo `random` favorece el balanceo de carga entre las réplicas. Aunque es muy simple, estadísticamente, todos los nodos recibirán un número similar de invocaciones a largo plazo. Otro forma habitual de conseguir balanceo de carga es `round-robin`, que también soporta IceGrid. Simplemente define el grupo de réplicas así:

Listado 8p.2: Aplicación IceGrid con un grupo de réplicas (tipo `round-robin`) (`replica-group/round-robin.xml`)

```
<replica-group id="ToolGroup">
    <load-balancing type="round-robin" n-replicas="1"/>
</replica-group>
```

La replicación tiene otros usos, como por ejemplo aumentar la disponibilidad del servicio, es decir, que el servicio siga estando disponible aunque una o varias réplicas fallen. Eso se puede conseguir con el tipo `ordered`. Cada adaptador se marca con un atributo entero `priority`. El Locator elegirá siempre el adaptador disponible con valor de prioridad menor. Por tanto proporciona un mecanismo de respaldo (*failover*) automático. El descriptor en este caso seria:

Listado 8p.3: Aplicación IceGrid con un grupo de réplicas (tipo `ordered`) (`replica-group/ordered.xml`)

```
<?xml version="1.0" encoding="UTF-8" ?>
<icegrid>
   <application name="App">
      <server-template id="ToolServerTemplate">
         <parameter name="index"/>
         <server id="ToolServer${index}" activation="always" exe="/app/server.py">
            <properties>
               <property name="Ice.StdOut"/>
               <property name="Ice.StdErr"/>
            </properties>
            <adapter name="ToolAdapter" id="${server}.ToolAdapter"
               endpoints="default" replica-group="ToolGroup" priority="${index}"/>
         </server>
      </server-template>
      <node name="node1">
         <server-instance template="ToolServerTemplate" index="1"/>
      </node>
      <node name="node2">
         <server-instance template="ToolServerTemplate" index="2"/>
      </node>
      <replica-group id="ToolGroup">
         <load-balancing type="ordered" n-replicas="1"/>
      </replica-group>
   </application>
</icegrid>
```

Del mismo modo que en el caso anterior, puedes comprobar fijando la propiedad `Ice.Trace.Locator=1` en el fichero `locator.config` que todas las invocaciones se envían a `ToolServer1` (con prioridad 1). Si desactivas ese servidor, las invocaciones que realices a continuación serán atendidas por `ToolServer2`. Si el servidor `ToolServer1` vuelve a estar disponible, las invocaciones se dirigirán de nuevo a él.

Ice proporciona un último tipo llamado `adaptive`, que utiliza una heurística para determinar la carga de los distintos nodos en lo que se se ejecutan los adaptadores y elegir el menos cargado. Este tipo es más complejo y requiere que los servidores informen periódicamente al Locator de su carga actual. Se define de este modo:

Listado 8p.4: Aplicación IceGrid con un grupo de réplicas (tipo `adaptive`) (`replica-group/adaptive.xml`)

```
<replica-group id="ToolGroup">
    <load-balancing type="adaptive" load-sample="5" n-replicas="1"/>
</replica-group>
```

El atributo `load-sample` puede tomar únicamente los valores 1, 5 y 15. Indica el intervalo (en minutos) con el que los nodos IceGrid informan de su carga. Corresponden a los intervalos de muestreo estándar de los sistemas POSIX, que se puede consultar entre otros con el comando `uptime`.

Listado 8p.5: Salida del comando `uptime`

```
$ uptime
 22:47:12 up  6:30,  1 user,  load average: 2,23, 1,76, 1,67
```

Referencias

[CDK05] George Coulouris, Jean Dollimore, y Tim Kindberg. *Distributed Systems: Concepts and Design (International Computer Science)*. Addison-Wesley Longman, Amsterdam, edición 4th rev. ed., 2005.

[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.