Este documento fue compuesto con LaTeX. Imágenes generadas con OpenOffice.

Índice general

Capítulo 1

Introducción

1.1. Introducción general a los middlewares

1.1.1. Conceptos

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 distinto, un *middleware* se puede entender como una abstracción en la complejidad y en 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 contrapartida de los distintos desacuerdos en *hardware*, sistemas operativos, protocolos de red, y lenguajes de programación.

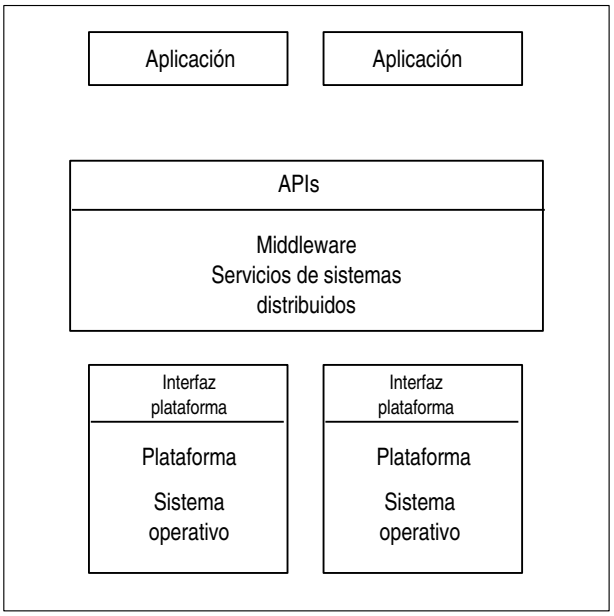

Figura 1.1: Esquema general de un *middleware*

1.1.2. Fundamentos básicos

La mayoría de los *middlewares* tratan de **acercar el modelo de programación a un punto de vista local**, es decir, enmascarando la llamada a los procedimientos remotos. Por otra parte, el enfoque más extendido es el de la generación de un *proxy* en la parte del cliente y de un *esqueleto* en la parte del servidor. Para ello, el cliente utiliza un objeto *proxy* con la misma interfaz definida en la parte del servidor, y que actúa como intermediario. El servidor, por otro lado, utiliza un *esqueleto* encargado de traducir los eventos de la red a invocaciones sobre el objeto en cuestión. Como se puede apreciar, existe un gran acercamiento de la versión distribuida a la versión centralizada.

Las principales **responsabilidades de un *proxy*** son las siguientes:

- Codificar la invocación y los argumentos en un mensaje.
- Esperar la respuesta y decodificar el valor o los valores de retorno.

Por otra parte, las principales **responsabilidades de un *esqueleto*** son las siguientes:

- Esperar una invocación y decodificar el mensaje y los argumentos.

- Invocar el método real.
- Codificar el valor o los valores de retorno en un mensaje de respuesta.

Una cuestión importante a tener en cuenta es la codificación de los datos en la red. Por ejemplo, es posible tener distintas representaciones de un mismo tipo de datos dependiendo del computador empleado. Para solucionar este problema, se suele definir una representación externa canónica, y transformar a y desde un formato binario a partir de la representación externa. Este proceso es conocido como proceso de *marshalling* y *unmarshalling*.

Por otra parte, el *middleware* también ha de gestionar problemas inherentes a las comunicaciones, como por ejemplo la identificación de mensajes, la gestión de retransmisiones, la gestión de conexiones, y la identificación de objetos. Por todo ello, la solución más extendida se basa en un núcleo de comunicaciones genérico y de un generador automático de *proxies* y *esqueletos*.

1.2. Características generales de ICE

ICE (**Internet Communication Engine**) [4] es un *middleware* orientado a objetos, es decir, ICE proporciona herramientas, *APIs*, y soporte de bibliotecas para construir aplicaciones cliente-servidor orientadas a objetos. Una aplicación ICE se puede usar en entornos heterogéneos: los clientes y los servidores pueden escribirse en diferentes lenguajes de programación, pueden ejecutarse en distintos sistemas operativos y en distintas arquitecturas, y pueden comunicarse empleando diferentes tecnologías de red. Además, el código fuente de estas aplicaciones puede portarse de manera independiente al entorno de desarrollo. Los **principales objetivos de diseño de ICE** son los siguientes:

- Proporcionar un *middleware* listo para usarse en sistemas heterogéneos.
- Proveer un conjunto completo de características que soporten el desarrollo de aplicaciones distribuidas reales en un amplio rango de dominios.
- Evitar una complejidad innecesaria, haciendo que ICE sea fácil de aprender y de usar.

- Proporcionar una implementación eficiente en ancho de banda, en uso de memoria, y en carga de CPU.
- Proporcionar una implementación basada en la seguridad, de forma que se pueda usar sobre redes no seguras.

Se puede decir que la filosofía de ICE se basa en construir una plataforma tan potente como CORBA, pero sin cometer todos los fallos de ésta y evitando una complejidad innecesaria.

1.3. Organización de este documento

Este documento está estructurado en cuatro partes:

- Parte I, Un vistazo a ICE, realiza una toma de contacto con el *middleware* ZeroC ICE y expone un ejemplo sencillo de aplicación en ICE.
- Parte II, Slice, alude al lenguaje de definición de interfaces de ZeroC ICE, comentando los aspectos básicos por un lado y algunos de los aspectos más avanzados por otro.
- Parte III, ICE avanzado, expone ciertos aspectos avanzados de ZeroC ICE.
- Parte IV, Servicios en ICE, realiza un estudio de los servicios propuestos por el *middleware* ZeroC ICE desde una perspectiva eminentemente práctica.

Parte I

Un vistazo a ICE

Capítulo 2

Toma de contacto con ICE

2.1. La arquitectura de ICE

2.1.1. Terminología

ICE introduce una serie de conceptos técnicos que componen su propio vocabulario, como ocurre con cualquier nueva tecnología. Sin embargo, el objetivo perseguido fue reutilizar la

mayor parte de terminología existente en sistemas de este tipo, de forma que la cantidad de términos introducidos fuera mínima. De hecho, si el lector ha trabajado con tecnologías relacionadas como CORBA, la terminología aquí descrita le será muy familiar.

2.1.1.1. 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, emiten solicitudes de servicio a un servidor.
- Los servidores son entidades pasivas, es decir, proporcionan un servicio en respuesta a las solicitudes de los clientes.

Normalmente, los clientes no son clientes *puros* en el sentido de que sólo solicitan peticiones. En su lugar, los clientes suelen ser entidades híbridas que asumen tanto el rol de cliente como el de servidor. De hecho, los sistemas cliente-servidor suelen describirse de una forma más eficiente como sistemas *peer-to-peer*.

2.1.1.2. Objetos ICE

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

- Un objeto ICE 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 ICE puede instanciarse en un único servidor o, de manera redundante, en múltiples servidores. Si un objeto tienes varias instancias simultáneamente, todavía sigue siendo un único objeto ICE.
- Cada objeto ICE tiene una o más interfaces. Una interfaz es una colección de operaciones soportadas por un objeto. 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 servidor. (El valor de retorno es simplemente un parámetro de salida especial).

- Un objeto ICE tiene una interfaz distinguida del resto y conocida como la *interfaz principal*. Además, un objeto ICE puede proporcionar cero o más interfaces alternativas, 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.

- Cada objeto ICE 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 dominio de comunicación en ICE.

2.1.1.3. *Proxies*

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

1. Localiza al objeto ICE.

2. Activa el servidor del objeto ICE si no está en ejecución.

3. Activa el objeto ICE dentro del servidor.

4. Transmite los parámetros de entrada al objeto ICE.

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.

2.1.1.4. Proxies textuales (*stringfied proxies*)

La información asociada a un proxy se puede representar como una cadena. Por ejemplo, la cadena

- SimplePrinter:default -p 10000

es una representación legible de un proxy. El núcleo de ejecución de ICE proporciona llamadas a su API 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.

2.1.1.5. 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/IP o UDP).

- Una dirección específica de un protocolo (como el nombre y el puerto de una máquina).

Con el objetivo de contactar con el objeto asociado a un proxy directo, el núcleo de ejecución de ICE 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.

2.1.1.6. 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 *SimplePrinter* es un proxy válido para un objeto bien conocido con la identidad *SimplePrinter*. Un proxy indirecto que incluye un identificador de adaptador de objetos tiene la forma textual *SimplePrinter@PrinterAdapter*. 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.

2.1.1.7. *Binding* directo y *binding* 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.

2.1.1.8. 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.

2.1.1.9. Replicación

En ICE, la replicación implica hacer disponibles en múltiples direcciones a los adaptadores de objetos (y a sus objetos). El **principal objetivo de la replicación** es el de proporcionar redundancia ejecutando el mismo servidor en distintas máquinas. Si en una de ellas se produce un error, el servidor permanece disponible en el resto.

Utilizar la replicación también implica que las aplicaciones estén diseñadas teniendo en cuenta dicho concepto. En particular, significa que un cliente pueda acceder a un objeto a través de una dirección y que obtenga el mismo resultado si hubiera accedido a través de otra.

ICE soporta una forma limitada de replicación cuando un proxy especifica múltiples direcciones para un objeto. El núcleo de ejecución de ICE selecciona una de esas direcciones de manera aleatoria para su intento de conexión inicial, e intenta contactar con el resto en caso de fallo. Por ejemplo, considere este proxy:

- SimplePrinter:tcp -h server1 -p 10001:tcp -h server 2 -p 10002

El proxy indica que el objeto identificado por *SimplePrinter* está disponible utilizando dos direcciones TCP, una en la máquina *server1* y otra en la máquina *server2*. Se supone que los administradores del sistema garantizan que el servidor está en ejecución tanto en la máquina *server1* como en la máquina *server2*.

2.1.1.10. Grupos de réplica

Además de la replicación basada en proxies comentada en la sección anterior, ICE soporta una forma más útil de replicación conocida como grupos de réplica que requiren el uso de un **servicio de localización**.

Un grupo de réplica tiene un identificador único y consiste en un número determinado de adaptadores de objetos. Un adaptador de objetos puede ser miembro de al menos un grupo de réplica. Dicho adaptador se considera como un adaptador de objetos replicado.

Después de haber establecido un grupo de réplica, su identificador puede utilizarse como un proxy indirecto en lugar de un identificador de adaptador. Por ejemplo, un grupo de réplica identificado por *PrinterAdapters* puede utilizarse como un proxy de la siguiente forma:

- SimplePrinter@PrinterAdapters

Un grupo de réplica es tratado por el servicio de localización como un adaptador de objetos virtual. El comportamiento del servicio de localización cuando resuelve un proxy indirecto que contiene el identificador de un grupo de réplica es un aspecto relacionado con la implementación. Por ejemplo, el servicio de localización podría tomar la decisión de devolver las direcciones de todos los adaptadores de objetos del grupo, en cuyo caso el núcleo de ejecución de la parte del cliente podría seleccionar una de esas direcciones de manera aleatoria utilizando la forma limitada de replicación comentada en la anterior sección. Otra posibilidad para el servicio de localización sería la de devolver una única dirección seleccionada en función de una determinada heurística.

Sin tener en cuenta cómo el servicio de localización resuelve un grupo de réplica, el principal beneficio es la indirección: el servicio de localización puede añadir más inteligencia al proceso de resolución actuando como intermediario.

2.1.1.11. 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 ICE. 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 ICE. 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 ICE en un determinado momento o a varios objetos ICE de manera simultánea. En el primer caso, la identidad del objeto ICE encarnado por el sirviente está implícita en el sirviente. En el segundo caso, el sirviente mantiene la identidad del objeto ICE con cada solicitud, de forma que pueda decidir qué objeto encarnar mientras dure dicha solicitud.

En cambio, un único objeto ICE puede tener múltiples sirvientes. Por ejemplo, podríamos tomar la decisión de crear un proxy para un objeto ICE 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 ICE. 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 ICE 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.

2.1.1.12. Semántica *at-most-once*

Las solicitudes ICE tienen una semántica *at-most-once*: el núcleo de ejecución de ICE 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 es idempotente es una operación 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 ejecución de ICE utiliza un mecanismo de recuperación de error más agresivo que para las operaciones no idempotentes.

2.1.1.13. Invocación de métodos síncrona

Por defecto, el modelo de envío de peticiones utilizado por ICE está basado en la llamada síncrona a procedimientos remotos: una invocación de operación se comporta como una llamada local a un procedimiento, es decir, el hilo del cliente se suspende mientras dure la llamada y se vuelve activar cuando la llamada se ha completado (y todos los resultados están disponibles).

2.1.1.14. Invocación de métodos asíncrona

ICE también proporciona invocación de métodos asíncrona (*asynchronous method invocation*) o AMI: un cliente puede invocar operaciones de manera asíncrona, es decir, el cliente utiliza un proxy de la manera habitual para invocar una operación pero, además de pasar los parámetros necesarios, también pasa un objeto de retrollamada (*callback object*) y retorna de la invocación inmediatamente. Una vez que la operación se ha completado, el núcleo de ejecución de la parte del cliente invoca a un método en el objeto de retrollamada pasado inicialmente, proporcionando los resultados de la operación a dicho objeto (o en caso de error, proporcionando la información sobre la excepción asociada).

El servidor no es capaz de diferenciar una invocación asíncrona de una síncrona. De hecho, en ambos casos el servidor simplemente aprecia que un cliente ha invocado una operación en un objeto.

2.1.1.15. Tratamiento de métodos asíncrono

El tratamiento de métodos asíncrono (*asynchronous method dispatch*) o AMD es el equivalente a AMI en el lado del servidor. En el tratamiento síncrono por defecto el núcleo de ejecución de la parte del servidor y mientras la operación se está ejecutando (o *durmiendo* debido a que espera para obtener un dato), el hilo de ejecución asociado al servidor sólo se libera cuando la operación se ha completado.

Con AMD se informa de la llegada de la invocación de una operación al código de la aplicación de la parte del servidor. Sin embargo, en lugar de forzar al proceso a atender la petición inmediatamente, la aplicación de la parte del servidor puede retrasar el procesamiento de dicha petición y liberar el hilo de ejecución asociado a la misma. El código de aplicación de la parte del servidor es ahora libre de hacer lo que quiera. Eventualmente, una vez que los resultados de la aplicación están disponibles, el código de aplicación de la parte del servidor realiza una llamada al API para informar al núcleo de ejecución de la parte del servidor de que la petición que se realizó con anterioridad ha sido completada. En este momento, los resultados de la ooperación invocada se envían 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 devuelva 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 esperara 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, centenas o incluso miles de clientes podrían bloquearse en la misma invocación sin estar vinculados a los hilos del servidor.

El tratamiento síncrono y asíncrono de métodos son transparentes al cliente, es decir, el cliente es incapaz de saber si un determinado servidor realiza un tratamiento síncrono o, por el contrario, un tratamiento asíncrono.

2.1.1.16. Invocación de métodos en una dirección (*Oneway Method Invocation*)

Los clientes pueden invocar una operación como una operación en una 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 actual 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 reales. 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 salidad, 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/IP o SSL.

2.1.1.17. Invocación de métodos en una dirección y por lotes (*batched oneway method invocation*)

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.

2.1.1.18. *Datagram invocations*

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

- Las invocaciones pueden llegar fuera de orden debido a la naturaleza del protocolo UPD.

Este tipo de invocaciones cobrán 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.

2.1.1.19. *Batched datagram invocations*

Este tipo de invocaciones son análogas a las *batched oneway invocations*, pero en el ámbito del protocolo UDP.

2.1.1.20. 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.

2.1.1.21. 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.

2.1.1.22. Propiedades

La mayor parte del núcleo de ejecución de ICE se puede 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 y son trasladadas al

núcleo de ejecución de ICE para configurar diversas opciones, como el tamaño del *pool* de hilos, el nivel de traceado, y muchos otros parámetros de configuración.

2.1.2. Slice (Lenguaje de especificación para ICE)

Como se mencionó anteriormente, cada objeto ICE tiene una interfaz con un determinado número de operaciones. Las interfaces, las operaciones, y los tipos de datos intercambiados entre el cliente y el servidor se definen utilizando el lenguaje Slice. Slice permite definir el contrato entre el cliente y el servidor de forma independiente del lenguaje de programación empleado. Las definiciones Slice se compilan por un compilador a una API para un lenguaje de programación específico, es decir, la parte de la API que es específica a las interfaces y los tipos que previamente han sido definidos y que consisten en código generado.

2.1.3. *Mappings* de lenguajes

Las reglas que rigen cómo se traslada cada construcción Slice en un lenguaje de programación específico se conocen como *mappings* de lenguajes. Por ejemplo, para el *mapping* a C++, una secuencia en Slice aparece como un vector STL, mientras que para el *mapping* a 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 *mapping* hacia dicho lenguaje.

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

2.1.4. Estructura cliente-servidor

Los clientes y los servidores ICE tienen la estructura lógica interna mostrada en la figura 2.1.

Tanto el cliente como el servidor se pueden entender como una mezcla de código de aplicación, código de bibliotecas, y código generado a partir de las definiciones Slice:

- El núcleo de ICE contiene el soporte de ejecución para las comunicaciones remotas

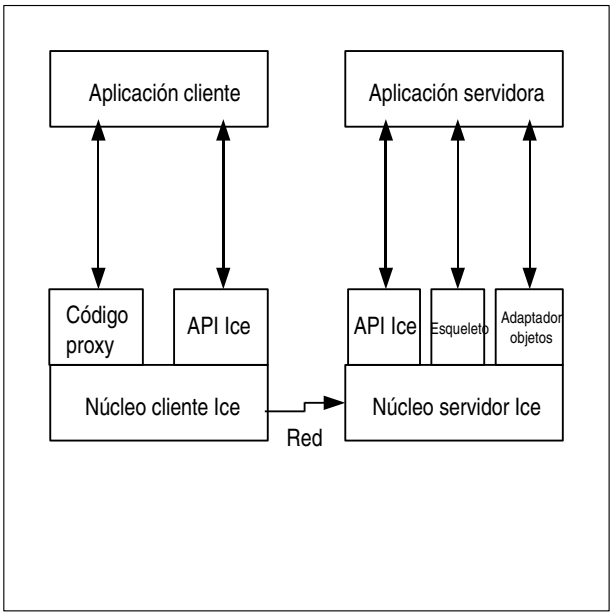

Figura 2.1: Estructura cliente-servidor

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.

- La parte genérica del núcleo de ICE, a la cual se accede a través del API de ICE. El desarrollador utiliza esta API para la gestión de tareas administrativas, como por ejemplo la inicialización y finalización del núcleo de ejecución de ICE.

- El código de los proxies se genera a partir de las definiciones en Slice y, por lo tanto, es específico a los tipos de objetos y de datos definidos en Slice. Dicho código tiene dos funciones principales: proporciona una interfaz para el cliente y proporciona código de *marshaling* y *unmarshaling*. *Marshaling* es el proceso de serializar una estructura de datos compleja, como una secuencia o un diccionario, para la transmisión por la red. Este código convierte los datos en una forma estándar para la transmisión de forma independiente de la arquitectura de las máquinas involucradas (como por ejemplo *big-endian*) y de las reglas de relleno. *Unmarshaling* es el proceso contrario.

- El código del esqueleto también se genera a partir de la definición en Slice y, por lo tanto, es específico a los tipos de objetos y datos definidos en Slice. Dicho código es el equivalente al generado para el proxy pero en la parte del servidor.

- El adaptador de objetos es una parte de la API de ICE específico al lado del servidor: sólo los servidores utilizan los adaptadores de objetos. Un adaptador de objetos tiene varios funciones, como traducir las peticiones de los clientes a los métodos específicos al lenguaje de programación empleado, asociarse a uno o más puntos finales de transporte, o crear los proxies que pueden pasarse a los clientes.

2.1.5. El protocolo ICE

ICE proporciona un protocolo RPC que puede utilizar tanto TCP/IP como UPD como capa de transporte subyacente. Además, ICE permite utilizar SSL, de forma que todas las comunicaciones entre el cliente y el servidor estén encriptadas.

El protocolo ICE define:

- Un determinado número de tipos de mensajes, como por ejemplo de solicitud y respuesta.

- 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 codificicación que determinan cómo se representa cada tipo de datos en la red.

- Una cabecera para cada tipo de mensaje que contiene detalles como el tipo del mensaje, el tamaño del mensaje, y el protocolo y la versión de codificiación empleados.

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 resulta útil si la aplicación intercambia grandes cantidades de datos entre el cliente y el servidor.

El protocolo ICE 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.

2.1.6. Persistencia de objetos

ICE dispone de un servicio de persistencia que se denomina **Freeze**. Freeze facilita el almacenamiento del estado de un objeto en una base de datos, de forma que el desarrollador define en Slice el estado almacenado para los objetos y el compilador Freeze genera código que permite almacenar y recuperar el estado de los objetos en y desde una base de datos, respectivamente. Por defecto, ICE utiliza *Berkeley DB* y su base de datos asociada.

ICE también proporciona herramientas que facilitan el mantenimiento de bases de datos y la migración del contenido de las bases de datos existentes a un nuevo esquema si las definiciones de los objetos cambian.

2.2. Servicios ofrecidos por ICE

El núcleo de ICE proporciona una sofisticada plataforma cliente-servidor para el desarrollo de aplicaciones distribuidas. Sin embargo, las aplicaciones reales necesitan normalmente algo más que las capacidades remotas, como por ejemplo la activación de servidores bajo demanda, la distribución de proxies a los clientes, la distribución de eventos asíncronos, la configuración de aplicaciones, etc.

ICE maneja una serie de servicios que gestionan estas características y otras. Dichos servicios se implementan como servidores ICE de forma que la aplicación desarrollada actúe como un cliente. Ninguno de estos servicios utilizan características internas a ICE ocultas a la aplicación en cuestión, por lo que en teoría es posible desarrollar servicios equivalentes por parte del desarrollador. Sin embargo, manteniendo estos servicios disponibles como parte de la plataforma permite al desarrollador centrarse en el desarrollo de la aplicación en lugar de construir la infraestructura necesaria en primer lugar. Además, la construcción de dichos

servicios no es un esfuerzo trivial, por lo que es aconsejable conocer y usar los servicios disponibles en lugar de reinventar la rueda en cada aplicación.

2.2.1. IceGrid

IceGrid es la implementación de un **servicio de localización ICE** que transforma la información simbólica en un proxy indirecto asociado a una pareja protocolo-dirección en lo que se refiere a resolución indirecta. Además de ser un servicio de localización, IceGrid tiene muchas otras características:

- IceGrid permite el registro de servidores para un arranque automático, es decir, habilita la activación de servidores bajo demanda (servidores que se activan cuando el cliente emite una solicitud).
- IceGrid proporciona herramientas que facilitan la configuración de aplicaciones complejas que contienen ciertos servidores.
- IceGrid soporta la replicación y el balanceado de carga.
- IceGrid automatiza la distribución de los ejecutables asociados a los servidores y de sus archivos vinculados.
- IceGrid proporciona un servicio que permite a los clientes obtener proxies para los objetos en los que están interesados.

2.2.2. IceBox

IceBox es un único **servidor de aplicaciones** que permite gestionar el arranque y la parada de un determinado número de componentes de aplicación. Dichos componentes se pueden desplegar como una biblioteca dinámica en lugar de un proceso. Esto hecho reduce la sobrecarga global del sistema, por ejemplo, permitiendo ejecutar varios componentes en una única máquina virtual Java en lugar de tener múltiples procesos, cada uno con su propia máquina virtual.

2.2.3. IceStorm

IceStorm es un **servicio de publicación-subscripción** que actúa como un distribuidor de eventos entre servidores y clientes. Los publicadores envían eventos al servicio IceBox, el cual los notifica a los subscriptores. De esta forma, un único evento publicado por el publicador se puede enviar a múltiples subscriptores. Los eventos están categorizados en función de un tema, y los subscriptores especifican los temas en los que están interesados. IceBox permite seleccionar entre distintos criterios de calidad de servicio para que las aplicaciones puedan obtener una relación fiabilidad-rendimiento apropiada.

2.2.4. IcePatch2

IcePatch2 es un servicio que permite la fácil **distribución de actualizaciones de software** a los clientes. Éstos simplemente han de conectarse al servidor IcePatch2 y solicitar actualizaciones para una determinada aplicación. El servicio chequea automáticamente la versión de los clientes y envía cualquier actualización disponible en un formato comprimido para gestionar eficientemente el ancho de banda. Estas actualizaciones pueden ser seguras utilizando el servicio Glacier2, de forma que sólo los clientes autorizados puedan descargarse actualizaciones.

2.2.5. Glacier2

Glacier2 es el **servicio de cortafuegos de ICE**, el cual permite que tanto los clientes como los servidores se comuniquen de forma segura a través de un cortafuegos sin comprometer la seguridad. El tráfico entre el cliente y el servidor queda completamente encriptado utilizando certificados de clave pública y es bidireccional. Glacier2 también proporciona soporte para la autenticación mutua y para la gestión segura de sesiones.

2.3. Beneficios arquitectónicos de ICE

La arquitectura propuesta por ICE proporciona un serie de beneficios a los desarrolladores de aplicaciones:

- Mantiene una semántica orientada a objetos.
- Proporciona un manejo síncrono y asíncrono de mensajes.
- Soporta múltiples interfaces.
- Es independiente de la máquina.
- Es independiente del lenguaje de programación.
- Es independiente de la implementación, es decir, el cliente no necesita conocer cómo el servidor implementa sus objetos.
- Es independiente del sistema operativo.
- Soporta el manejo de hilos.
- Es independiente del protocolo de transporte empleado.
- Mantiene transparencia en lo que a localización y servicios se refiere (IceGrid).
- Es seguro (SSL y Glacier2).
- Permite hacer persistentes las aplicaciones (Freeze).
- Tiene disponible el código fuente.

Capítulo 3

Un ejemplo sencillo de aplicación con ICE

3.1. ¡Hola mundo! con ICE

En esta sección se estudiará como crear una sencilla aplicación cliente-servidor utilizando los lenguajes Python, C++, Java, y Ruby para la parte cliente. Dicha aplicación en cuestión será una versión remota del clásico *¡Hola mundo!*. El objetivo de esta sección es establecer una primera toma de contacto en lo que a desarrollo con ICE se refiere.

3.1.1. Escribiendo la definición en Slice

El primer paso en lo que al desarrollo de aplicaciones ICE se refiere consiste en la definición de un fichero que especifique las interfaces utilizadas por la aplicación. Para el sencillo ejemplo que se presenta a continuación, dicha definición sería la siguiente:

```
module Demo {
```

```
   interface HolaMundo {
      void saludar ();
   };
};
```

La definición en Slice consiste en un módulo *Demo*, el cual contiene una única interfaz denominada *HolaMundo*. Dicha interfaz es muy simple y sólo proporciona una operación: *saludar*. Esta operación no acepta parámetros de entrada y no devuelve ningún valor.

3.1.2. Escribiendo la aplicación en Python

Esta sección muestra cómo crear la aplicación *¡Hola Mundo!* en Python.

3.1.2.1. Escribiendo el servidor

El servidor, cuyo código se explicará a continuación es el siguiente:

```
import sys, traceback, Ice
Ice.loadSlice('../slice/holaMundo.ice', ['-I' '/usr/share/slice'])
import Demo

class HolaMundoI (Demo.HolaMundo):
    def saludar (self, current = None):
        print '¡Hola Mundo!'

class Server (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        adapter = self.communicator().createObjectAdapterWithEndpoints(
            'HolaMundoAdapter', 'default -p 10000')
        adapter.add(HolaMundoI(), Ice.stringToIdentity('HolaMundo'))
        adapter.activate()
        self.communicator().waitForShutdown()
        return 0

Server().main(sys.argv)
```

Como se puede apreciar, el código es muy sencillo gracias a la potencia que nos ofrece Python. De hecho, sólo necesitaremos utilizar las mínimas instrucciones de cara a poner nuestra aplicación en funcionamiento.

La primera parte del programa está vinculada a la importanción de módulos y a la carga del archivo en el que previamente definimos nuestras especificaciones en Slice. Es importante resaltar que es necesario incluir el módulo *Ice* en todas nuestras aplicaciones ICE. Así mismo,

merece la pena hablar sobre la operación *loadSlice*, la cual nos permite generar el código asociado al archivo definido en Slice de forma dinámica, es decir, permitiendo que los archivos Slice se carguen en tiempo de ejecución y se traduzcan dinámicamente en código Python, el cual es inmediatamente compilado y queda disponible para la aplicación del usuario. En el caso específico del cliente, la interfaz *HolaMundo* se traduce en una clase que representa el esqueleto asociado a dicha interfaz. Como se verá posteriormente, su hómologo en la parte del cliente está representado por el proxy.

A continuación se define la clase *HolaMundoI*, la cual es una especialización de la clase *Demo.HolaMundo*, que a su vez representa al esqueleto generado por ICE a partir de la interfaz *HolaMundo*. Ahora simplemente se han de implementar las operaciones definidas en dicha interfaz. En este caso, se propone una implementación de la operación *saludar*, la cual imprime por pantalla un mensaje.

El siguiente paso consiste en escribir el código asociado a la aplicación servidora en cuestión. Para ello, y como se observa en el código expuesto, se define la clase *Server*. Esta clase contiene todo el código asociado a la instanciación de los recursos necesarios para obtener la parte servidora de la aplicación. En primer lugar, se observa cómo la clase *Server* hereda de la clase *Ice.Application*, proporcionada por ICE para encapsular todo el código asociado a las actividades de inicialización y finalización. El idea de esta clase es que el desarrollador cree una especialización de la misma y que implemente el método abstracto *run* en la clase derivada. El aspecto que se obtiene al utilizar esta clase es el siguiente:

```
class Server (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
....
# Server code.
....
        self.communicator().waitForShutdown()
        return 0

Server().main(sys.argv)
```

Utilizar la clase *Ice.Application* hace más sencilla la creación de aplicaciones ICE.

Finalmente, quedan por ejecutar tres instrucciones para completar la parte servidora:

1. Crear un adaptador de objetos: para ello se utiliza la operación *createObjectAdapterWithEndpoints*, especificando el nombre del adaptador de objetos (*HolaMundoAdapter*) y

la dirección en la que escuchará dicho adaptador de objetos (*default -p 10000*). En este caso, dicha dirección utiliza el protocolo TCP/IP por defecto y el puerto 10000.

2. Informar al adaptador de objetos de la existencia de un nuevo sirviente: para ello se utiliza la operación *add*, especificando la instancia específica del sirviente (*HolaMundoI()*) y el identificador asociado a dicha instancia (*Ice.stringToIdentity('HolaMundo')*). En este caso, la cadena *HolaMundo* es el nombre del sirviente.

3. Activar el adaptador de objetos: para ello se utiliza la operación *activate*.

Ya sólo se necesita instanciar la clase *Server* y el servidor quedará a la escucha para atender las peticiones de los clientes.

Llegados a este punto, es importante resaltar que el código comentado anteriormente es esencialmente el mismo para todos los servidores. Así mismo, se puede comprobar que Python minimiza la cantidad de código necesaria para el desarrollo de aplicaciones ICE, lo cual lo hace especialmente útil para temas de prototipado.

3.1.2.2. Escribiendo el cliente

La parte asociada al cliente, la cual es más sencilla aún si cabe que la del servidor y que se explicará a continuación, se expone a continuación:

```
import sys, traceback, Ice
Ice.loadSlice('../slice/holaMundo.ice', ['-I' '/usr/share/slice'])
import Demo

class Client (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        basePrx = self.communicator().stringToProxy('
                              HolaMundo:default -p 10000')
        holaMundoPrx = Demo.HolaMundoPrx.checkedCast(basePrx)
        holaMundoPrx.saludar()
        self.communicator().waitForShutdown()
        return 0

Client().main(sys.argv)
```

El código del cliente es idéntico al del servidor a excepción del código asociado al método *run*. En este caso, la aplicación sólo ha de crear un proxy al objeto registrado en la parte del

servidor y ejecutar la operación *saludar* en dicho proxy, ya que éste actúa como embajador de dicho objeto. Para ello se han de llevar a cabo los siguientes pasos:

1. Obtener un proxy al objeto remoto: para ello se utiliza la operación *stringToProxy*, pasándole como parámetro la cadena *'HolaMundo:default -p 10000'*, la cual contiene la identidad del objeto y el puerto usado por el servidor.

2. Realizar una conversión del proxy obtenido en el paso anterior (el cual es del tipo *Ice::ObjectPrx*) a un proxy del tipo *Demo::HolaMundoPrx*: para ello se utiliza la operación *Demo.HolaMundoPrx.checkedCast*, la cual envía un mensaje al servidor para comprobar que realmente está asociado a la interfaz *Demo::HolaMundo*. En ese caso, la llamada devuelve un proxy del tipo *Demo.HolaMundoPrx*. En caso contrario, la llamada devuelve *None*.

3.1.3. Escribiendo la aplicación en C++

El sencillo ejemplo *¡Hola Mundo!* es prácticamente idéntico a su homólogo en Python, pero con la salvedad del cambio de sintaxis asociado.

3.1.3.1. Escribiendo el servidor

El servidor, cuyo código se explicará a continuación es el siguiente:

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

using namespace std;
using namespace Demo;

class HolaMundoI : public HolaMundo {
public:
  virtual void saludar (const Ice::Current&);
};

void
HolaMundoI::
saludar (const Ice::Current&)
{
  cout << "¡Hola Mundo!" << endl;
}
```

```
class Server : virtual public Ice::Application {
  virtual int run (int, char*[]) {
    Ice::ObjectAdapterPtr adapter = communicator()->
      createObjectAdapterWithEndpoints("HolaMundoAdapter",
                                       "default -p 10000");
    Ice::ObjectPtr object = new HolaMundoI;
    adapter->add(object, communicator()->stringToIdentity("HolaMundo"));
    adapter->activate();
    communicator()->waitForShutdown();
    return 0;
  }
};

int main (int argc, char *argv[]) {
  Server server;
  return server.main(argc, argv);
}
```

Como se puede apreciar, la funcionalidad del código mostrado es la misma que para el lenguaje Python.

En primer lugar, se incluyen los archivos de cabecera necesarios para hacer funcionar la aplicación servidora. Estos son el archivo *Ice/Application.h*, para hacer uso de la funcionalidad definida en la clase *Application*, y el archivo *HolaMundo.h*, el cual se genera a partir de compilar el archivo *HolaMundo.ice* para el lenguaje C++. Posteriormente, se importan los contenidos de los espacios de nombrado *std* y *Demo*.

A continuación, se implementa la clase *HolaMundoI*, la cual deriva de la clase *HolaMundo* y, que a su vez, fue generada a partir de las definiciones Slice del archivo *HolaMundo.ice*. El resto del código es similar que para el ejemplo en Python:

1. Se crea el adaptador de objetos.

2. Se informa al adaptador de objetos de la existencia del sirviente.

3. Se activa el adaptador de objetos.

3.1.3.2. Escribiendo el cliente

La parte cliente es análoga a la definido con el lenguaje Python y se expone a continuación:

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

using namespace std;
using namespace Demo;

class Client : virtual public Ice::Application {
  virtual int run (int, char*[]) {
    Ice::ObjectPrx base = communicator()->
      stringToProxy("HolaMundo:default -p 10000");
    HolaMundoPrx holaMundoPrx = HolaMundoPrx::checkedCast(base);
    holaMundoPrx->saludar();
    communicator()->waitForShutdown();
    return 0;
  }
};

int main (int argc, char *argv[]) {
  Client client;
  return client.main(argc, argv);
}
```

3.1.4. Escribiendo la aplicación en Java

El primer paso para crear la aplicación en Java consiste en compilar la definión Slice para generar los proxies y esqueletos asociados. Para ello, se pueden ejecutar los siguientes comandos:

```
mkdir generated
slice2java --output-dir generated HolaMundo.ice
```

El siguiente paso es implementar la interfaz definida para el ejemplo básico:

```
public class HolaMundoI extends Demo._HolaMundoDisp {
    public void printString (String s, Ice.Current current) {
System.out.println("¡Hola Mundo!");
    }
}
```

3.1.4.1. Escribiendo el servidor

A continuación se presenta el código del servidor:

```
public class Server extends Ice.Application {
    public int run (String[] args) {
Ice.ObjectAdapter adapter = communicator().
    createObjectAdapterWithEndpoints("HolaMundoAdapter", "default -p 10000");
```

```
Ice.Object object = new HolaMundoI();
adapter.add(object, Ice.Util.stringToIdentity("HolaMundo"));
adapter.activate();
communicator().waitForShutdown();
return 0;
    }
}
```

3.1.4.2. Escribiendo el cliente

A continuación se presenta el código del cliente:

```
public class Client extends Ice.Application {
    public int run (String[] args) {
Ice.Objectprx base = communicator().stringToProxy(
            "HolaMundo:default -p 10000");
Demo.HolaMundoPrx prx = Demo.HolaMundoPrxHelper.
    checkedCast(base);
if (prx == null)
    throw new Error("Proxy no válido");
prx.saludar();
communicator().waitForShutdown();
return 0;
    }
}
```

Como se puede apreciar, tanto el cliente como el servidor son idénticos a los especificados en otros lenguajes.

3.1.5. Escribiendo la parte cliente de la aplicación en Ruby

La parte cliente de la aplicación escrita en Ruby es prácticamente idéntica a la definida en Python:

```
require ''Ice''
Ice::loadSlice(''../slice/HolaMundo.ice'')

class Client < Ice::Application
  def run (args)
    comm = Ice::Application::communicator()
    basePrx = comm.stringToProxy(
        ''HolaMundo:default -p 10000'')
    holaMundoPrx = HolaMundoPrx.checkedCast(basePrx)
    holaMundoPrx.saludar()
    return 0
  end
end
```

```
app = Client.new()
exit(app.main(ARGV))
```

Parte II

Slice

Capítulo 4

El lenguaje Slice

4.1. Introducción

Slice (*Specification Language for Ice*) es el mecanismo de abstracción fundamental para separar las interfaces de los objetos de su implementación. Slice establece un contrato entre el cliente y el servidor que describe los tipos y las interfaces de los objetos utilizados por la aplicación. Esta descripción es independiente del lenguaje de implementación, por lo que no importa que el cliente esté implementado utilizando el mismo lenguaje que en el servidor.

Las definiciones en Slice se compilan para un determinado lenguaje de implementación a través de un compilador. Dicho compilador se encarga de traducir las definiciones inde-

pendiente del lenguaje en definiciones de tipos específicas del lenguaje y en APIs. Son estos tipos y estas definiciones los que serán utilizados por el desarrollador de cara a proporcionar la funcionalidad de la aplicación y a interactuar con ICE. Los algoritmos de traducción para los distintos lenguajes de implementación se conocen como *mappings* de lenguajes. Actualmente, ICE define *mappings* para C++, Java, C#, Visual Basic .NET, Python, PHP, y Ruby (estos dos últimos en lo que a la parte del cliente se refiere).

4.2. Aspectos básicos

4.2.1. Compilación

El compilador de Slice produce archivos fuente que han de ser combinados con el código de la aplicación para generar los ejecutables asociados al cliente y al servidor.

En la figura 4.1 se muestra la situación en la que tanto el cliente como el servidor están implementados utilizando C++. El compilador Slice genera dos archivos a partir de una definición Slice en un archivo fuente *Printer.ice*: un fichero de cabecera (*Printer.h*) y un fichero fuente (*Printer.cpp*).

- El fichero de cabecera *Printer.h* contiene definiciones que se corresponden a los tipos utilizados en la definición Slice. Este archivo se incluye tanto en el código del cliente como en el del servidor para asegurar que ambos están de acuerdo con los tipos y las interfaces utilizadas por la aplicación.

- El fichero fuente *Printer.cpp* proporciona un API al cliente para enviar los mensajes a los objetos remotos. El código fuente del cliente (*Client.cpp*) contiene la lógica de la aplicación del lado del cliente. El código fuente generado y el código del cliente se compilan y se enlazan para obtener el ejecutable asociado al cliente.

Por otra parte, el fichero fuente asociado al servidor (*Server.cpp*) contiene la lógica de la aplicación de la parte del servidor (la implementación de los objetos, es decir, los sirvientes). El proceso de compilación y enlazado es homólogo al de la parte del cliente.

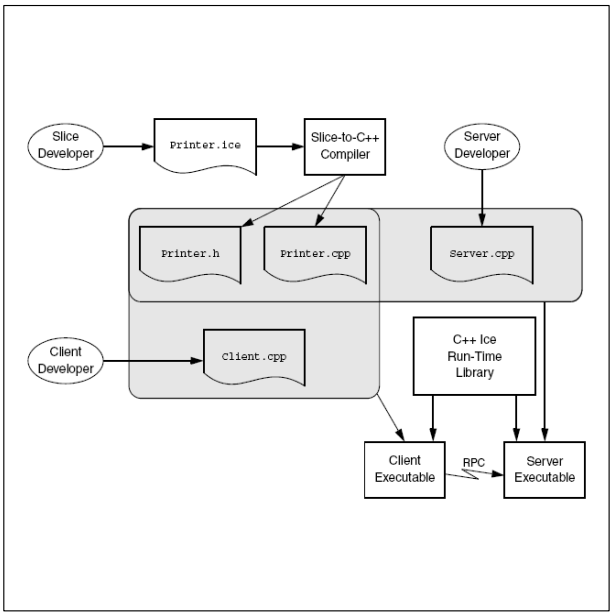

Figura 4.1: Proceso de desarrollo si el cliente y el servidor comparten el mismo lenguaje de programación

El proceso asociado a la utilización de distintos lenguajes de programación en el lado del cliente y en el del servidor es prácticamente idéntico al comentado anteriormente. En la figura 4.2 se muestra un ejemplo.

4.2.2. Ficheros fuente

Los archivos que contiene definiciones Slice han de terminar con la extensión *.ice*, como por ejemplo *Printer.ice*.

Slice es un lenguaje libre de forma, es decir, se pueden utilizar espacios, tabulaciones verticales y horizontales, o retornos de carro para formatear el código de la manera deseada.

Por otra parte, Slice es preprocesado por el preprocesador C++, de forma que se pueden utilizar las directivas de preprocesado comunes, como por ejemplo *#include*. Sin embargo, estas sentencias han de indicarse al principio del archivo, justo antes de cualquier definición Slice.

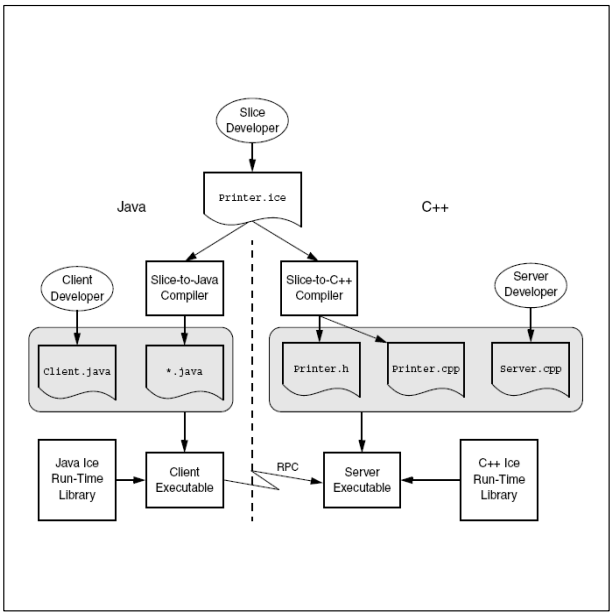

Figura 4.2: Proceso de desarrollo si el cliente y el servidor no comparten el mismo lenguaje de programación

4.2.3. Reglas léxicas

Las reglas léxicas de Slice son muy parecidas a las de los lenguajes C++ y Java, a excepción de algunas diferencias en lo que a identificadores se refiere.

Los comentarios permitidos por Slice son de los tipos utilizados en C y en C++:

```
/*
 * Comentario estilo C.
*/

// Comentario estilo C++.
```

Los identificadores han de comenzar por un carácter alfabético seguido de cualquier secuencia de caracteres alfanuméricos. Los identificadores Slice están restringidos al rango de caracteres alfabéticos ASCII. Por otra parte, Slice no permite el uso de guiones de subrayado en los identificadores. Además, no son sensibles a mayúsculas. Para profundizar más sobre las reglas léxicas de Slice ver [4].

4.2.4. Módulos

Un problema común en grandes sistemas son los conflictos de nombres conforme éstos crecen. Slice proporciona el constructor *module* para solventar este problema:

```
module Demo {
  module Client {
    // Definiciones aquí...
  };
  module Server {
    // Definiciones aquí...
  };
};
```

Un módulo pueden contener cualquier construcción definida en Slice, incluso otros módulos. Además, Slice necesita que cualquier definición esté incluida dentro de un módulo, es decir, no se pueden tener definiciones globales.

Los módulos pueden abrirse y cerrarse en cualquier lugar, lo cual resulta muy útil en proyectos grandes, de forma que se puede estructurar el contenido de un módulo en varios archivos fuente. Además, cuando se produce un cambio en uno de estos ficheros, sólo se necesita recompilar el fichero que ha cambiado.

4.2.5. Tipos básicos en Slice

Slice proporciona los siguientes tipos de datos básicos:

- bool
- byte
- short
- int
- long
- float
- double

- string

Para una información más detallada sobre los mismos consultar [4].

4.2.6. Tipos definidos por el usuario

Además de los tipos básicos comentados previamente, Slice proporciona los siguientes tipos de datos definidos por el usuario:

- Enumeraciones.
- Estructuras.
- Secuencias.
- Diccionarios.

Una enumeración en Slice es parecida a la definida en C++:

```
enum Fruta{Pera, Manzana, Naranja};
```

Una estructura en Slice contiene uno o más miembros de un determinado tipo, incluyendo tipos complejos definidos por el usuario:

```
struct Hora {
  short hora;
  short minute;
  short second;
};
```

Una secuencia es un vector de elementos de longitud variable:

```
sequence<Fruta> BandejaDeFruta;
```

Un diccionario es un conjunto de parejas clave-valor:

```
struct Empleado {
  long numero;
  string primerApellido;
  string segundoApellido;
};

dictionary<long, Empleado> Empleados;
```

Aunque los diccionarios se podrían implementar como una secuencia de estructuras, resulta más adecuado utilizar diccionarios por dos motivos principalmente:

- Un diccionario hace explícito el diseño del desarrollador de la aplicación.
- A nivel de lenguaje de programación, las secuencias se implementan como vectores y requieren una búsqueda linear para localizar un elemento. Por otra parte, los diccionarios están implementados de acuerdo a una estructura de datos eficiente en tiempos de búsqueda.

Para una información más detallada sobre los tipos de datos definidos por el usuario consultar [4].

Slice también soporta la definición de constantes a través de uno de los siguientes tipos de datos:

- Un tipo integral (bool, byte, short, int, long, o un tipo enumerado).
- float o double.
- string.

4.2.7. Interfaces, operaciones, y excepciones

El principal objetivo de Slice es la definición de interfaces, por ejemplo:

```
struct Hora {
  short hora;
  short minute;
  short second;
};

interface Reloj {
  Hora obtenerHora();
  void establecerHora(Hora h);
};
```

La definición anterior especifica un tipo de interfaz denominado *Reloj*. Dicha interfaz soporta dos operaciones: *obtenerHora* y *establecerHora*. Los clientes acceden a un objeto que soporta la interfaz *Reloj* invocando una operación en un proxy para dicho objeto: para

obtener la hora actual el cliente invoca a la hora operación *obtenerHora*, y para establecerla invoca a la operación *establecerHora*, especificando un parámetro del tipo *Hora*.

La definición de una operación ha de tener un tipo de retorno y cero o más definiciones de paramétros. Por defecto, los parámetros enviados del cliente al servidor son parámetros de entrada. Para pasar un parámetro del servidor al cliente se pueden emplear los parámetros de salida, utilizando para ello el modificador *out*:

```
void obtenerHora(out Hora h);
```

En cuanto a las operaciones que devuelven varios parámetros de salida existen distintas opciones, como por ejemplo especificar que todos esos parámetros sean de salida o asociar el valor de retorno al parámetro de salida más importante y utilizar el modificador *out* para el resto:

```
bool siguiente(out Registro r);
```

Slice no soporta ningún tipo de sobrecarga de operaciones. Todas las operaciones de una interfaz han de tener un nombre distinto entre sí. Este hecho se debe a que existen ciertos lenguajes que no soportan esta característica.

Existen distintos modificadores que se pueden aplicar a la definición de las operaciones en función de su naturaleza. Algunas operaciones no modifican el estado del objeto sobre el que operan, mientras que otras son idempotentes, es decir, se obtiene el mismo resultado tanto si se invocan una vez como si se invocan varias. El desarrollador puede informar a ICE sobre estos hechos de cara a que éste utilice mecanismos más agresivos con el objetivo de ahorrar recursos y proporcionar una implementación más eficiente. Para ello, se utilizan los modificadores *nonmutating* e *idempotent*, respectivamente:

```
struct Hora {
  short hora;
  short minute;
  short second;
};

interface Reloj {
  nonmutating Hora obtenerHora();
  idempotent void establecerHora(Hora h);
};
```

En lo relativo a excepciones, ICE permite la definición de excepciones de usuario para indicar al cliente las condiciones de error que se produzcan:

```
exception Error {};
exception RangeError {
  Hora errorEnLaHora;
  Hora tiempoMin;
  Hora tiempoMax;
};

interface Reloj {
  nonmutating Hora obtenerHora();
  idempotent void establecerHora(Hora h)
                  throws RangeError, Error;
};
```

Las excepciones en Slice son parecidas a las estructuras en el sentido de que pueden tener un número arbitrario de miembros, pero con la salvedad de que las excepciones pueden no tener miembros. Además, Slice soporta la herencia de excepciones. De hecho, mantiene un árbol de herencia en el que se especifican las excepciones propias al entorno de ejecución de ICE.

Es importante tener en cuenta que una operación sólo puede lanzar una excepción que previamente fue especificada en su definición en Slice. Además, las excepciones no se consideran tipos de datos de primera clase, por lo que no pueden pasarse como parámetros en las operaciones ni utilizarlos como miembros en la definición de estructuras, entre otras restricciones.

Siguiendo con el ejemplo del reloj se podría definir un servidor en el que estuvieran presentas las zonas horarias de todo el mundo:

```
exception ErrorGenerico {
  string razon;
};

struct Hora {
  short hora;
  short minute;
  short second;
};

exception ValorHoraErroneo extends ErrorGenerico {};

interface Reloj {
  nonmutating Hora obtenerHora();
```

```
  idempotent void establecerHora(Hora h) throws ValorHoraErroneo;
};

dictionary<string, Reloj*> MapaZonas;

exception NombreZonaErroneo extends ErrorGenerico {};

interface RelojMundial {
  idempotent void añadirZona(string nombre, Reloj* relojZona);
  void eliminarZona(string nombre) throws NombreZonaErroneo;
  nonmutating Reloj* encontrarZona(string nombre) throws NombreZonaErroneo;
  nonmutating MapaZonas listaZonas();
  idempotent void establecerZonas(MapaZonas zonas);
};
```

La interfaz *RelojMundial* actúa como un gestor de zonas, es decir, un gestor de parejas nombre de la zona y reloj. Dicha interfaz proporciona operaciones para añadir zonas, eliminar zonas, encontrar zonas, obtener la lista completa de zonas, y establecer la lista de zonas. El concepto más importante de este ejemplo reside en que las propias interfaces son tipos de datos por sí mismas y, por lo tanto, se pueden pasar como parámetros y pueden ser devueltas, como se aprecia en las operaciones *añadirZona* y *eliminarZona*, respectivamente. El operador se conoce como operador proxy. Su argumento más a la izquierda ha de ser una interfaz (o una clase), y su tipo de retorno es un proxy. Un proxy se puede entender como un puntero que referencia a un objeto. La semántica de los proxies es muy parecida a los punteros asociados a una instancia de clase en C++.

Cuando un cliente pasa un proxy *Reloj* a la operación *añadirZona*, el proxy hace referencia al objeto de tipo *Reloj* real en el servidor. El objeto ICE *Reloj* referenciado por ese proxy puede estar implementado en el mismo proceso servidor que la interfaz *RelojMundial* o en un proceso de servidor distinto. Desde el punto de vista del cliente, la localización física de dicho objeto no importa. En otras palabras, el proxy actúa como el embajador local del objeto remoto: invocar una operación en el proxy se traduce en invocar la operación en la implementación del objeto real. Si el objeto está implementado en un espacio de direcciones distinto al del cliente, ICE realiza una llamada a procedimiento remoto. Si el objeto está implementado en el mismo espacio de direcciones, ICE realiza una llamada local desde el proxy a la implementación del objeto.

Al igual que ocurre con las excepciones, las interfaces también soportan el concepto de

herencia.

4.3. Aspectos avanzados

En esta sección se comentarán algunos aspectos avanzados de Slice. Sin embargo, para obtener una información más completa y detallada se recomienda estudiar [4].

4.3.1. Clases

Además de las interfaces, Slice permite la definición de clases. Las clases son como las interfaces en el sentido de que tienen operaciones, y se parecen a las estructuras debido a que pueden incluir atributos. El resultado son unos objetos híbridos que pueden tratarse al igual que las interfaces y pasarse por referencia, o pueden tratarse como atributos y pasarse por valor. El beneficio de las clases es una mayor flexibilidad a la hora de desarrollar aplicaciones: las clases permiten modelar el comportamiento a implementar en el lado del cliente y las interfaces permiten representar el comportamiento a implementar en el lado del servidor. Las clases soportan la herencia y, por lo tanto, el polimorfismo.

El utilizar clases en ciertos contextos, como por ejemplo definir una clase con operaciones, implica reflexionar sobre ciertos aspectos asociados a la arquitectura de la aplicación. El tener una clase con operaciones definidas en ellas implica utilizar código nativo del lado del cliente y, por lo tanto, elimina la transparencia de implementación proporcionada por las interfaces.

Una clase Slice también puede utilizarse como un sirviente en un servidor, es decir, una instancia de una clase se puede utilizar para proporcionar el comportamiento de un interfaz:

```
interface Tiempo {
  nonmutating Hora obtenerHora();
  idempotent void establecerHora(Hora h);
};

class Reloj implements Tiempo {
  Hora hora;
};
```

Parte III

ICE avanzado

Capítulo 5

Propiedades de ICE y configuración

5.1. Introducción

ICE utiliza un mecanismo de configuración que permite controlar muchos aspectos del comportamiento de las aplicaciones ICE en tiempo de ejecución, como el tamaño máximo de los mensajes, el número de hilos, o si generar mensajes de trazas de red. Dicho mecanismo de configuración no sólo es útil para configurar ICE, sino que también se puede utilizar para proporcionar parámetros de configuración a las propias aplicaciones. Este mecanismo es muy simple en el sentido de que se puede utilizar con una sencilla API, y escalable en el sentido de que se adapta a la necesidades de la mayoría de aplicaciones.

5.2. Propiedades

ICE y sus distintos subsistemas se configuran a través de las *propiedades*. Una propiedad es un par nombre-valor, por ejemplo:

```
Ice.UDP.SndSize=65535
```

En este ejemplo, el nombre de la propiedad es *Ice.UPD.SndSize* y su valor es *65535*.

5.3. Archivos de configuración

Los valores de las propiedades se establecen normalmente en archivos de configuración. Un archivo de configuración contiene un determinado número de parejas nombre-valor, cada uno en una línea del mismo. El carácter # se utiliza para introducir comentarios que se extienden para la línea actual en la que se introduce dicho carácter. Un ejemplo es el siguiente:

```
# Ejemplo de archivo de configuración para ICE.

Ice.MessageSizeMax = 2048 # Tamaño máximo de 2MB.
Ice.Trace.Network = 3     # Máximo nivel de traza de red.
Ice.Trace.Protocol =      # Desactivada la traza de protocolo.
```

Para C++, Python, y .NET, ICE lee los contenidos del archivo de configuración cuando se crea el *communicator*. Por defecto, el nombre del archivo de configuración se determina leyendo el contenido de la variable de entorno ICE_CONFIG.

5.4. La propiedad Ice.Config

La propiedad *Ice.Config* tiene un significado especial para ICE: determina la ruta del archivo de configuración a partir del cual leerá los valores de las distintas propiedades. Por ejemplo:

```
$ ./server --Ice.Config=/usr/local/holaMundo/config
```

Para aspectos más detallados como acceder y establecer los valores de las propiedades desde una aplicación se recomienda consultar [4].

Capítulo 6

Programación asíncrona

6.1. Introducción

La mayoría de las tecnologías *middleware* más modernas tratan de facilitar la transición de los programadores al desarrollo de aplicaciones distribuidas. Para ello, lo que hacen es acercar el modelo de programación tradicional a los entornos distribuidos, de forma que las llamadas a procedimientos remotos sean prácticamente idénticas a las llamadas locales. Sin

embargo, la implementación de un objeto en una aplicación distribuida puede estar implementado en otra máquina distinta a la que invoca una operación suya, por lo que existen ciertas diferencias semánticas que el programador ha de tener en cuenta, como por ejemplo la sobrecarga derivada de la invocación a operaciones remotas o los errores derivados de la propia red. A pesar de estas cuestiones, la experiencia del programador con la programación orientada a objetos es aún relevante, y este modelo de programación *síncrona*, en el que el hilo que invoca una operación se queda bloqueado hasta que ésta finaliza, es familiar y fácil de entender.

ICE es inherentemente un plataforma *middleware* asíncrona que simula el comportamiento síncrono para el beneficio de las aplicaciones (y de sus programadores). Cuando una aplicación ICE ejecuta una operación síncrona en un proxy vinculado a un objeto remoto, los parámetros de entrada de la operación se codifican junto con el mensaje a transportar, y el hilo de ejecución que lleva a cabo esa invocación se bloquea con el objetivo de simular la llamada síncrona. Mientras tanto, ICE opera en un segundo plano procesando los mensajes hasta que se recibe la respuesta y el hilo que ejecutó la llamada queda bloqueado para decodificar los resultados.

Existen muchas situaciones en las que la naturaleza bloqueante de la programación síncrona resulta muy restrictiva. Por ejemplo, la aplicación puede disponer de trabajo útil para ejecutar mientras está a la espera de una invocación remota. De esta forma, utilizar una invocación síncrona fuerza a la aplicación a posponer ese trabajo hasta que se reciba la respuesta asociada o a desarrollarlo en otro hilo de ejecución distinto. Cuando ninguna de estas alternativas resulta aceptable, la facilidades asíncronas proporcionadas por ICE suponen una solución eficaz para mejorar el rendimiento y la escalabilidad, o para simplificar ciertas tareas complejas.

6.1.1. Invocación de métodos asíncrona o AMI (*Asynchronous Method Invocation*)

AMI es el término empleado para describir el soporte del lado del cliente en el modelo de programación asíncrono. Utilizando AMI, una invocación remota no bloquea el hilo de ejecución asociado mientras se espera la respuesta asociada a dicha invocación. En su lugar,

dicho hilo puede continuar sus actividades y ICE notifica a la aplicación cuando la respuesta llega. Esta notificación se efectúa a través de una retrollamada de un objeto proporcionado por la aplicación desarrollada.

6.1.2. Tratamiento de métodos asíncrono o AMD (*Asynchronous Method Dispatch*)

El número de solicitudes síncronas simultáneas que un servidor es capaz de soportar queda determinado por el modelo de concurrencia del servidor. Si todos los hilos están ocupados atendiendo a operaciones que requieren mucho tiempo, entonces no existen hilos disponibles con los que procesar nuevas peticiones y, por lo tanto, es posible que los clientes experimenten una inaceptable pérdida de respuesta.

AMD, el equivalente en el lado del servidor de AMI, trata este problema de escalabilidad. Utilizando AMD, un servidor es capaz de recibir una solicitud y entonces suspender su procesamiento para liberar el hilo asociado tan pronto como sea posible. Cuando dicho procesamiento se reanuda y los resultados están disponibles, el servidor envía una respuesta explícita utilizando un objeto de retrollamada proporcionado por el núcleo de ejecución de ICE.

En términos prácticos, una operación AMD típicamente encola los datos de la solicitud (como por ejemplo el objeto de retrollamada y los argumentos de la operación) para procesarlos posteriormente por un hilo de la aplicación (o un conjunto de hilos). De esta forma, el servidor minimiza el número de hilos asociados al tratamiento de operaciones y es capaz de soportar miles de clientes de manera simultánea y eficiente.

Un caso de uso alternativo para AMD es una operación que requiere una mayor cantidad de procesamiento después de completar una solicitud del cliente. Con el objetivo de minimizar el tiempo de espera del cliente, la operación devuelve los resultados mientras en el hilo asociado al tratamiento de dicha solicitud espera, y entonces continúa utilizando dicho hilo para el trabajo adicional.

6.1.3. Controlando la generación de código utilizando metadatos

Para indicar el uso de un modelo de programación asíncrono (AMI, AMD, o ambos) se utilizan metadatos en las definiciones Slice. El programador puede especificar estos metadatos a dos niveles: a nivel de interfaz o de clase, o a nivel de operación individual. Si se especifica para una interfaz o una clase, entonces el soporte asíncrono se genera para todas sus operaciones. De forma alternativa, si el soporte asíncrono se necesitasólo para ciertas operaciones, entonces el código generado puede minimizarse especificando los metadatos sólo para los operaciones que así lo requieran.

Los métodos de invocación síncronos se generan siempre en un proxy, de forma que especificando los metadatos AMI simplemente se añaden los métodos de invocación asíncronos. Por otra parte, especificar los metadatos AMD implica que los métodos de tratamiento síncrono sean reemplazados por sus homólogos asíncronos.

Un ejemplo en Slice es el siguiente:

```
[``ami''] interface I {
  bool esValido();
  float calcularMedia();
};

interface J {
  [``amd''] void comenzarProceso();
  [``ami'', ``amd''] int finalizarProceso();
};
```

En este ejemplo, todos los métodos asociados a la interfaz I se generan con soporte para invocaciones síncronas y asíncronas. En la interfaz J, la operación *comenzarProceso* utiliza un tratamiento asíncrono, mientras que la operación *finalizarProceso* soporta una invocación y un tratamiento asíncronos.

Especificar metadatos a nivel de operación, en lugar de a nivel de interfaz o de clase, no sólo minimiza la cantidad de código generado, sino lo que es más importante, también minimiza la complejidad. Aunque el modelo asíncrono es más flexible, también resulta más complejo de utilizar. Por lo tanto, es decisión del desarrollador limitar el uso del modelo asíncrono para aquellas operaciones en las que proporcione determinadas ventajas, utilizando el modelo síncrono para el resto, el cual resulta más sencillo.

6.1.4. Transparencia

El uso del modelo asíncrono no afecta a lo que se envía por el medio. De hecho, el modelo de invocación utilizado por el cliente es transparente al servidor, mientras que el modelo de tratamiento utilizado por el servidor es transparente para el cliente. Por lo tanto, un servidor es incapaz de distinguir si el modelo de invocación del cliente es síncrono o asíncrono, mientras que el cliente es incapaz de distinguir un tratamiento síncrono o asíncrono del servidor.

6.2. AMI

En esta sección se describe la implementación de ICE para AMI y cómo utilizarla. El primer paso será discutir una forma parcial de cómo implementar AMI utilizando invocaciones en un único sentido. Este estudio tiene como objetivo mostrar los beneficios de AMI. Posteriormente se comentará el *mapping* a Python y algunos ejemplos relacionados.

6.2.1. Simulando AMI utilizando invocaciones *oneway*

Como se comentó anteriormente, las invocaciones síncronas no resultan apropiadas para cierto tipo de aplicaciones. Un ejemplo son las interfaces gráficas de usuario, en las que no resulta aceptable que un usuario quede a la espera de la finalización de una tarea tras haber provocado un evento en la interfaz.

Dicha aplicación podría evitarse utilizando invocaciones en un único sentido, las cuales por definición no pueden devolver un valor de retorno ni manejar parámetros de salida. Debido a que el núcleo de ejecución de ICE no espera respuesta alguna, la invocación bloquea el hilo de ejecución asociado mientras se lleva a cabo la codificación de los parámetros y la copia del mensaje al buffer de transporte local. Sin embargo, el uso de invocaciones en un único sentido puede requerir cambios inaceptables en la definición de las interfaces. Por ejemplo, una invocación en dos sentidos que devuelva valores de retorno o lance excepciones debe convertirse en dos operaciones al menos: una para el cliente de forma que al invocarla utilice una semántica *oneway* y que contenga sólo los parámetros de entrada, y otra (o más) para que el servidor notifique al cliente los resultados pertinentes.

Para ilustrar estos cambios se presenta el siguiente ejemplo:

```
interface I {
  int op (string s, out long l);
};
```

Como se puede apreciar, la operación *op* no es adecuada para invocaciones en un único sentido, ya que tiene un parámetro de salida y especifica un valor de retorno no nulo. Para amoldar la anterior definición a una invocación *oneway* de la operación *op* se podrían efectuar los siguientes cambios:

```
interface IRetroLlamada {
  void resultados (int retorno, long l);
};

interface I {
  void op (IRetroLlamada* rl, string s);
};
```

Los cambios con respecto a la definición original son los siguientes:

- Se ha añadido una interfaz *IRetroLlamada*, la cual contiene la operación resultados cuyos parámetros representan a los resultados de la operación original. El servidor invoca esta operación para notificar al cliente que la operación ha terminado.
- Se ha modificado la operación *I::op* para adecuarse a la semántica *oneway*: ahora tiene un tipo de retorno nulo y sólo toma parámetros de entrada.
- Se ha añadido un parámetro a la interfaz *I::op* que permite al cliente proporcionar un proxy para su objeto de retrollamada.

Como se puede apreciar, se han llevado a cabo cambios substanciales en relación a la definición de la interfaz original para obtener la implementación requerida por el cliente. Ahora el cliente actúa también como servidor, debido a que ha de crear una instancia de *IRetroLlamada* y registrarla en un adaptador de objetos con el objetivo de recibir las notificaciones asociadas a la finalización de operaciones.

Una repercusión más relevante está asociada al impacto de estos cambios sobre el servidor. El hecho de que un cliente invoque a una operación de forma síncrona o asíncrona debería

ser irrelevante para el servidor, pero en este caso se ha generado una gran acoplación entre el servidor y el cliente. De hecho, la situación podría ser incluso peor si se hubiera incluido alguna excepción en la operación inicial.

El hecho de describir esta simulación AMI sirve para explicar la estrategia utilizada por ICE internamente, la cual es bastante similar a la aquí expuesta pero con algunas diferencias:

1. No se requiere cambiar el tipo de sistema para utilizar AMI. La representación de los datos en la red es idéntica y, por lo tanto, clientes y servidores síncronos y asíncronos pueden coexistir en un mismo sistema, utilizando las mismas operaciones.

2. La solución proporcionada por AMI acomoda el manejo de excepciones de una manera razonable.

3. Utilizar AMI no requiere que el cliente actúe también de servidor.

6.2.2. *Mappings*

Una operación para la cual se ha especificado el metadato AMI soporta los modelos de invocación síncrono y asíncrono. Además del método del proxy para la invocación síncrona, el generador de código crea un método para la invocación asíncrona, junto con una clase de retrollamada. El código generado sigue un patrón similar al expuesto en la anterior sección, el cual se apreciará mucho mejor al observar el siguiente caso específico.

6.2.2.1. *Mapping* a Python

Para cada operación AMI, el *mapping* a Python genera un método adicional en el proxy con el mismo nombre de la operación especificada en la interfaz Slice más el sufijo *_async*. Este método tiene un valor de retorno nulo. El primer parámetro es una referencia al objeto de retrollamada, como se describe a continuación. El resto de parámetros hace referencia a los parámetros de entrada de la operación, los cuales se especifican en el orden de declaración definido.

El método asíncrono del proxy requiere que su objeto de retrollamada defina dos operaciones:

- *def ice_response(self, <parámetros>)*: indica que la operación ha finalizado con éxito. Los parámetros representan el valor de retorno y los parámetros de salida de la operación.

- *def ice_exception(self, ex)*: indica que se ha lanzado una excepción de usuario o una excepción local.

Se puede suponer, por ejemplo, la siguiente definición:

```
interface I {
  [``ami''] int op (short s, out long l);
};
```

Los prototipos de las operaciones requeridas para el objeto de retrollamada de la operación *op* son los siguientes:

```
class ...
  #
  # Prototipos de las operaciones:
  #
  # def ice_response(self, _resultado, l)
  # def ice_exception(self, ex)
```

La operación del proxy para la invocación asíncrona de la operación *op* se genera de la siguiente forma:

```
def op_async(self, __cb, s);
```

6.2.3. Ejemplo

Para demostrar el uso de AMI en ICE se define la siguiente interfaz asociada a un sencillo motor de computación:

```
module Demo {
  sequence<float> Row;
  sequence<Row> Grid;

  exception ErrorDeRango {};

  interface Model {
    [``ami''] Grid interpolar(Grid datos, float factor)
         throws ErrorDeRango;
  };
};
```

Dado un elemento de dos dimensiones de valores en punto flotante y un factor, la operación *interpolar* devuelve un nuevo elemento del mismo tamaño con los valores interpolados de una determinada manera. La implementación asociada a la parte del cliente para la invocación asíncrona en Python podría estar basada en el siguiente código:

```
class AMI_Model_interpolarI(object):
  def ice_response(self, resultado):
    print ``Datos recibidos''
    # ... post-procesado ...

  def ice_exception(self, ex):
    try:
      raise ex
    except Demo.ErrorDeRango, e:
      print ``Interpolación fallida: Error de rango.''
    except Ice.LocalException, e:
      print ``Interpolación fallida: '' + str(e)
```

La implementación de la operación *ice_response* informa de un resultado correcto, mientras que la operación *ice_exception* muestra el diagnóstico asociado a la excepción lanzada.

El código para invocar a la operación *interpolar* es muy sencillo:

```
modelo = ...
cb = AMI_Model_interpolarI()
grid = ...
inicializarGrid(grid)
modelo.interpolar_async(cb, grid, 0.5)
```

6.2.4. Cuestiones asociadas a la concurrencia

El conjunto de hilos del cliente es el encargado de habilitar en ICE el soporte para las invocaciones asíncronas. Esos hilos tienen como principal responsabilidad procesar los mensajes de respuesta. Es importante tener en cuenta ciertos factores:

- Un objeto de retrollamada no debe utilizarse para varias invocaciones simultáneas.
- Las llamadas a los objetos de retrollamada se efectúan a través de los hilos de la parte del núcleo de ejecución de ICE asociada al cliente. Por lo tanto, la sincronización puede ser necesaria en el caso de que la aplicación interactúe con el objeto de retrollamada al mismo tiempo que llega la respuesta.

- El número de hijos en el cliente determina el máximo número posible de retrollamadas simultáneas para las invocaciones asíncronas.

6.2.5. *Timeouts*

Soportar el uso de *timeouts* para invocaciones asíncronas presenta un reto especial para ICE. El problema surge debido a que tras una invocación asíncrona, ICE devuelve el control al hilo que efectuó la llamada tan pronto como sea posible.

Consecuentemente, el núcleo de ejecución de ICE ha de utilizar otro mecanismo: un hilo dedicado que, entre otras tareas, revisa periódicamente las invocaciones asíncronas pendientes. De esta forma, si una invocación mantiene un *timeout* que ha espirado, dicho hilo termina la invocación informando con una excepción del tipo *Ice::TimeoutException* a la operación *ice_exception* del objeto de retrollamada asociado a la invocación asíncrona.

Para todo este proceso se puede utilizar la propiedad de configuración *Ice.MonitorConnections*, cuyo valor se especifica en segundos, que determina la frecuencia con la que el hilo dedicado lleva a cabo dichas revisiones.

6.3. AMD

Como se discutió en la sección anterior, los *mappings* para AMI permiten que las aplicaciones puedan utilizar tanto el modelo de invocación síncrono como el modelo de invocación asíncrono. Sin embargo, y en el caso de una operación AMD, la implementación asociada a este modelo de tratamiento asíncrono en el servidor no permite utilizar los dos modelos. De hecho, especificar el metadato AMD en una operación implica que dicha operación, la cual mantenía un modelo de tratamiento síncrono, pase a utilizar un modelo de tratamiento asíncrono.

La operación de tratamiento asíncrono tiene un prototipo similar al utilizado en AMI: el tipo de retorno es nulo, y los argumentos consisten en un objeto de retrollamada y en los parámetros de entrada de la operación. En AMI, el objeto de retrollamada lo proporciona la aplicación, mientras que en AMD dicho objeto lo proporciona el núcleo de ejecución de

ICE, el cual proporciona los métodos necesarios para devolver los parámetros asociados a la operación y lanzar las excepciones correspondientes en su caso.

6.3.1. *Mappings*

6.3.1.1. *Mapping* a Python

Para cada operación AMD, el *mapping* a Python genera un método de tratamiento con el mismo nombre que la operación y con el sufijo *_async*. Este método devuelve *None*. El primer parámetro es la referencia al objeto de retrollamada, como se mostrará posteriormente. El resto de parámetros hacen referencia a los parámetros de entrada de la operación, siguiendo el orden en el que se declararon.

El objeto de retrollamada define dos operaciones:

- *def ice_response(self, <parámetros>)*: indica que la operación ha finalizado con éxito. Los parámetros representan el valor de retorno y los parámetros de salida de la operación.
- *def ice_exception(self, ex)*: indica que se ha lanzado una excepción de usuario o una excepción local.

Se puede suponer, por ejemplo, la siguiente definición:

```
interface I {
  [``amd''] int op (short s, out long l);
};
```

Los prototipos de las operaciones requeridas para el objeto de retrollamada de la operación *op* son los siguientes:

```
class ...
  #
  # Prototipos de las operaciones:
  #
  # def ice_response(self, _resultado, l)
  # def ice_exception(self, ex)
```

El método de tratamiento para la invocación asíncrona de la operación *op* es el siguiente:

```
def op_async(self, __cb, s);
```

6.3.2. Ejemplo

Para demostrar el uso de AMD en ICE se continúa con el ejemplo comentado anteriormente, pero primero es necesario añadir a la operación el metadato AMD:

```
module Demo {
  sequence<float> Row;
  sequence<Row> Grid;

  exception ErrorDeRango {};

  interface Model {
    [``ami'', ``amd''] Grid interpolar(Grid datos, float factor)
        throws ErrorDeRango;
  };
};
```

La implementación asociada en Python tendría una clase sirviente que derivaría de *Demo.Model* y que proporcionaría una definición para el método *interpolar_async* que crearía un *Trabajo* para gestionar el objeto de retrollamada y los argumentos, y añadiría dicho *Trabajo* a una cola. El método utiliza un candado para gestionar el acceso concurrente a la cola:

```
class ModelI(Demo.Model):
  def __init__(self):
    self._mutex = threading.Lock()
    self._jobs = []

  def interpolate_async(self, cb, data, factor, current=None):
    self._mutex.acquire()
    try:
      self._jobs.append(Job(cb, data, factor))
    finally:
      self._mutex.release()
```

Una vez que se ha almacenado la información en una cola, la operación devuelve el control al núcleo de ejecución de ICE, de forma que el hilo de tratamiento asociado quede disponible para procesar otra solicitud. Un hilo de la aplicación elimina el siguiente *Trabajo* de la cola e invoca a la operación *ejecutar*, la cual utiliza a *interpolarGrid* para llevar a cabo el trabajo computacional asociado:

```
class Trabajo(object):
  def __init__(self, cb, grid, factor):
    self._cb = cb
    self._grid = grid
    self._factor = factor
```

```
def execute(self):
  if not self.interpolateGrid():
    self._cb.ice_exception(Demo.ErrorDeRango())
    return
  self._cb.ice_response(self.grid)

def interpolarGrid(self):
  # ...
```

Si *interpolarGrid* devuelve falso, entonces se invoca a la operación *ice_exception* para indicar que ha ocurrido un error de rango. La sentencia *return* que sigue a la llamada *ice_exception* es necesaria porque esta llamada no arroja una excepción, sino que simplemente codifica el argumento de la excepción y se lo envía la cliente.

Si la interpolación tuvo éxito, se invoca a la operación *ice_response* para enviar los datos modificados al cliente.

Capítulo 7

Transferencia eficiente de ficheros

7.1. Introducción

La transferecia de ficheros entre una aplicación cliente y una servidora puede ser tratada de distintas formas según la naturaleza de la aplicación. Por ejemplo, si el cliente tiene que obtener un fichero de un tamaño relativamente grande parece razonable que transferir todo el contenido del mismo en una única llamada remota no es una alternativa viable. Además, dicha transferencia puede manejarse de una forma más eficiente utilizando ciertas características que ICE proporciona. En Slice tendríamos la siguiente definición:

```
#include <Ice/BuiltinSequences.ice>

module FileTransfer {
  interface FileStore {
    Ice::ByteSeq read(string name);
```

```
    void write(string name, Ice::ByteSeq bytes);
  };
};
```

Como se puede apreciar, esta interfaz transfiere el contenido de un fichero utilizando una única llamada a procedimiento remoto. Si la aplicación reside en una red local con máquinas con suficiente memoria, estas operaciones simplemente funcionarían estableciendo la propiedad *Ice.MessageSizeMax* a un valor lo suficientemente grande. Sin embargo, y cuanto más grande es el fichero a transferir, generalmente es mejor segmentar dicho fichero y transferirlo utilizando varias llamadas remotas en lugar de una sola.

En las sucesivas secciones se irán presentando distintas aproximaciones para realizar de forma eficiente la transferencia de ficheros, comenzando por una versión sencilla y mejorándola de forma paralela a la eficiencia de dicha transferencia.

7.2. Versión inicial

Como se comentó en la anterior sección, un prototipo de interfaz inicial permitiría transferir un fichero en varios bloques de datos:

```
#include <Ice/BuiltinSequences.ice>

module FileTransfer {
  exception FileAccessException {
    string reason;
  };

  interface FileStore {
    Ice::ByteSeq read(string name, int offset, int num)
      throws FileAccessException;
    void write(string name, int offset, Ice::ByteSeq bytes);
  };
};
```

La operación *read* solicita un determinado número de bytes comenzando en el valor indicado por *offset*. Dicha operación devuelve como mucho *num* bytes. De este modo, el cliente estaría recibiendo el fichero en bloques hasta que recibiese un bloque vacío que indicase el fin de la transferencia.

Esta interfaz, sin embargo, introduce un problema de eficiencia debido a que una única invocación remota se divide en varias invocaciones remotas, lo cual incrementa la latencia. No

obstante, si el fichero es lo suficientemente grande no hay forma de evitar dichas invocaciones, especialmente si el cliente o el servidor no tienen mucha memoria. Por ejemplo, para transferir un fichero de 1 GB en una única invocación, el cliente necesita tener 2 GB de memoria: 1 GB asociado al fichero en cuestión y 1 GB para que Ice cree un buffer de *marshalling* internamente. En el lado del servidor ocurriría lo mismo.

Teniendo en cuenta la interfaz previamente definida, el cliente operaría de la siguiente forma para obtener el contenido de un fichero:

```
// C++
Ice::ObjectPrx base = communicator()->
   stringToProxy("FileStore:tcp -h host -p port");
FileStorePrx fileStorePrx = FileStorePrx::checkedCast(base);

string name = argv[1];
FILE* fp = fopen(name.c_str(), "wb");
Ice::ByteSeq data;
int offset = 0;
int len = 1000 * 1024;

for (;;) {
  try {
    data = fileStorePrx-> read(name, offset, len);
  }
  catch (FileAccessException ex) {
    cout << ex.reason << endl;
    break;
  }
  if (data.size() == 0) {
    break;
  }
  if (fwrite(&data[0], 1, data.size(), fp)
      != data.size()) {
    cerr << "error writing: " << strerror(errno) << endl;
    break;
  }
  offset += data.size();
}

fclose(fp);
```

Como se puede apreciar, el cliente lee datos del almacén y los escribe directamente en el fichero de salida. En la parte del servidor, la operación *read* leería y transferiría los bytes especificados por dicha operación:

```
// C++
::Ice::ByteSeq
FileStoreI::read(const ::std::string& name,
```

```
 ::Ice::Int offset,
 ::Ice::Int num,
 const ::Ice::Current&) const {

  FILE* fp = fopen(name.c_str(), "rb");

  if (fp == 0) {
    FileAccessException ex;
    ex.reason = "cannot open '" + name +
                "' for reading: " + strerror(errno);
    throw ex;
  }

  if (fseek(fp, offset, SEEK_SET) != 0) {
    fclose(fp);
    return Ice::ByteSeq();
  }

  Ice::ByteSeq data(num);
  ssize_t r = fread(&data[0], 1, num, fp);
  fclose(fp);
  if (r!= num)
    data.resize(r);

  return data;

}
```

7.3. Usando AMI

La implementación propuesta en la anterior sección presenta dos problemas de eficiencia. El primero es la latencia de red. El segundo, y más serio aún, es el tiempo empleado en escribir los datos asociados al fichero, en el que toda la actividad de red cesa. Idealmente, se persigue explotar el paralelismo de forma que el servidor entregue el siguiente bloque de datos mientras que el cliente todavía está escribiendo el bloque anterior. Para ello, se puede utilizar AMI enviando una solicitud para el siguiente bloque antes de escribir los datos en el disco.

El primer paso consiste en modificar la definición de la interfaz en Slice:

```
#include <Ice/BuiltinSequences.ice>

module FileTransfer {
  exception FileAccessException {
    string reason;
  };
```

```
  interface FileStore {
    ["ami"] Ice::ByteSeq read(string name, int offset, int num)
      throws FileAccessException;
    void write(string name, int offset, Ice::ByteSeq bytes);
  };
};
```

A continuación, es preciso reescribir parte del bucle principal del cliente:

```
// C++
...
FileStore_readIPtr cb = new FileStore_readI;
fileStorePrx->read_async(cb, name, offset, len);

for (;;) {
  cb->getData(bytes);
  if (bytes.empty()) {
    break;
  }
  offset += bytes.size();
  fileStorePrx->read_async(cb, name, offset, len);
  if (fwrite(&bytes[0], 1, bytes.size(), fp)
      != bytes.size()) {
    cerr << "error writing: " << strerror(errno) << endl;
    break;
  }
}

fclose(fp);
```

El cliente comienza el proceso enviando de forma asíncrona una solicitud de lectura utilizando la operación *read_async* y, posteriormente, entrando en el bucle de lectura-escritura. En dicho bucle, el cliente invoca a *getData* en el objeto de retrollamada, bloqueándose hasta que los datos están disponibles. Una vez que *getData* retorna, el cliente emite una solicitud para el siguiente bloque de datos de forma inmediata. De esta manera, mientras está escribiendo los datos en el disco, el hilo de la parte del cliente puede leer la siguiente respuesta. En teoría, esta aproximación debería proporcionar más paralelismo y mejorar la eficiencia de la aplicación.

La implementación del objeto de retrollamada AMI es sencilla:

```
// C++
class FileStore_readI
  : public AMI_FileStore_read,
    public IceUtil::Monitor<IceUtil::Mutex>
{
```

```
public:
  FileStore_readI() : _done(false)
  {
  }
  virtual void ice_response(const ::Ice::ByteSeq&);
  virtual void ice_exception(const ::Ice::Exception&);
  void getData(::Ice::ByteSeq&);
private:
  bool _done;
  auto_ptr<Ice::Exception> _exception;
  Ice::ByteSeq _bytes;
};

typedef IceUtil::Handle<FileStore_readI> FileStore_readIPtr;
```

La variable *_done* se utiliza para esperar a que una invocación pendiente esté completada, mientras que *_exception* y *_bytes* almacenan los resultados de la invocación.

La implementación de *ice_response* sería la siguiente:

```
// C++
void
FileStore_readI :: ice_response(const ::Ice::ByteSeq& bytes)
{
  Lock sync(*this);
  _bytes = bytes;
  _done = true;
  notify();
}
```

Esta operación se invoca cuando un hilo de la parte cliente recibe la respuesta de una invocación asíncrona. De esta forma, la secuencia de bytes se pasa como argumento a esta operación y se almacena en la variable *_bytes*, pudiendo notificar al monitor que la respuesta ha llegado.

La implementación de *ice_exception* sería la siguiente:

```
// C++
void
FileStore_readI :: ice_exception(const ::Ice::Exception& ex)
{
  Lock sync(*this);
  _exception.reset(ex.ice_clone());
  _done = true;
  notify();
}
```

Dicha operación se ejecuta en el caso de que una invocación desemboque en una excepción. En tal caso, se almacenan los resultados en la variable *_exception* y se notifica al monitor de que ha llegado una respuesta.

Finalmente, *getData* espera a que se complete una invocación asíncrona:

```
// C++
void
FileStore_readI :: getData(::Ice::ByteSeq& bytes)
{
  Lock sync(*this);
  while (!_done) {
    wait();
  }
  _done = false;
  if (_exception.get()) {
    auto_ptr<Ice::Exception> ex = _exception;
    _bytes.clear();
    ex->ice_throw();
  }
  bytes.swap(_bytes);
}
```

La operación espera una notificación de *ice_response* o de *ice_exception*, utilizando para ello la variable *_done*. En caso de que no se produzca una excepción, se intercambia el contenido de las variables *_bytes* y *bytes* (se utiliza *swap* para evitar una copia extra de la secuencia de bytes).

7.4. Incrementando la eficiencia con dos llamadas AMI

De cara a seguir mejorando la eficiencia en la transferencia de archivos se deben considerar tres variables:

- La cantidad de tiempo empleado en realizar el *marshalling* y enviar la petición AMI, es decir, el tiempo empleado por *read_async*.
- El tiempo empleado para esperar la respuesta de la invocación asíncrona, es decir, el tiempo empleado por *getData*.
- El tiempo empleado en escribir los datos al disco, es decir, el tiempo empleado en *fwrite*.

Tanto el tiempo empleado por la operación *read_async* como el tiempo empleado en la escritura de datos en el disco no puede reducirse. Sin embargo, sí que se puede gestionar de

una manera más eficiente el tiempo empleado mientras se espera que la operación *getData* termine. Se pueden reducir los efectos de la latencia de red mantiendo más de una llamada AMI activa. Si se utilizan dos, la primera puede obtener el resultado actual mientras que la segunda emite una solicitud para obtener el siguiente bloque del fichero. De esta forma, el servidor está permanentemente ocupado debido a que tan pronto como envía la respuesta de la primera invocación, la segunda ya está disponible para ser procesada. De manera similar se mejora el comportamiento del cliente. Sin embargo, hay que tener en cuenta que este enfoque es apropiado si el cuello de botella del sistema es la latencia de red, ya que si el problema estuviese en el disco este enfoque no mejoraría apenas la eficiencia.

El bucle principal del cliente quedaría de la siguiente forma:

```
// C++
...
FileStore_readIPtr curr, next;

for (;;) {
  if (!curr) {
    curr = new FileStore_readI;
    next = new FileStore_readI;
    fileStorePrx->read_async(curr, name, offset, len);
  }
  else {
    swap(curr, next);
  }
  fileStorePrx->read_async(next, name, offset + len, len);
  curr->getData(bytes);
  if (bytes.empty()) {
    break;
  }
  if (fwrite(&bytes[0], 1, bytes.size(), fp)
      != bytes.size()) {
    cerr << "error writing: " << strerror(errno) << endl;
    rc = EXIT_FAILURE;
  }
  offset += bytes.size();
}

fclose(fp);
```

En este enfoque se tienen dos objetos de retrollamada, *next* y *curr*, que se intercambian en cada iteración del bucle. El resto del mismo es igual al de la anterior aproximación.

7.5. Uso de la característica *zero-copy* del *mapping* a C++

Otra de las posibles mejoras a aplicar a este sistema de transferencia de ficheros está relacionada con la implementación de la operación *ice_response*:

```
// C++
void
FileStore_readI :: ice_response(const ::Ice::ByteSeq& bytes)
{
  Lock sync(*this);
  _bytes = bytes;
  _done = true;
  notify();
}
```

En el anterior código se puede apreciar cómo el contenido de *bytes* se almacena en la variable de clase *_bytes*. Además, y como se comentó en anteriores secciones, Ice copia estos datos a un buffer interno. Como resultado, 1 GB de datos enviados a la aplicación ocupan 3 GB de memoria: 1 GB para el buffer, 1 GB para *bytes*, y 1 GB para *_bytes*. Este hecho se puede gestionar de forma más eficiente si utilizamos la característica *zero-copy* del *mapping* a C++ para evitar la copia del buffer al vector.

El primer paso consiste en añadir la directiva correspondiente a la operación de la interfaz:

```
interface FileStore {
  ["ami", "cpp:array"] Ice::ByteSeq read(string name, int offset, int num)
    throws FileAccessException;
  ...
};
```

De esta forma, la operación *ice_response* pasa a tener la signatura siguiente:

```
// C++
void
FileStore_readI::
ice_response(const ::std::pair<const ::Ice::Byte*, const ::Ice::Byte*>& bytes);
```

Ahora Ice pasa una pareja de *const Ice::Byte** a la operación. El primer puntero apunta al comienzo de la secuencia y el segundo puntero apunta al final (misma semántica que las operaciones *begin* y *end* de los iteradores STL). Dichos punteros se refieren directamente a posiciones del buffer de *marshalling* de Ice, de forma que se puede evitar una copia de datos cuando se almacenan en la variable *_bytes*:

```
// C++
void
FileStore_readI::
ice_response(const ::std::pair<const ::Ice::Byte*, const ::Ice::Byte*>& bytes)
{
  Lock sync(*this);
  Ice::ByteSeq(bytes.first, bytes.second).swap(_bytes);
  _done = true;
  notify();
}
```

Con esta aproximación, el cliente requiere 2 GB de memoria para recibir un fichero de 1 GB de capacidad: 1 GB para el buffer de Ice y 1 GB para la variable de clase *_bytes*.

7.6. Utilizando AMD en el servidor

La operación *read* de la parte del servidor era la siguiente:

```
// C++
::Ice::ByteSeq
FileStoreI::read(const ::std::string& name,
 ::Ice::Int offset,
 ::Ice::Int num,
 const ::Ice::Current&) const {

  FILE* fp = fopen(name.c_str(), "rb");

  if (fp == 0) {
    FileAccessException ex;
    ex.reason = "cannot open '" + name +
                "' for reading: " + strerror(errno);
    throw ex;
  }

  if (fseek(fp, offset, SEEK_SET) != 0) {
    fclose(fp);
    return Ice::ByteSeq();
  }

  Ice::ByteSeq data(num);
  ssize_t r = fread(&data[0], 1, num, fp);
  fclose(fp);
  if (r!= num)
    data.resize(r);

  return data;

}
```

La implementación es sencilla: se abre el fichero, se posiciona en la localización correcta, se asigna el número correcto de bytes, se leen los datos, y se devuelve el buffer. De nuevo, se necesita 1 GB para el buffer usado por Ice y 1 GB para los datos. Además, y en función del soporte del compilador empleado para NRVO (*Named Return Value Optimization*), el valor de retorno podría requerir una copia adicional. Utilizando AMD, Ice posibilita enviar los datos en lugar de devolver el valor de retorno. De esta forma, la implementación de *read_async* quedaría de la siguiente forma:

```
// C++
void
FileStoreI::read_async(const ::FileTransfer::AMD_FileStore_readPtr& cb,
       const ::std::string& name ,
       ::Ice::Int offset,
       ::Ice::Int num,
       const ::Ice::Current&)
{
  ...
  Ice::ByteSeq bytes(num);
  ssize_t r = fread(&bytes[0], 1, num, fp);
  fclose(fp);
  pair<const Ice::Byte*, const Ice::Byte*> ret;
  ret.first = &bytes[0];
  ret.second = &bytes[r];

  cb->ice_response(ret);

}
```

7.7. Aún más eficiente

En las anteriores secciones se han presentado mejoras significativas, pero todavía existen problemas de eficiencia. Para cada petición el servidor realiza varias acciones:

- Abrir y cerrar el fichero.
- Posicionarse en la localización correcta del fichero.
- Asignar el buffer de lectura para cada petición.

Si se hacen ciertas suposiciones sobre cómo el cliente utiliza el interfaz, se pueden proporcionar ciertas mejoras en la eficiencia. Para llevarlas a cabo se añade una nueva interfaz *File*:

```
#include <Ice/BuiltinSequences.ice>

module FileTransfer {
exception FileAccessException {
string reason;
};

interface File {
["ami", "amd", "cpp:array"] Ice::ByteSeq next();
};
interface FileStore {
File* read(string name, int num);
throws FileAccessException;
void write(string name, int offset, Ice::ByteSeq bytes);
};
};
```

Para leer el contenido de un fichero, el cliente efectúa una llamada a *FileStore::read* y recibe un proxy a un objeto *File*. Entonces, el cliente invoca repetidamente a la operación *next* de dicho objeto hasta que recibe una secuencia vacía, la cual indica el final del archivo. El servidor destruye automáticamente el objeto *File* cuando el cliente obtiene EOF. Con esta inferfaz, los distintos bloques de datos se almacenan secuencialmente por el cliente, es decir, el cliente no especifica un determinado desplazamiento asociado al fichero. A su vez, este hecho permite al objeto *File* mantener en caché el manejador asociado al archivo, evitando abrir y cerrar el fichero para cada bloque de datos. Además, la implementación de *next* no necesita ejecutar la operación *fseek* para establecer explícitamente el desplazamiento del fichero cada vez.

La implementación inicial de *File* sería la siguiente:

```
// C++
class FileI
  : public File
{
public:

  FileI(FILE* fp, int num) :
    _fp(fp),
    _num(num),
    _bytes(new Ice::Byte[num])
  {
  }

  ~FileI()
  {
    delete[] _bytes;
```

```
  }

  virtual void next_async(const ::FileTransfer::AMD_File_nextPtr&, const ::Ice::Curren

private:

  FILE* _fp;
  const int _num;
  Ice::Byte* _bytes;

};

void
FileI::next_async(const ::FileTransfer::AMD_File_nextPtr& cb,
  const ::Ice::Current& current)
{
  pair <const Ice::Byte*, const Ice::Byte*> ret(0, 0);
  ssize_t r = fread(_bytes, 1, _num, _fp);
  if (r == 0) {
    fclose(_fp);
    current.adapter->remove(current.id);
  }
  else {
    ret.first = _bytes;
    ret.second = _bytes + r;
  }
  cb->ice_response(ret);
}
```

El servidor no soporta llamadas concurrentes desde el mismo cliente como una decisión de implementación. De hecho, no existe beneficio para un único cliente en invocar al servidor de forma concurrente, y añadir protección para la concurrencia implicaría un descenso de la eficiencia (esto es distinto de que distintos clientes utilicen el servidor concurrentemente). Para forzar esta restricción se puede configurar al servidor para que utilice el modelo de concurrencia *thread-per-connection* (*hilo por conexión*):

```
# config.server
Ice.ThreadPerConnection=1
```

La parte del cliente necesita adaptarse a esta nueva interfaz:

```
...
try {
  int num = 1000 * 1024;
  FilePrx file = fileStorePrx->read(name, num);

  int offset = 0;
  File_nextIPtr curr, next;
```

```
  Ice::ByteSeq bytes;

  for (;;) {
    if (!curr) {
      curr = new File_nextI;
      next = new File_nextI;
      file->next_async(curr);
    }
    else {
      swap(curr, next);
    }
    file->next_async(next);
    curr->getData(bytes);
    if (bytes.empty()) {
      break;
    }
    if (fwrite(&bytes[0], 1, bytes.size(), fp)
        != bytes.size()) {
      cerr << "error writing: " << strerror(errno) << endl;
      rc = EXIT_FAILURE;
    }
    offset += bytes.size();
  }

}
catch (const FileAccessException& e) {
  cerr << "FileAccessException: " << e.reason << endl;
  rc = EXIT_FAILURE;
}
...
```

7.8. Conclusiones

En este capítulo se ha presentado un enfoque dirigido a mejorar la eficiencia en la transferencia de archivos. Es muy importante diseñar las aplicaciones teniendo en cuenta las capacidades de una máquina para manejar distintos hilos de ejecución. Por fortuna, Ice proporciona mecanismos para aprovechar estas posibilidades, mejorando la eficiencia en la transferencia de ficheros.

Parte IV

Servicios en ICE

Capítulo 8

IceGrid

8.1. Introducción

IceGrid es el servicio de activación y localización de aplicaciones ICE. Dada la creciente importancia de la computación grid, IceGrid se introdujo en la versión 3.0 de ICE para soportar todas las responsabilidades asociadas a la activación y localización de aplicaciones con el objetivo de desarrollar y administrar las mismas en entornos grid.

Una posible configuración grid sería una colección homogénea de computadores que ejecutaran programas parecidos. Cada computador en el grid sería un clon del resto, y todos serían capaces de llevar a cabo una tarea. Por lo tanto, el propio desarrollador tendría que escribir código que se ejecutara en las distintas máquinas, y ICE proporcionaría la infraestructura necesaria que permitiría que los distintos componentes del grid se comunicaran entre sí. Sin embargo, escribir dicho código es sólo una parte del problema. Algunas otras son las siguientes:

- ¿Cómo instalar y actualizar una aplicación en todos los componentes de un grid?
- ¿Cómo mantener los servidores en ejecución?

- ¿Cómo distribuir la carga entre los servidores?
- ¿Cómo migrar un servidor de una máquina a otra?
- ¿Cómo añadir una nueva máquina al grid?

Todos estos temas están directamente relacionados con un grid. IceGrid proporciona soluciones a todas estas preguntas utilizando ciertas características:

- Servicio de localización: IceGrid permite que los clientes contacten con los servidores de forma indirecta, haciendo que las aplicaciones sean más flexibles en lo que a requerimientos se refiere.
- Activación de servidores bajo demanda: IceGrid puede asumir la responsabilidad de activar un servidor bajo demanda, es decir, cuando un cliente intenta contactar con un objeto situado en dicho servidor. Esta acción es completamente transparente para el cliente.
- Distribución de aplicaciones: IceGrid proporciona un mecanismo para la distribución de aplicaciones en un conjunto de máquinas, sin la necesidad de utilizar un sistema de archivos compartidos o complicados *scripts*.
- Balanceado de carga y replicación: IceGrid soporta la replicación agrupando los adaptadores de objetos de determinados servidores en un único adaptador de objetos virtual. Durante la resolución indirecta, el cliente puede ser asociado a cualquiera de estos adaptadores. Además, IceGrid monitoriza la carga de cada máquina y puede utilizar esa información para decidir qué *endpoint* devolver al cliente.
- Asignación de recursos y sesiones: un cliente IceGrid establece una sesión con el objetivo de asignar un recurso, como un objeto o un servidor. IceGrid evita que otros clientes utilicen ese recurso hasta que el cliente inicial lo libere o expire la sesión. Las sesiones pueden ser seguras utilizando el servicio Glacier2.
- Tolerancia a fallos automática: ICE soporta los reintentos automáticos y la tolerancia a fallos en cualquier proxy que contenga múltiples puntos de transporte asociados.

Combinado con el soporte de IceGrid para la replicación y el balanceado de carga, esta característica permite que una solicitud fallida se reenvíe de forma transparente al cliente en el siguiente punto de transporte.

- Preguntas dinámicas: las aplicaciones pueden interactuar directamente con IceGrid para localizar objetos de numerosas formas.
- Monitorización del estado: IceGrid soporta que las interfaces Slice permitan monitorizar sus actividades y recibir notificaciones sobre eventos significativos, habilitando el desarrollo de herramientas para la integración de los eventos de estado de IceGrid en algún *framework* existente.
- Administración: IceGrid incluye herramientas de administración tanto desde la línea de órdenes como gráficas.
- Despliegue: utilizando ficheros XML, el desarrollador puede describir los servidores a desplegar en cada máquina. También permite el uso de plantillas para simplificar la descripción de servidores idénticos.

8.2. Ejemplo de aplicación

En esta sección se presenta una aplicación sencilla que demuestra las principales capacidades de IceGrid. La aplicación consiste en un codificador de pistas de música en archivos con formato mp3. Este proceso suele ser largo y consumir bastantes ciclos de CPU, por la aplicación distribuida aquí presentada acelerará el proceso haciendo uso de los distintos servidores utilizados por la misma, de forma que se puedan procesar varias canciones en paralelo.

La definición en Slice de la interfaz es muy sencilla:

```
module Ripper {

  exception EncodingFailException {
    string reason;
  };

  sequence<byte> ByteSeq;
  sequence<short> Samples;
```

```
  interface Mp3Encoder {
    // Input: PCM samples for left and right channels.
    // Output: MP3 frame(s).
    ByteSeq encode (Samples leftSamples, Samples rightSamples)
throws EncodingFailException;
    // You must flush to get the last frame(s).
    // Flush also destroys the encoder object.
    ByteSeq flush()
throws EncodingFailException;
  };

  interface Mp3EncoderFactory {
    Mp3Encoder* createEncoder();
  };

};
```

La implementación del algoritmo de codificación no es relevante para el propósito de este manual, por lo que el objetivo perseguido será mostrar las capacidades de IceGrid que permiten desarrollar una aplicación eficiente.

La arquitectura inicial de la aplicación se puede observar en la figura 8.1. Como se puede apreciar, ésta consiste en un sencillo nodo IceGrid responsable del servidor de codificación y que se ejecuta en la máquina *ComputeServer*. Además, también se puede comprobar cómo el cliente invoca a la operación *createEncoder* y las acciones que IceGrid lleva a cabo.

Por lo tanto, el código del cliente es muy sencillo:

```
class Client (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        obj = self.communicator().stringToProxy('EncoderFactory')
        ripper = Ripper.Mp3EncoderFactoryPrx.checkedCast(obj)
        encoder = ripper.createEncoder()
        encoder.encode(None, None)
        self.communicator().waitForShutdown()
        return 0

Client().main(sys.argv, 'client.cfg')
```

El cliente utiliza un proxy indirecto para el objeto *EncoderFactory*, en concreto un proxy *bien-conocido*. Este tipo de proxies se refieren a objetos bien-conocidos, es decir, cuya identidad es suficiente para permitir que un cliente los localice.

El descriptor asociado a la aplicación es el siguiente:

```
<icegrid>
```

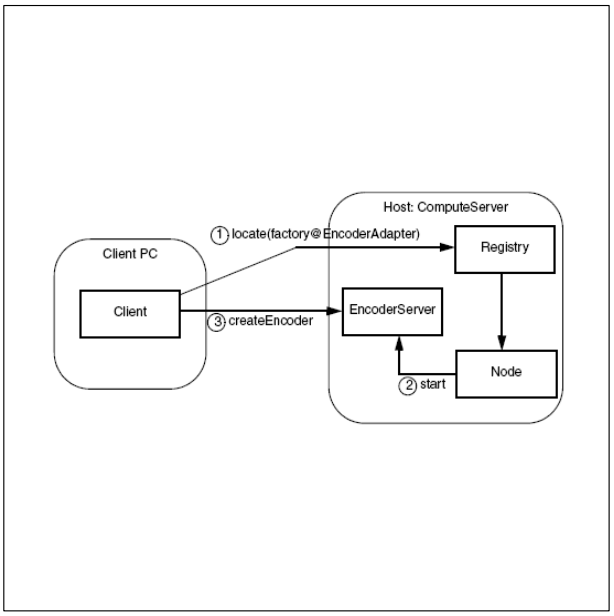

Figura 8.1: Arquitectura básica de la aplicación

```
  <application name="Ripper">

    <replica-group id="EncoderAdapters">
      <load-balancing type="adaptive"/>
      <object identity="EncoderFactory" type="::Ripper::MP3EncoderFactory"/>
    </replica-group>

    <server-template id="EncoderServerTemplate">
      <parameter name="index"/>
      <parameter name="exepath" default="./Server.py"/>
      <server id="EncoderServer${index}"
        exe="${exepath}"
activation="on-demand">
<adapter name="EncoderAdapter"
  replica-group="EncoderAdapters"
  register-process="true"
  endpoints="tcp"/>
      </server>
    </server-template>

    <node name="Node1">
      <server-instance template="EncoderServerTemplate" index="1"/>
    </node>

    <node name="Node2">
      <server-instance template="EncoderServerTemplate" index="2"/>
    </node>
```

```
  </application>
</icegrid>
```

Este descriptor permite la descripción de la aplicación estudiada. Dicho descriptor está basado en el uso de una plantilla, de forma que se definirá una instancia de esa plantilla para cada uno de los nodos de nuestra aplicación. En este caso existen dos nodos, los cuales se diferencian por el parámetro *index*. Mediante esta descripción, tenemos dos objetos del tipo *EncoderFactory* que se tratan como un único objeto virtual gracias a la replicación que se hace explícita mediante los grupos de réplica. En la plantilla también se especifica el sirviente asociado a cada uno de los objetos y el tipo de activación, que en este caso es bajo demanda. Así mismo, también se hace uso de otro concepto de IceGrid: el balanceado de carga. IceGrid permite distintos tipos de balanceado de carga, pero el elegido para esta aplicación ha sido adaptativo, es decir, IceGrid utilizará la información del sistema para elegir el adaptador de objetos menos cargado cuando se efectúa una solicitud.

Como se comentó anteriormente, la arquitectura de IceGrid consiste en un registro y un número determinado de nodos. En este caso, la aplicación contiene dos nodos. Es común hacer que el registro corra en el mismo proceso que uno de los nodos. Para ello se expone la configuración asociada al registro y al node *Node1*:

```
#Registry properties
IceGrid.Registry.Client.Endpoints=default -p 10000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Admin.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=registry

#Node properties
IceGrid.Node.Name=Node1
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=node
IceGrid.Node.CollocateRegistry=1
IceGrid.Node.Trace.Activator=3
Ice.Default.Locator=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000
```

Para la configuración del nodo *Node2* tenemos que definir otro nombre y otro directorio de datos en el caso de que corra en la misma máquina que el primer nodo:

```
#Node properties
IceGrid.Node.Endpoints=default
```

```
IceGrid.Node.Name=Node2
IceGrid.Node.Data=node2
IceGrid.Node.Trace.Activator=3
Ice.Default.Locator=IceGrid/Locator:tcp -h <máquina_registro> -p 10000
```

Como se puede apreciar, con la propiedad *Ice.Default.Locator* se especifica la dirección en la que se está ejecutando el registro. Además, esta misma propiedad habrá que especificarla en el archivo de configuración del cliente:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h <máquina_registro> -p 10000
```

Por último, queda por especificar el código del servidor:

```
class Mp3EncoderI (Ripper.Mp3Encoder):
    def encode (self, leftSamples, rightSamples, current):
        print 'Encoding...'
        return None

    def flush (self, current):
        print 'Flushing...'
        return None

class Mp3EncoderFactoryI (Ripper.Mp3EncoderFactory):
    def createEncoder (self, current):
        prx = current.adapter.addWithUUID(Mp3EncoderI())
        adapterId = current.adapter.getCommunicator().getProperties().
            getProperty(current.adapter.getName() + ".AdapterId")
        return Ripper.Mp3EncoderPrx.uncheckedCast(prx.ice_adapterId(adapterId))

class Server (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        adapter = self.communicator().createObjectAdapter('EncoderAdapter')
        adapter.add(Mp3EncoderFactoryI(), Ice.stringToIdentity('EncoderFactory'))
        adapter.activate()
        self.communicator().waitForShutdown()
        return 0

Server().main(sys.argv)
```

Para ejecutar la aplicación es necesario seguir los siguientes pasos:

1. Arrancar el registro.

2. Arrancar los nodos.

3. Desplegar la aplicación.

4. Ejecutar el cliente.

En Unix se pueden ejecutar las siguientes operaciones:

1. icegridnode –Ice.Config=<registro_nodo1.cfg>
2. icegridnode –Ice.Config=<nodo2.cfg>
3. icegridadmin –Ice.Config=<icegridadmin.cfg> -e “application add Ripper.xml”
4. ./Client.py

Capítulo 9

Freeze

9.1. Introducción

Freeze representa el conjunto de servicios persistentes de ICE:

- ***Freeze evictor*** es una implementación altamente escalable de un servicio de localización de ICE que proporciona persistencia automática y desalojo de objetos ICE con un código de aplicación mínimo.
- ***Freeze map*** es un contenedor asociativo genérico. Las aplicaciones interactúan con *Freeze map* como cualquier otro contenedor asociativo, excepto que las claves y los valores de *Freeze map* son persistentes.

Como se podrá comprobar posteriormente, integrar *Freeze map* o *Freeze evictor* en una aplicación ICE es muy sencillo: una vez definidos los datos persistentes en Slice, Freeze se encarga de los detalles concretos de persistencia. Freeze está implementado utilizando *Berkeley DB*, una base de datos compacta de alto rendimiento.

9.2. *Freeze map*

Un *Freeze map* es un contenedor asociativo y persistente en el que tanto la clave como el valor pueden ser tipos primitivos o tipos definidos por el usuario en Slice. Para cada pareja clave-valor, el desarrollador utiliza una herramienta de generación de código para obtener una clase específica del lenguaje que se ajusta a las convenciones específicas de ese lenguaje. Por ejemplo, en C++ la clase generada está vinculada a *std:map*, y en Java implementa la interfaz *java.util.SortedMap*. La mayoríad de la lógica asociada al almacenamiento y recuperación del estado a y desde una base de datos, respectivamente, está implementada en una clase base Freeze. Las clases generadas derivan de esta clase, por lo que contienen poco código y, por lo tanto, son eficientes en cuanto al tamaño del mismo.

9.2.1. Ejemplo de aplicación

A continuación se presenta un sencillo ejemplo de uso de *Freeze map* en C++, en el que primero se genera un contenedor y, posteriormente, se realizan operaciones básicas sobre él:

```
#include <Freeze/Freeze.h>
#include <StringIntMap.h>

int
main(int argc, char *argv[])
{

  Ice::CommunicatorPtr communicator =
    Ice::initialize(argc, argv);

  Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator, "db");

  StringIntMap map(connection, "simple");
  map.clear();

  Ice::Int i;
  StringIntMap::iterator p;

  // Poblar...
  for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
  }

  // Iterar y cambiar valores...
  for (p = map.begin(); p != map.end(); ++p)
```

```
    p.set(p->second + 1);

  // Encontrar y eliminar el último elemento...
  p = map.find("z");
  assert(p != map.end());
  map.erase(p);

  // Limpieza...
  connection->close();
  communicator->destroy();

  return 0;

}
```

El primer paso consiste en generar el contenedor utilizando el siguiente comando:

```
slice2freeze --dict StringIntMap,string,int StringIntMap
```

El compilador *slice2freeze* crea las clases necesarias en C++ para el *Freeze map*. En este caso, se crea un contenedor denominado *StringIntMap* con un tipo de clave *string* y un valor *int*. El último argumento hace referencia al nombre de los archivos de salida, que en este caso son *StringIntMap.h* y *StringIntMap.cpp*.

9.3. *Freeze evictor*

Freeze evictor combina la persistencia y la escalabilidad en una única facilidad, la cual se puede integrar de una manera sencilla en aplicaciones ICE.

Como implementación de la interfaz *ServantLocator*, el *Freeze evictor* toma ventaja de la separación básica entre objeto ICE y sirviente para activar los sirvientes bajo demanda desde un almacén persistente y desactivarlos de nuevo utilizando las restricciones de desalojo seleccionadas. Aunque una aplicación puede tener miles de objetos ICE en su base de datos, no resulta práctico tener sirvientes para todos esos objetos residiendo en memoria de forma simultánea. La aplicación puede conservar recursos y ganar en escalabilidad estableciendo un límite máximo para el número de sirvientes activos, permitiendo al *Freeze evictor* manejar los detalles asociados a la activación, persistencia, y desactivación de sirvientes.

Freeze evictor mantiene una cola de servidores activos, utilizando un algoritmo de desalojo *menos recientemente usado*: si la cola está llena, el sirviente menos recientemente usado deja su puesto a un nuevo sirviente.

9.3.1. Ejemplo de aplicación

En general, incorporar un *Freeze evictor* en una aplicación requiere los siguientes pasos:

1. Evaluar si las definiciones Slice actuales son adecuados para manejar los tipos de datos persistentes.
2. Si no es así, típicamente se definirá una nueva clase derivada que reflejará las restricciones asociadas al estado persistente. Es buena práctica definir estos datos en un archivo distinto al vinculado a la interfaz pública mostrada al cliente y utilizando otro módulo distinto.
3. Generar el código (utilizando *slice2freeze* o *slice2freezej*) para las nuevas definiciones.
4. Crear un *evictor* y registrarlo como un localizador de sirvientes con un adaptador de objetos.
5. Crear instancias de los tipos persistentes de la aplicación y registrarlos con el *evictor*.

El ejemplo aquí presentado es el conocido *¡Hola mundo!*, cuya especificación Slice es la siguiente:

```
module Demo {
  interface Hello {
    void puts(string str);
  };

  class HelloPersistent implements Hello {
    int useCount;
  };
};
```

En este caso, la persistencia viene reflejada por la clase *HelloPersistent*, la cual añade persistencia para reflejar el número de veces que se invoca a la operación *puts*.

La parte de la aplicación encargada de crear el evictor sería la siguiente:

```
#include <Freeze/Freeze.h>
#include <IceUtil/IceUtil.h>
#include <Hello.h>

using namespace std;
using namespace Demo;
```

```
class Server: public Ice::Application
{

public:
  virtual int run (int argc, char* argv[]) {
    shutdownOnInterrupt();

    // Crear la fábrica de objetos ICE HelloFactory.
    communicator()
      ->addObjectFactory(new HelloFactory(),
Demo::HelloPersistent::ice_staticId());

    // Crear el adaptador de objetos.
    Ice::ObjectAdapterPtr oa = communicator()
      ->createObjectAdapterWithEndpoints("HelloOA", "tcp -p 8888");

    // Inicializar el evictor.
    Freeze::EvictorPtr e =
      Freeze::createEvictor(oa, "db", "hello", new HelloInitializer());
    oa->addServantLocator(e, "");

    // Creación de los objetos si no estaban gestionados por el evictor.
    for (int i=0; i<5; ++i) {
      char str[] = "hello?";
      str[5]='0' + i;
      Ice::Identity ident = communicator()->stringToIdentity(str);
      if (e->hasObject(ident)) {
cout << ident.name << " ya estaba en este evictor" << endl;
      }
      else {
Ice::ObjectPrx obj = e->add(new HelloI(), ident);
cout << communicator()->proxyToString(obj) << endl;
      }
    }

    oa->activate();
    communicator()->waitForShutdown();
    cout << "Done" << endl;
    return 0;
  }

};

int
main(int argc, char* argv[])
{

  Server* app = new Server();
  app->main(argc, argv);
  exit(0);

}
```

El sirviente ha heredar de *Demo::HelloPersistent* (clase generada por el compilador de Slice) y de *IceUtil::AbstractMutex* (por cuestiones asociadas a la sincronización):

```
class HelloI: public Demo::HelloPersistent,
      public IceUtil::AbstractMutexI<IceUtil::Mutex>
{

  friend class HelloInitializer;
public:
  HelloI() { useCount = 0; }
  virtual void puts(const string& str, const Ice::Current& current) {
    cout << current.id.name << " (" << useCount << "): " << str << endl;
    useCount += 1;
  }

};
```

También es necesario definir clases asociadas a las implementaciones de la fábrica de objetos y a la inicialización del sirviente:

```
class HelloFactory: public Ice::ObjectFactory
{

public:
  virtual Ice::ObjectPtr create(const string& type) {
    return new HelloI();
  }

  virtual void destroy() {}

};

class HelloInitializer: public Freeze::ServantInitializer
{

public:
  virtual void initialize(const Ice::ObjectAdapterPtr& oa,
  const Ice::Identity& ident,
  const string& str,
  const Ice::ObjectPtr& obj) {
    HelloIPtr ptr = HelloIPtr::dynamicCast(obj);
    cout << ident.name << " Inicial: " << ptr->useCount << endl;
  }

};
```

Capítulo 10

Glacier2

10.1. Introducción

En las distintas aplicaciones ICE con propósitos docentes tanto el cliente como el servidor suelen ejecutarse en la misma máquina. Sin embargo, en entornos de red más reales la situación es mucho más complicada: las máquinas en las que se ejecutan los clientes y los servidores suelen estar detrás de un *router* o cortafuegos que no sólo restringe las conexiones entrantes, sino que también hace que dichas máquinas se sitúen en un espacio de direcciones privado utilizando NAT.

Dentro de este marco, **Glacier2** es una solución de cortafuegos ligera para aplicaciones ICE, que permite a los clientes y a los servidores comunicarse a ravés de un cortafuegos de una manera segura. El tráfico cliente-servidor queda totalmente encriptado usando certificados de clave pública de forma bidireccional. Además, Glacier2 proporciona soporte para autenticación y gestión de sesiones. El escenario virtual creado por Glacier2 se puede observar en la figura 10.2.

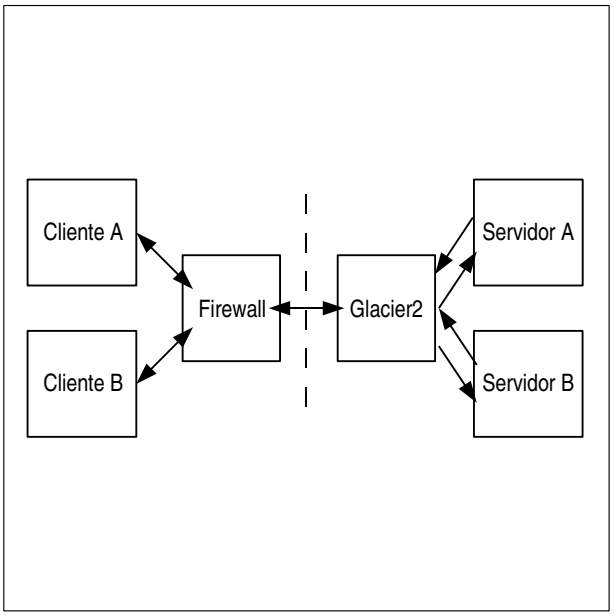

Figura 10.1: Escenario creado por Glacier2

El uso de Glacier2 proporciona las siguientes ventajas:

- Los clientes requieren un cambio mínimo.
- Sólo es necesario un puerto para soportar cualquier número de servidores.
- El número de conexiones a los servidores es reducido.
- Los servidores no se dan cuenta de la presencia de Glacier2, ya que éste actúa como un cliente más. Además, IceGrid puede seguir usándose de forma transparente.
- Las retrollamadas de los servidores se envían sobre la conexión establecida entre el cliente y el servidor.
- Glacier2 no requiere conocimiento de la aplicación.
- Glacier2 ofrece servicios de gestión y autenticación de sesiones.

El núcleo de ICE proporciona un *router* genérico, representado por la interfaz *Ice::Router*, que permite a un tercer servicio interceptar las peticiones a un proxy configurado correcta-

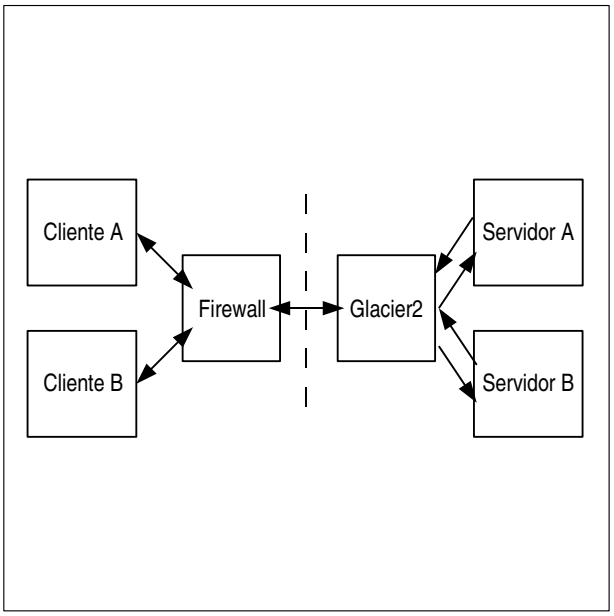

Figura 10.2: Escenario creado por Glacier2

mente y las entrega al servidor previsto. Glacier2 es una implementación de este servicio, aunque sería posible llevar a cabo otras implementaciones. Glacier2 normalmente se ejecuta en una máquina de la red privada detrás de un cortafuegos, pero también puede operar en una máquina con acceso a redes públicas y privadas.

En la parte del cliente, los proxies deben configurarse para utilizar Glacier2 como router. Dicha configuración puede llevarse a cabo de una forma estática para todos los proxies creados o especificándola en el código para un proxy en concreto. Un proxy configurado para utilizar Glacier2 se denomina *routed proxy*. De forma resumida, Glacier2 actúa como un cliente local en nombre de un cliente remoto.

10.2. Ejemplos de aplicaciones

10.2.1. Ejemplo básico

A continuación, se presenta el clásico ejemplo del *¡Hola mundo!* utilizando Glacier2 y IceGrid. Para utilizar Glacier2 y una configuración mínima es necesario desarrollar las siguientes tareas:

- Escribir un archivo de configuración para el *router*.
- Escribir un archivo de contraseñas para el *router* [1].
- Decidir si utilizar el gestor de sesiones interno del *router* o utilizar uno propio.
- Arrancar el *router* en una máquina con acceso a redes privadas y públicas.
- Modificar la configuración del cliente para utilizar el *router*.
- Modificar el cliente para crear una sesión hacia el *router*.

Para configurar el router se utiliza el siguiente archivo de configuración:

```
Glacier2.Client.Endpoints=tcp -h localhost -p 8000
Glacier2.SessionTimeout=60
```

El *endpoint* definido por *Glacier2.Client.Endpoints* por el núcleo de ejecución de ICE en un cliente para interactuar directamente con el router, el cual se define en la interfaz de red pública porque debe ser accesible para los clientes.

La propiedad *Glacier2.SessionTimeout* está vinculada a la duración de una sesión. Ésta puede terminar en caso de que un cliente lo haga de forma explícita, o puede expirar en el caso de que el número de segundos definidos en dicha propiedad se haya cumplido. Es recomendable establecer siempre un valor para esta propiedad.

El siguiente paso es definir un archivo de contraseñas e informar al *router* de ello:

```
Glacier2.CryptPasswords=passwords.cfg
```

[1] En el siguiente ejemplo se propondrán métodos de autenticación de usuarios alternativos.

El formato de este archivo se corresponde a una serie de parejas nombre-contraseña, con cada pareja en una línea distinta del archivo y un espacio en blanco entre ambos campos:

```
david QupfJsB7qmL4E
```

Se puede utilizar la utilidad *openssl* para generar las contraseñas:

```
openssl
OpenSSL> passwd
Password:
Verifying - Password:
QupfJsB7qmL4E
```

El siguiente paso consiste en arrancar el *router* pero, como en este ejemplo se ha utilizado IceGrid también, antes es necesario indicar a Glacier2 cómo puede contactar con el servicio de localización IceGrid. Para ello se añade lo siguiente al archivo de configuración del *router*:

```
Ice.Trace.Network=2
Ice.Default.Locator=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000
```

La aplicación a desplegar se especifica en el archivo *app.xml*:

```
<icegrid>
 <application name="DemoGlacier2">
  <node name="localhost">
   <server id="HolaMundo" exe="./Server.py" activation="on-demand">
    <adapter name="HolaMundoAdapter" id="HolaMundoAdapter"
             register-process="true" endpoints="tcp">
     <object identity="holaMundo" type="::Demo::HolaMundo"/>
    </adapter>
   </server>
  </node>
 </application>
</icegrid>
```

Luego ya se pueden ejecutar los siguientes comandos:

```
icegridnode --Ice.Config=icegrid.cfg
icegridadmin --Ice.Config=icegridadmin.cfg -e ``application add app.xml''
glacier2router --Ice.Config=router.cfg
```

El contenido del archivo *icegrid.cfg* es el siguiente:

```
#Registry properties
IceGrid.Registry.Client.Endpoints=default -p 10000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Admin.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=registry

#Node properties
IceGrid.Node.Name=localhost
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=node
IceGrid.Node.CollocateRegistry=1
IceGrid.Node.Trace.Activator=3
Ice.Default.Locator=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000
```

El contenido del archivo *icegridadmin.cfg* es el siguiente:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000
```

El siguiente paso es crear el cliente y especificar su archivo de configuración:

```
import Ice, Glacier2, sys
Ice.loadSlice('HolaMundo.ice', ['-I' '/usr/share/slice'])
import Demo

class Client (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        router = Glacier2.RouterPrx.checkedCast(self.communicator().getDefaultRouter()
        session = router.createSession('david', 'david')
        obj = self.communicator().stringToProxy('holaMundo')
        prx = Demo.HolaMundoPrx.uncheckedCast(obj)
        prx.saludar('david')
        self.communicator().waitForShutdown()
return 0

Client().main(sys.argv, 'client.cfg')
```

En el código se puede apreciar cómo el cliente crea una sesión hacia el *router*, indicando el nombre del usuario y su contraseña. Como se puede apreciar, no resulta deseable exponer la contraseña en dicho código, por lo que en el próximo ejemplo se estudiarán métodos de autenticación adicionales.

El archivo de configuración *client.cfg* es el siguiente:

```
Ice.Default.Router=Glacier2/router:tcp -h localhost -p 8000
Ice.ACM.Client=0
Ice.MonitorConnections=60
Ice.RetryIntervals=-1
```

La propiedad *Ice.Default.Router* permite que el cliente conozca la ubicación de Glacier2. La propiedad *Ice.ACM.Client* gobierna el comportamiento de la gestión de conexiones activas (*active connection management* o ACM), la cual permite ahorrar recursos cerrando conexiones *ociosas* de forma periódica. Esta característica ha de ser desabilitada en un cliente que utiliza un *router* Glacier2, ya que de otra forma ACM finalizaría las sesiones de forma prematura.

Para los clientes que utilizan AMI se establece la propiedad *Ice.MonitorConnections* de manera que los *timeouts* se gestionen correctamente. Si esta propiedad no se define, toma por defecto el valor de la propiedad *Ice.ACM.Client*, que en este caso tiene el valor 0. Por lo tanto, siempre que se desactiva ACM es necesario especificar un valor para *Ice.ACM.Client*. Finalmente, estableciendo *Ice.RetryIntervals* a -1 se desactivan los reintentos automáticos, los cuales no son útiles para los proxies configurados para usar un *router* Glacier2.

Por último, queda por definir el servidor, el cual es muy sencillo e implementa la interfaz *HolaMundo* definida en otras secciones:

```
import Ice, Glacier2, sys
Ice.loadSlice('HolaMundo.ice', ['-I' '/usr/share/slice'])
import Demo

class HolaMundoI (Demo.HolaMundo):
    def saludar (self, nombre, current):
        print '¡Hola ' + nombre + '!'

class Server (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        adapter = self.communicator().createObjectAdapter('HolaMundoAdapter')
        adapter.add(HolaMundoI(), Ice.stringToIdentity('holaMundo'))
        adapter.activate()
        self.communicator().waitForShutdown()
return 0

Server().main(sys.argv)
```

El archivo de configuración asociado es el siguiente:

```
Ice.Default.Router=IceGrid/Locator:tcp -h 127.0.0.1 -p 10000
```

10.2.2. Ejemplo avanzado

En esta sección se presenta un ejemplo que permite validar la identidad de los usuarios a través de la interfaz *Glacier2.PermissionsVerifier*. Es un ejemplo sencillo porque lo que pretende es ilustrar al lector con la funcionalidad de esta intefaz.

El servidor en Python es el siguiente:

```
import sys, traceback, Ice, Glacier2

class PermissionsVerifierI (Glacier2.PermissionsVerifier):
    def checkPermissions (self, usuario, password, current):
        # Validación de permisos...
        return (True, '')

class Server (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        adapter = self.communicator().createObjectAdapterWithEndpoints(
            'VerifierAdapter', 'default -p 10000')
        adapter.add(PermissionsVerifierI(), Ice.stringToIdentity('verifier'))
        adapter.activate()
        self.communicator().waitForShutdown()
        return 0

Server().main(sys.argv)
```

Como se puede apreciar, hay que implementar la operación *checkPermissions* del sirviente asociado a la interfaz *PermissionsVerifier*. En este caso, se validaría a cualquier usuario.

El cliente también es muy básico:

```
import sys, traceback, Ice, Glacier2

class Client (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        obj = self.communicator().stringToProxy('verifier:default -p 10000')
        prx = Glacier2.PermissionsVerifierPrx.checkedCast(obj)
        ret, err = prx.checkPermissions('usuario', 'password')
        if ret:
            print 'Usuario validado.'
        else:
            print 'Usuario no validado.'
        self.communicator().waitForShutdown()
        return 0

Client().main(sys.argv)
```

Aunque en este ejemplo se ha utilizado TCP como mecanismo de transporte, este método no es seguro debido a que la contraseña del usuario viaja en claro por la red. Por ello, resultaría más apropiado utilizar SSL como mecanismo de transporte.

Capítulo 11

IceBox

11.1. Introducción

IceBox es un *framework* para servicios de aplicaciones ICE fácil de usar. Este elemento se puede entender como una implementación del patrón *configurador de servicios* en lo relativo a servicios ICE. Este patrón es una técnica útil para la configuración de servicios de forma que su administración queda centralizada. En términos prácticos, este hecho significa que los servicios se desarrollan como componentes que se cargan dinámicamente y que pueden configurarse a través de un servidor de propósito general según sea necesario.

Existen distintas ventajas a la hora de utilizar IceBox:

- Los servicios cargados por un mismo servidor IceBox pueden configurarse para obtener las ventajas derivadas de las optimizaciones de asignación de ICE.
- La composición de una aplicación consistente en varios servicios se lleva a cabo a través de la configuración, sin la necesidad de compilar o enlazar.
- Varios servicios Java se pueden activar en una única instancia de la máquina virtual de Java (JVM).

- Los servicios implementan una interfaz del servicio IceBox, obteniendo un *framework* común para los desarrolladores y utilidades de administración centralizadas.
- IceBox soporta la integración con IceGrid.

Para crear un servicio IceBox es necesario implementar una de las interfaces de IceBox. La implementación que se expondrá en la siguiente sección implementa la interfaz *IceBox::Service*:

```
module IceBox {
  local interface Service {
    void start (string name,
                Ice::Communicator communicator,
                Ice::StringSeq args);
    void stop ();
  };
};
```

Como se puede apreciar, un servicio sólo necesita implementar dos operaciones, las cuales son invocadas por el gestor del servicio. *start* es invocada después de que arranque el servicio, y *stop* cuando el servidor IceBox finaliza su ejecución.

11.2. Ejemplo de aplicación

El ejemplo aquí presentado está basado en el clásico *¡Hola mundo!*, y en este caso se presentará utilizando el lenguaje Java. La clase asociada a la definición del servicio es muy sencilla:

```
public class HelloServiceI extends Ice.LocalObjectImpl
    implements IceBox.Service
{

    public void
    start(String name,
          Ice.Communicator communicator,
          String[] args)
    {
      _adapter = communicator.createObjectAdapter(name);
      Ice.Object object = new HelloI(communicator);
      _adapter.add(object, Ice.Util.stringToIdentity("hello"));
      _adapter.activate();
    }
```

```
    public void
    stop()
    {
      _adapter.deactivate();
    }

    private Ice.ObjectAdapter _adapter;

}
```

Como se puede apreciar la clase es muy sencilla, y básicamente se resume en la implementación del método *start*, en la que se crea el adaptador de objetos con el mismo nombre del servicio, se activa el sirviente del tipo *HelloI*, y se activa el adaptador de objetos.

Para configurar un servicio en un servidor IceBox se utiliza una única propiedad con distintos propósitos: define el nombre del servicio, proporciona el punto de entrada a ese servicio al gestor de servicios, y define las propiedades y los argumentos para el servicio:

```
IceBox.Service.\textit{name}=\textit{punto_entrada} [argumentos]
```

Para la aplicación descrita anteriormente podríamos tener la siguiente propiedad:

```
IceBox.Service.Hello=HelloServiceI
```

Para configurar IceBox se puede utilizar el siguiente archivo de configuración:

```
IceBox.InstanceName=DemoIceBox
IceBox.ServiceManager.Endpoints=tcp -p 9998
IceBox.Service.Hello=HelloServiceI
Hello.Endpoints=tcp -p 10000:udp -p 10000
Ice.Warn.Connections=1
```

Para arrancarlo se puede utilizar el siguiente comando:

```
java IceBox.Server --Ice.Config=config.icebox
```

Capítulo 12

IceStorm

12.1. Introducción

IceStorm es un eficiente servicio de publicación-subscripción para aplicaciones ICE. Actúa de mediador entre el publicador y el subscriptor, ofreciendo varias ventajas:

- Para distribuir información a los subscriptores sólo es necesaria una simple llamada al servicio IceStorm.
- Los monitores interactúan con el servidor IceStorm para llevar a cabo la subscripción.
- Los cambios de la aplicación para incluir IceStorm son mínimos.

Una aplicación indica su interés en recibir mensajes subscribiéndose a un *topic*. Un servidor IceStorm soporta cualquier número de *topics*, los cuales son creados dinámicamente y distinguidos por un nombre único. Cada *topic* puede tener varios publicadores y subscriptores. Un *topic* es equivalente a una interfaz Slice: las operaciones del interfaz definen los tipos de mensajes soportados por el *topic*. Un publicador usa un *proxy* al interfaz *topic* para enviar sus mensajes, y un subscriptor implementa la interfaz *topic* (o derivada) para recibir

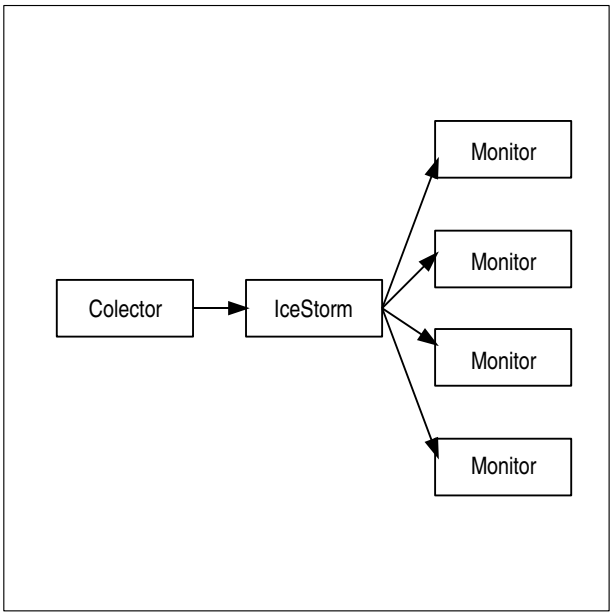

Figura 12.1: Ejemplo de aplicación con IceStorm

los mensajes. Realmente dicha interfaz representa el contrato entre el publicador (cliente) y el subscriptor (servidor), excepto que IceStorm encamina cada mensaje a múltiples receptores de forma transparente.

12.2. Ejemplo de aplicación

En esta sección se presentará el ejemplo asociado a la aplicación mostrada en la figura 12.1, la cual está basada en un monitor de medida de condiciones meteorológicas. En este ejemplo se mostrará cómo crear, suscribirse, y publicar mensajes en un *topic*.

Las definiciones en Slice de esta aplicación son las siguientes:

```
module Demo {

  struct Measurement {
    string tower;        // Tower id.
    float windSpeed;     // Knots.
    short windDirection; // Degrees.
    float temperature;   // Degrees Celsius.
  };
```

```
  interface Monitor {
    void report(Measurement m);
  };

};
```

Monitor representa la interfaz *topic*, y sólo tiene una operación (*report*).

Para implementar un publicador, que en este caso se refiere a la parte de la aplicación que toma la condición climática, se han de seguir una serie de sencillos pasos:

1. Obtener un proxy para el *TopicManager*. Este es el principal objeto IceStorm, utilizado tanto por los publicadores como por los subscriptores.

2. Obtener un proxy para el topic *Weather*, creando el *topic* si no existe o recuperándolo.

3. Obtener un proxy para el objeto publicador del *topic Weather*. El objetivo de este proxy es el de publicar mensajes, por lo que deberá hacerse una conversión a una interfaz *Monitor*.

4. Recoger y publicar las medidas.

El publicador en Python tendría un aspecto parecido al siguiente código:

```
import sys, traceback, time, Ice, IceStorm
Ice.loadSlice('WeatherMonitor.ice', ['-I' '/usr/share/slice'])
import Demo

class Publisher (Ice.Application):
    def run (self, argv):
        self.shutdownOnInterrupt()
        properties = self.communicator().getProperties()
        proxy = properties.getProperty('IceStorm.TopicManager.Proxy')
        obj = self.communicator().stringToProxy(proxy)
        topicManager = IceStorm.TopicManagerPrx.checkedCast(obj)
        try:
            topic = topicManager.retrieve('Weather')
        except IceStorm.NoSuchTopic:
            topic = topicManager.create('Weather')
        pub = topic.getPublisher()
        if (pub.ice_isDatagram()):
            pub = pub.ice_oneway()
        monitor = Demo.MonitorPrx.uncheckedCast(pub)
        for i in range (1, 10):
            m = Demo.Measurement()
            # Get the measurement...
```

```
            monitor.report(m)
            time.sleep(1)
        self.communicator().waitForShutdown()
        return 0

Publisher().main(sys.argv, 'publisher.cfg')
```

Básicamente, lo que se hace es obtener un proxy al *topic manager*, para después obtener un proxy al *topic Weather*. El siguiente paso es obtener un proxy al publicador, el cual se convierte a un proxy del tipo *Monitor*, es decir, a un proxy de la interfaz definida en el archivo de definiciones. Por último, y a modo de ejemplo, el publicador realiza 10 publicaciones utilizando la operación *report*.

El archivo de configuración asociado al publicador es el siguiente:

```
IceStorm.TopicManager.Proxy=DemoIceStorm/TopicManager:default -p 10000
```

El siguiente paso consiste en definir el suscriptor, a partir de una serie de pasos:

1. Obtener un proxy para el *TopicManager*.
2. Crear un adaptador de objetos que albergue al sirviente asociado a la interfaz *Monitor*.
3. Instanciar y activar dicho sirviente en el adaptador de objetos.
4. Suscripción al *topic Weather*.
5. Procesas los mensajes recibidos hasta el momento de su finalización.
6. Eliminar la suscripción al *topic Weather*.

El código en Python sería parecido al siguiente:

```
import sys, traceback, Ice, IceStorm
Ice.loadSlice('WeatherMonitor.ice', ['-I' '/usr/share/slice'])
import Demo

class MonitorI (Demo.Monitor):
    def report (self, measurement, current):
        # Process the measurement...
        print 'Obtaining a new measurement'

class Subscriber (Ice.Application):
    def run (self, argv):
```

```
        self.shutdownOnInterrupt()
        properties = self.communicator().getProperties()
        proxy = properties.getProperty('IceStorm.TopicManager.Proxy')
        obj = self.communicator().stringToProxy(proxy)
        topicManager = IceStorm.TopicManagerPrx.checkedCast(obj)

        adapter = self.communicator().createObjectAdapter('MonitorAdapter')
        monitorPrx = adapter.addWithUUID(MonitorI())

        try:
            topic = topicManager.retrieve('Weather')
            topic.subscribe(None, monitorPrx)
        except IceStorm.NoSuchTopic:
            # Process error...
            print 'Topic not found!'

        adapter.activate()

        self.communicator().waitForShutdown()
        topic.unsubscribe(monitorPrx)
        return 0

Subscriber().main(sys.argv, 'subscriber.cfg')
```

El archivo de configuración del suscriptor sería el siguiente:

```
MonitorAdapter.Endpoints=tcp
IceStorm.TopicManager.Proxy=DemoIceStorm/TopicManager:default -p 10000
```

Para probar la aplicación hay que tener en cuenta que IceStorm es un servicio relativamente ligero y que se implementa como un servicio IceBox. Luego el primer paso sería arrancar IceStorm:

```
icebox --Ice.Config=icebox.cfg
```

El contenido de dicho archivo de configuración tendría un aspecto parecido al siguiente:

```
IceBox.ServiceManager.Endpoints=tcp -p 9998
IceBox.Service.IceStorm=IceStormService,31:createIceStorm --Ice.Config=icestorm.cfg
```

El siguiente paso sería utilizar la herramienta administrativa *icestormadmin* para indicar las propiedades asociadas a la instancia de IceStorm:

```
icestoradmin --Ice.Config=icestorm.cfg
```

Donde dicho archivo de configuración[1] tendría un aspecto parecido a:

[1] Para más información estudiar la parte de configuración de IceStorm en [4].

```
IceStorm.TopicManager.Proxy=DemoIceStorm/TopicManager:default -p 10000
IceStorm.TopicManager.Endpoints=default -p 10000
IceStorm.InstanceName=DemoIceStorm
IceStorm.Publish.Endpoints=default
IceStorm.Trace.TopicManager=2
IceStorm.Trace.Topic=1
IceStorm.Trace.Subscriber=1
IceStorm.Trace.Flush=1
IceStorm.Flush.Timeout=2000
Freeze.DbEnv.IceStorm.DbHome=db
```

Y por último, ya sólo quedaría arrancar los distintos subscriptores y el publicador de eventos.

Capítulo 13

IcePatch2

13.1. Introducción

IcePatch2 es un eficiente servicio de actualización de archivos, fácil de configurar y fácil de usar. Incluye los siguientes componentes:

- El servidor IcePatch (*icepatch2server*).
- Un cliente IcePatch basado en texto (*icepatch2client*).
- Una herramienta basada en texto para comprimir archivos y calcular las sumas de comprobación (*icepatch2calc*).
- Una API Slice y una biblioteca C++ para el desarrollo de clientes IcePatch2.

Como ocurre con el resto de servicios de ICE, IcePatch2 puede configurarse para utilizar facilidades de ICE como Glacier2 o IceSSL.

Conceptualmente, IcePatch2 es muy simple. El servidor tiene la responsabilidad asociada a un directorio de archivos (el directorio de datos), el cual contiene los directorios y archivos a distribuir entre los distintos clientes IcePatch2. El desarrollador puede utilizar *icepatch2calc*

para comprimir estos archivos y crear un archivo que contenga una suma de comprobación para cada uno de ellos. El servidor transmite estos archivos al cliente, el cual obtiene el directorio de datos y su contenido, actualizando cualquier archivo que haya cambiado desde su última ejecución.

13.2. Ejemplo de aplicación

Añadir la distribución de aplicaciones al ejemplo comentado en la sección de IceGrid es muy simple, ya que requiere sólo unos cambios en el descriptor de la aplicación:

```
<icegrid>
  <application name="Ripper">

    <server-template id="IcePatch2">

       <parameter name="instance-name"
         default="${application}.IcePatch2"/>
       <parameter name="endpoints" default="default"/>
       <parameter name="directory"/>

       <server id="${instance-name}" exe="icepatch2server"
         application-distrib="false" activation="on-demand">

         <adapter name="IcePatch2" endpoints="${endpoints}">
           <object identity="${instance-name}/server"
     type="::IcePatch2::FileServer"/>
         </adapter>

         <properties>
            <property name="IcePatch2.Admin.Endpoints"
      value="tcp -h 127.0.0.1"/>
            <property name="IcePatch2.Admin.RegisterProcess"
      value="1"/>
            <property name="IcePatch2.InstanceName"
      value="${instance-name}"/>
            <property name="IcePatch2.Directory"
      value="${directory}"/>
         </properties>
       </server>

    </server-template>

    <replica-group id="EncoderAdapters">
      <load-balancing type="round-robin"/>
      <object identity="EncoderFactory" type="::Ripper::MP3EncoderFactory"/>
    </replica-group>

    <server-template id="EncoderServerTemplate">
```

```
      <parameter name="index"/>
      <parameter name="exepath" default="./Server.py"/>
      <server id="EncoderServer${index}"
        exe="${exepath}"
activation="on-demand">
<adapter name="EncoderAdapter"
  replica-group="EncoderAdapters"
  register-process="true"
  endpoints="tcp"/>
      </server>
    </server-template>

    <distrib/>
    <node name="Node1">
      <server-instance template="EncoderServerTemplate" index="1"/>
      <server-instance template="IcePatch2" directory="./distrib.simple"/>
    </node>

    <node name="Node2">
      <server-instance template="EncoderServerTemplate" index="2"/>
    </node>

  </application>
</icegrid>
```

Debido a que cada nodo IceGrid es en realidad un cliente IcePatch2, cada nodo lleva a cabo la actualización como si se tratara de un cliente IcePatch2: descarga todo lo necesario si no tiene una copia local de la distribución o realiza una actualización incremental de aquellos archivos que han cambiado.

Bibliografía

[1] Benoit Foucher. Grid Computing with IceGrid. *Connections*, pages 2–7, December 2005.

[2] Michi Henning. Teach Yourself IceGrid in 10 Minutes. *Connections*, pages 11–15, November 2006.

[3] Michi Henning. The Rise and Fall of Corba. 2006.

[4] Henning M. and Spruiell M. Distributed Programming with Ice. Technical report, ZeroC ICE, October 2006.

[5] Matthew Newhook. Advanced Use of Glacier2. *Connections*, pages 7–11, May 2005.

[6] Matthew Newhook. An Introduction to IceStorm. *Connections*, pages 2–7, June 2005.

[7] Matthew Newhook. Session Management with Glacier2. *Connections*, pages 2–6, April 2005.

[8] Matthew Newhook. IceGrid Security. *Connections*, pages 2–17, September 2006.

[9] Matthew Newhook. Session Management with IceGrid. *Connections*, pages 2–9, October 2006.

[10] Mark Spruiell. Ice for Ruby. *Connections*, pages 2–11, December 2006.