Computación Distribuida
con ZeroC Ice

Introducción

Un middleware de comunicaciones es un sofisticado sistema de IPC (Inter-Process Communication) orientado a mensajes. A diferencia de los otros IPC como los sockets, los middlewares suelen ofrecer soporte para interfaces concretas entre las entidades que se comunican, es decir, permiten definir la estructura y semántica para los mensajes.
Se puede entender un middleware como un software de conectividad que hace posible que aplicaciones distribuidas pueden ejecutarse sobre distintas plataformas heterogéneas, es decir, sobre plataformas con distintos sistemas operativos, que usan distintos protocolos de red, y que incluso involucran distintos lenguajes de programación en la aplicación distribuida.
Desde otro punto de vista, se puede ver el middleware como una abstracción de la complejidad y de la heterogeneidad que las redes de comunicaciones imponen. De hecho, uno de los objetivos de un middleware es ofrecer un acuerdo en las interfaces y en los mecanismos de interoperabilidad, como solución a los distintos desacuerdos en hardware, sistemas operativos, protocolos de red, y lenguajes de programación.
Existen muchísimos middlewares de comunicaciones. En el caso de RPC (Remote Procedure Call) se indican un conjunto de funciones que podrán ser invocadas remotamente por un cliente. Los demás, y la mayoría de los actuales, son RMI, es decir, son middlewares orientados a objetos. La figura 1.1 muestra el esquema de invocación remota a través de un núcleo de comunicaciones típico de este tipo de middlewares.

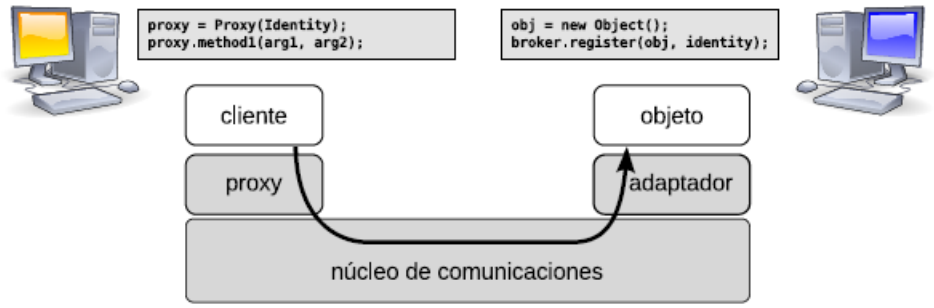


FIGURA 1.1: Invocación a método remoto

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

El middleware encapsula técnicas de programación de sockets y gestión de concurrencia que pueden ser realmente complejas de aprender, implementar y depurar; y que con él podemos aprovechar fácilmente. El middleware se encarga también de gestionar problema inherentes a las comunicaciones: idéntica y numera los mensajes, comprueba duplicados, controla retransmisiones, conectividad, asigna puertos, gestiona el ciclo de vida de las conexiones, identifica los objetos, proporciona soporte para invocación y despachado asíncrono; y un largo etcétera.

ZeroC Ice

Ice (Internet Communication Engine) es un middleware de comunicaciones orientado a objetos desarrollado por la empresa ZeroC Inc Ice soporta múltiples lenguajes (Java, C#, C++, ObjectiveC, Python, Ruby, PHP, etc.) y multiplataforma (Windows, GNU/Linux, Solaris, Mac OS X, Android, IOS, etc.) lo que proporciona una gran flexibilidad para construir sistemas muy heterogéneos o integrar sistemas existentes. Además ofrece *servicios comunes* muy valiosos para la propagación de eventos, persistencia, tolerancia a fallos, seguridad, etc.

Algunas ventajas importantes de Ice:

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

Tanto el cliente como el servidor se pueden ver como una mezcla de código de aplicación, código de bibliotecas, y código generado a partir de la especificación de las interfaces remotas. El núcleo de Ice contiene el soporte de ejecución para las comunicaciones remotas en el lado del cliente y en el del servidor. De cara al desarrollador, dicho núcleo se corresponde con un determinado número de bibliotecas con las que la aplicación puede enlazar. El desarrollador utiliza el API para la gestión de tareas administrativas, como por ejemplo la inicialización y finalización del núcleo de ejecución.

Especificación de interfaces

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

Los middlewares permiten especificar la interfaz mediante un lenguaje de programación de estilo declarativo. A partir de dicha especificación, un compilador genera código que encapsula toda la lógica necesaria para (des)serializar los mensajes específicos produciendo una representación externa canónica de los datos.

A menudo el compilador también genera «esqueletos»2 para el código dependiente del problema. El ingeniero únicamente tiene que rellenar ese esqueleto con la implementación concreta.

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

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

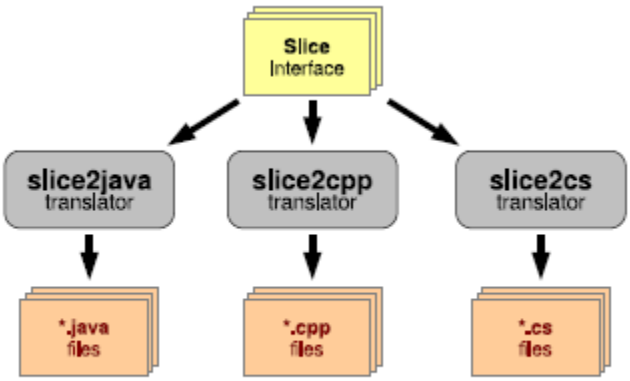


FIGURA 1.2: Ice proporciona compiladores para cada uno de los lenguajes soportados

Terminología

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

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:

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

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

Objetos

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

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

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

Proxies

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

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

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

Información de direccionamiento que permite al núcleo de ejecución de la parte del cliente contactar con el servidor correcto. Información asociada a la identidad del objeto que identifica qué objeto particular es el destino de la petición en el servidor. Información sobre el identificador de faceta opcional que determina a qué faceta del objeto en concreto se refiere el proxy.

Sirvientes (*servants*)

Como se comentó anteriormente, un objeto Ice es una entidad conceptual que tiene un tipo, una identidad, e información de direccionamiento. Sin embargo, las peticiones de los clientes deben

terminar en una entidad de procesamiento en el lado del servidor que proporcione el comportamiento para la invocación de una operación. En otras palabras, una petición de un cliente ha de terminar en la ejecución de un determinado código en el servidor, el cual estará escrito en un determinado lenguaje de programación y ejecutado en un determinado procesador.
El componente en la parte del servidor que proporciona el comportamiento asociado a la invocación de operaciones se denomina sirviente. Un sirviente encarna a uno o más objetos distribuidos. En la práctica, un sirviente es simplemente una instancia de una clase escrita por un el desarrollador de la aplicación y que está registrada en el núcleo de ejecución de la parte del servidor como el sirviente para uno o más objetos. Los métodos de esa clase se corresponderían con las operaciones de la interfaz del objeto Ice y proporcionarían el comportamiento para dichas operaciones.
Un único sirviente puede encarnar a un único objeto en un determinado momento o a varios objetos de manera simultánea. En el primer caso, la identidad del objeto encarnado por el sirviente está implícita en el sirviente. En el segundo caso, el sirviente mantiene la identidad del objeto con cada solicitud, de forma que pueda decidir qué objeto encarnar mientras dure dicha solicitud.

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

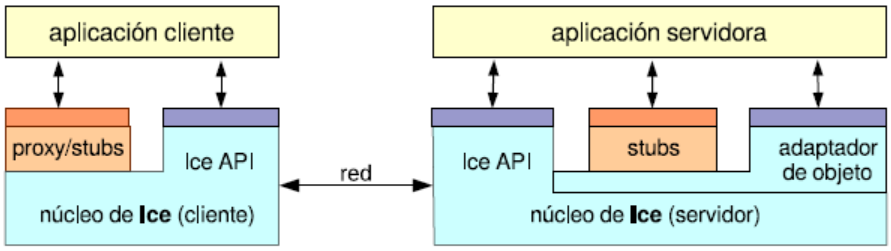


FIGURA 1.3: Componentes básicos del middleware

Semántica *at-most-once*

Las solicitudes Ice tienen una semántica *at-most-once*: el núcleo de comunicaciones hace todo lo posible para entregar una solicitud al destino correcto y, dependiendo de las circunstancias, puede volver a intentar una solicitud en caso de fallo. Ice Garantiza que entregará la solicitud o, en caso de que no pueda hacerlo, informará al cliente con una determinada excepción. Bajo ninguna circunstancia una solicitud se entregará dos veces, es decir, los reintentos se llevarán a cabo sólo si se conoce con certeza que un intento previo falló.
Esta semántica es importante porque asegura que las operaciones que no son idempotentes puedan usarse con seguridad. Una operación idempotente es aquella que, si se ejecuta dos veces, provoca el mismo efecto que si se ejecutó una vez. Por ejemplo, x = 1; es una operación idempotente, mientras que x++; no es una
operación idempotente.

Métodos de entrega

Invocaciones en una sola dirección (*oneway*)

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

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

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

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

Invocaciones en modo datagrama

Este tipo de invocaciones mantienen una semántica best-efort similar a las invocaciones oneway. Sin embargo, requieren que el mecanismo de transporte empleado sea orientado a datagramas (UDP).
Estas invocaciones tienen las mismas características que las invocaciones oneway, pero abarcan un mayor número de escenarios de error:

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

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

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

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

Excepciones

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

«Hola mundo» distribuido

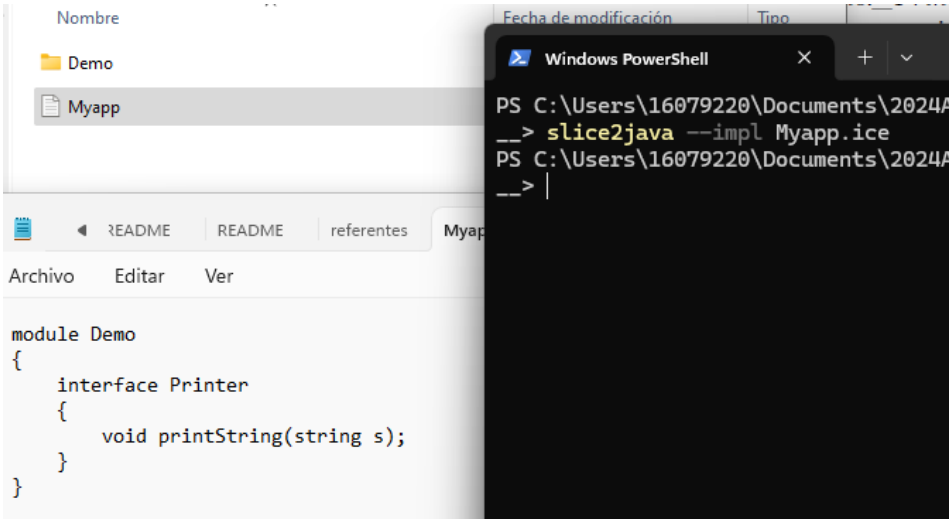
En esta primera aplicación, el servidor proporciona un objeto que dispone de un único método remoto llamado printString (). Este método imprime en la salida estándar del servidor la cadena que el cliente le pase como parámetro. Tal como hemos visto, lo primero que necesitamos es escribir la especificación de la interfaz remota para estos objetos. El siguiente listado corresponde al fichero Myapp.ice y contiene la interfaz *Printer* en lenguaje Slice.

```
module Demo
{
    interface Printer
    {
        void printString(string s);
    }
}
```

Nota: el archivo debería tener el mismo nombre de la interfaz. Debería llamarse Printer.ice, para efectos ilustrativos la hemos llamado diferente.

Lo más importante de este fichero es la declaración del método printString (). El compilador de interfaces generará los stubs que incluyen una versión básica de la interfaz *Printer* en el lenguaje de programación que el programador decida. Cualquier clase que herede de esa interfaz *Printer* debería redefinir (especializar) un método printString (), que podrá ser invocado remotamente, y que debe tener la misma signatura. De hecho, en la misma aplicación distribuida puede haber varias implementaciones del mismo interfaz en una o varias computadoras y escritos en diferentes lenguajes.
El compilador también genera los «cabos» para el cliente.
Los clientes escritos en distintos lenguajes o sobre distintas arquitecturas podrán usar los objetos remotos que cumplan la misma interfaz. Para generar los stubs de cliente y servidor en Java (para este ejemplo) usamos el *translator* slice2java. Esto lo hace gradle automáticamente y genera un paquete cuyo nombre se corresponde con el del módulo que hay en el fichero Myapp.ice (en este caso Demo). Dentro se encuentran las clases que se usarán tanto en el cliente como en el servidor, pero su contenido exacto no nos interesa ahora mismo.

Para efectos de ilustración puedes copiar el archivo Myapp.ice a una carpeta separada y ejecutar el comando: slice2java Myapp.ice
Verifica el contenido de la carpeta demo



Sirviente

El compilador de interfaces genera la clase *Demo.Printer*. La implementación del *sirviente* debe heredar de esa interfaz, proporcionando una implementación (por sobrecarga) de los métodos especificados en la interfaz Slice. El propio compilador de interfaces puede generar una clase «hueca» que sirva al programador como punto de partida para implementar el sirviente:

Al ejecutar:
slice2java --impl Printer.ice

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

```
package Demo;

public final class PrinterI implements Printer
{
    public PrinterI()
    {
    }

    @Override
    public void printString(String s, com.zeroc.Ice.Current current)
    {
        System.out.println(message);
    }
}
```

La única modificación relevante respecto al fichero generado está en la línea System.out.println(message);.

Servidor

Nuestro servidor consiste principalmente en la implementación de una clase que hace uso de los servicios y herramientas proporcionadas por ICE:

```
public class Server
{
    public static void main(String[] args)
    {
        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectAdapter adapter =
communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
            com.zeroc.Ice.Object object = new MyappI();
            adapter.add(object, com.zeroc.Ice.Util.stringToIdentity("SimplePrinter"));
            adapter.activate();
            communicator.waitForShutdown();
        }
    }
}
```

Inicializamos el Communicator de Ice llamando a **Ice.Util.initialize**. Pasamos argumentos a esta llamada porque el servidor puede tener una línea de comandos argumentos que son de interés para el communicator; para este ejemplo, el servidor no requiere ningún argumento de línea de comandos.

Creamos un adaptador de objetos llamando a **createObjectAdapterWithEndpoints** en la instancia del communicator. Los argumentos que pasamos son "SimplePrinterAdapter" (que es el nombre del adaptador) y "default -p 10000", que indica al adaptador que escuche las llamadas entrantes. solicitudes utilizando el protocolo predeterminado (TCP/IP) en el puerto número 10000.

En este punto, se inicializa ICE y creamos un servidor. Para nuestra interfaz de Impresora creamos una instancia de un objeto MyappI. Informamos al adaptador (adapter) de la presencia de un nuevo sirviente (servant), se registra el sirviente en el adaptador mediante el método add(); los argumentos de add() son el servant que acabamos de crear, más un identificador. En este caso, la cadena "SimplePrinter" es el nombre del servant. (Si tuviéramos varias impresoras, cada una tendría un nombre diferente o, más correctamente,
una identidad de objeto diferente.)

A continuación, activamos el adaptador llamando a su método de actívate(). El adaptador se crea inicialmente en un estado de retención; esto es útil si tenemos muchos servants que comparten el mismo adaptador y no quieren que se procesen solicitudes hasta que se hayan creado instancias de todos los servants.

Finalmente, llamamos a waitForShutdown. Esta llamada suspende el hilo principal. hasta que finalice la implementación del servidor, ya sea haciendo una llamada para detener el tiempo de ejecución, o en respuesta a una señal. (Por ahora, simplemente interrumpiremos el servidor en la línea de comando cuando ya no lo necesitemos.)

Tenga en cuenta que ese código es esencialmente lo mismo para todos los servidores. Puedes poner ese código en una clase auxiliar y, posteriormente, no tendrá que volver a implementarla de nuevo.

En términos prácticos el adaptador requiere un enpoint —un punto de conexión a la red materializado por un protocolo (TCP o UDP), un host y un puerto. En este caso esa información se extrae de un fichero de configuración a partir del nombre del adaptador (PrinterAdapter).

Cliente

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

```
public class Client
{
    public static void main(String[] args)
    {
        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectPrx base = communicator.stringToProxy("SimplePrinter:default -p 10000");
            Demo.PrinterPrx printer = Demo.PrinterPrx.checkedCast(base);
            if(printer == null)
            {
                throw new Error("Invalid proxy");
            }
            printer.printString("Hola Mundo!!");
        }
    }
}
```

Tenga en cuenta que el diseño general del código es el mismo que para el servidor: usamos el mismo try{}catch{} para lidiar con errores.

Inicializamos ICE con un llamando a Ice.Util.initialize. El siguiente paso es obtener un proxy para la impresora remota. Creamos un proxy llamando a stringToProxy en el communicator, con la cadena

("SimplePrinter:default -p 10000"). Tenga en cuenta que la cadena contiene la identidad del objeto y el número de puerto que fueron utilizados por el servidor.

El proxy devuelto por **stringToProxy** es de tipo **Ice.ObjectPrx,** que está en la raíz del árbol de herencia para interfaces y clases. Pero para hablar realmente con nuestra impresora, necesitamos un proxy para una interfaz de Impresora, no una interfaz de Objeto. Para hacer esto, necesitamos hacer un downcast llamando a **PrinterPrxHelper.checkedCast.**

Un checkedCast envía un mensaje al servidor, preguntando efectivamente "¿Es este un proxy para una interfaz de impresora?" Si es así, la llamada devuelve un proxy. de tipo Demo.Printer; de lo contrario, si el proxy denota una interfaz de algún otro tipo, la llamada devuelve nulo.

Probamos que el downcast tuvo éxito y, si no, arrojamos un mensaje de error que finaliza el cliente.