Tema 4.
Sistemas distribuidos
Middleware. Modelos de interacción.
ICE – Internet Communications Engine

Marisol García Valls

Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
mvalls@it.uc3m.es

Arquitectura de sistemas II

Grado en Ingeniería Telemática
Curso: 3º, cuatrimestre: 2º

Universidad Carlos III de Madrid

Índice

Bibliografía

- Manual de Ice 3.5.1
  http://download.zeroc.com/Ice/3.5/Ice-3.5.1.pdf

Capítulo 1.1 "Ice overview"

Capítulo 1.2. "Writing a HelloWorld application"

1.2.1 "Writing a Slice definition"

1.2.2 "Writing an Ice application in C++"

Lecturas no exhaustivas (consultivas):

- En el libro anterior: "The Slice language"
- S. B. Lippman, J. Lajoie, B. E. Moo.
  "C++ Primer". 5th Edition. Addison-Wesley. 2013.

Middleware

Síncronos:

- RPC: remote procedure call
- Orientado a objetos e invocaciones remotas (RMI de Java o ICE)
- Componentes (CORBA)

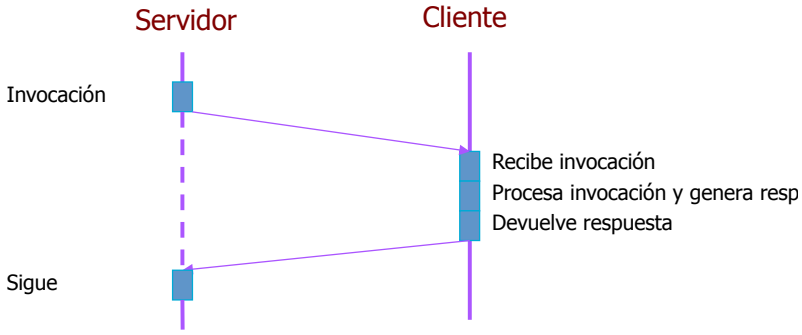
¡ Clasificación no estricta !

Asíncronos:

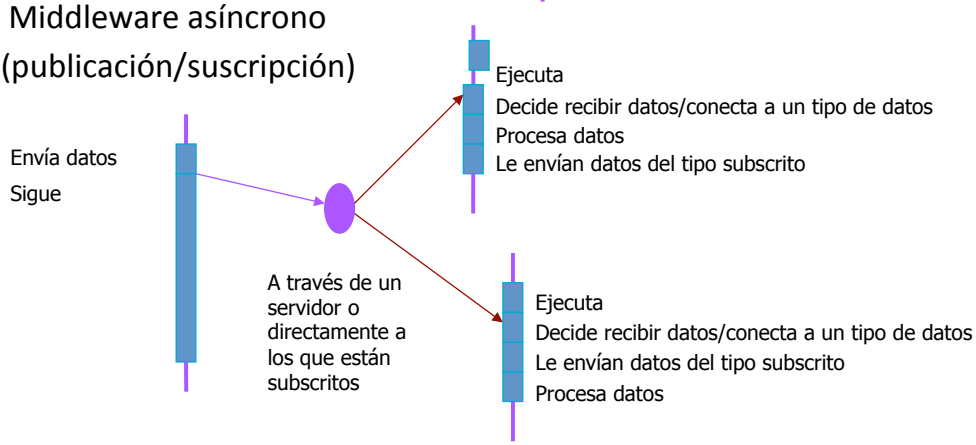
- Orientado a mensajes (JMS)
- Publicación/suscripción (ICE, DDS)

Síncrono vs. Asíncrono

- Middleware síncrono (invocación remota)



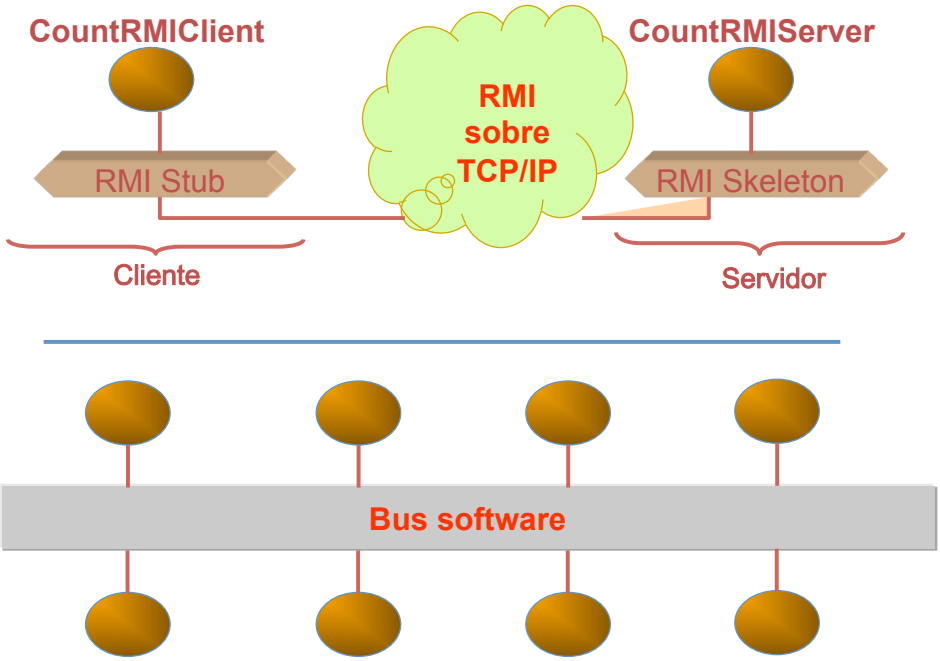
- Middleware asíncrono (publicación/suscripción)



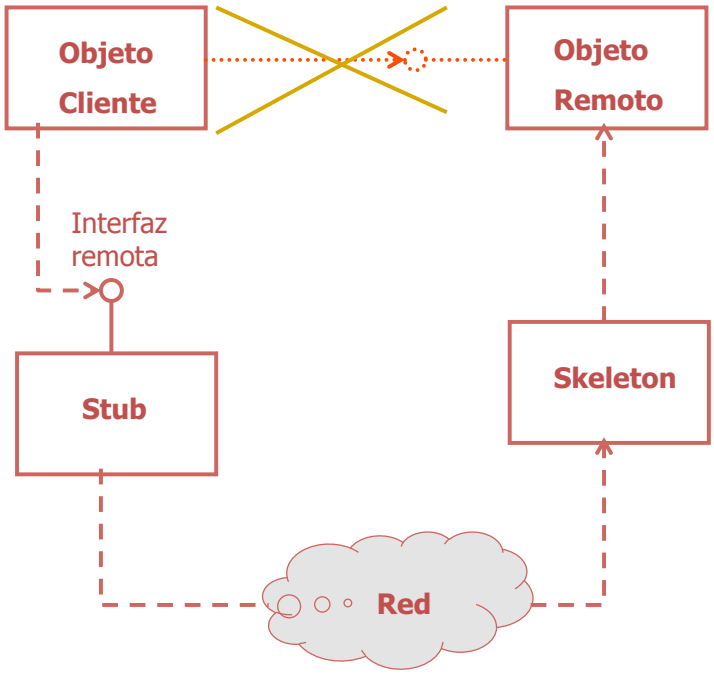
Middleware orientado a objetos

- Permite realizar invocaciones a métodos remotos (invocaciones remotas).
  - En un programa centralizado las invocaciones entre objetos son locales.
- En un programa distribuido existen objetos distribuidos y puede haber invocaciones locales y remotas.
- Los **objetos remotos** son aquellos que exponen parte de sus funciones o métodos como una interfaz remota:
  - Tienen métodos remotos que **pueden ser invocados** por objetos cliente.
- Conceptos clave en el modelo distribuido son:
  - Referencia a objeto remoto e interfaz remota
- Las invocaciones concurrentes sobre un objeto distribuido son posibles y pueden ocasionar condiciones de carrera.
- Un objeto distribuido que pueda sufrir invocaciones concurrentes deberá proteger su estado adecuadamente.

Invocación de métodos remotos. **Ej. RMI de Java**



Stubs y Skeletons



Transferencia de objetos: serialización

- La **serialización** (***serialization***) es un mecanismo que permite convertir un objeto o una instancia en un **flujo de datos para ser almacenado**.
  - Implica la traducción de estructuras de datos o estado de instancias u objetos a un formato que puede ser almacenado en un fichero, buffer de memoria, o transmitido por la red.

- Esto permite implementar persistencia.

- **Deserialización** (***deserialization***) es el proceso que permite que una instancia almacenada como flujo de datos (*data stream*) sea reconstituida.
  - Una instancia serializada puede ser reconstituida después en la misma u otra máquina.
  - Se crea un **clon semánticamente idéntico al objeto original**.

- Problemas a resolver:
  - Ordenamiento de bytes (*endianness*)
  - Organización de la memoria
  - Representaciones diferentes de las estructuras de datos en diferentes lenguajes de programación

    - Se requiere un formato de almacenamiento independiente de la arquitectura.

Serialización: detalles de rendimiento

- La **codificación** de los datos es **secuencial**.
  - La extracción de una parte de los datos requiere que el objeto completo sea leído de principio a fin y sea reconstruido.
- A veces este mecanismo simple es una ventaja:
  - Es simple, permite usar interfaces I/O comunes para pasar el estado de una instancia

- En aplicaciones que requieren más alto rendimiento es conveniente:
  - Mecanismo más complejo, con una **organización no lineal** del almacenamiento.

- Almacenamiento de **punteros**:
  - Es muy sensible – las instancias a las que apuntan pueden cargarse en una zona diferente de la memoria
  - *Swizzling* – convertir punteros dependientes de zonas específicas de memoria a símbolos o posiciones portables. (remover)
  - *UNswizzling* – convertir referencias basadas en nombres o posiciones a referencias (punteros) directas.

- **Ejecución diferencial** basada en código común de serialización
  - Detectar diferencias entre objetos a serializar y sus copias anteriores
  - Obtener realimentación para la siguiente detección de diferencias (*on the fly*)

Diferencias entre serialización y marshalling

- **Marshalling** es un concepto similar a la serialización (p.ej., Python) aunque no para todos los entornos (p.ej., Java RFC 2713).

- Hacer un marshal de un objeto implica guardar su estado y la dirección base de su código (*codebase*).

- **Unmarshalling** consiste en obtener una copia del objeto original posiblemente descargando automáticamente las definiciones de clase del objeto.
  - Unmarshalling trata a los objetos remotos de forma diferente (RFC 2713).

- NOTA: Se escribe indistintamente: marshalling o marshaling.

Paso de Parámetros a Métodos Remotos

Hay **3 mecanismos básicos**

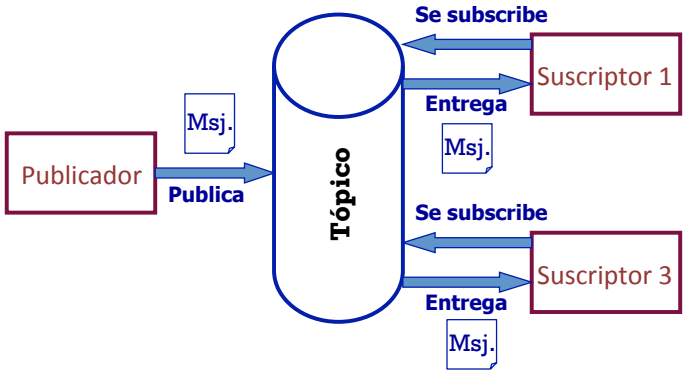
1. **Tipos primitivos**: se pasan por valor (copia). Todos son serializables
2. **Objetos Remotos**: se pasan por referencia (stubs, usados para invocar métodos remotos).
3. **Objetos Locales**: se pasan por valor (sólo si son serializables), se crea un nuevo objeto en el nodo que recibe la copia.

Middleware de publicación/suscripción

- Generalmente los publicadores (P) y los subscriptores (S) son **anónimos**.
  - Los **datos** se **clasifican** por "**tópicos**"
- Las aplicaciones (nodos) **se suscriben** a los **datos** que necesitan: **suscriptores**.
- Las aplicaciones **publican** los **datos** que quieren compartir: **publicadores**.
  - Los publicadores envían un mensaje a un **tópico** o tema (*topic*).
- Publicadores y subscriptores pueden **publicar o subscribirse** a la jerarquía de contenidos **de forma dinámica**.
- El **sistema se encarga de distribuir los mensajes** que llegan desde los múltiples publicadores de un tópico a los múltiples subscriptores de ese tópico.
- Los **tópicos retienen los mensajes** sólo **durante el tiempo que se tarda en distribuirlos** a los subscriptores actuales.
- **Implementaciones** de PS:
  - A través de un **servidor central** (**JMS**)
  - Los datos pasan **directamente** entre los publicadores y los suscriptores (**DDS**)

Mensajería con esquemas publicador-suscriptor

- Cada mensaje puede tener **múltiples consumidores**.
- Publicadores y subscriptores **tienen dependencias temporales**.
  - Un cliente que se subscribe a un tópico sólo puede consumir mensajes publicados después de que el cliente haya creado la subscripción,
  - El subscriptor debe estar activo para poder consumir mensajes



- Se usa en situaciones en las que cada mensaje pueda ser procesado por cero, uno o muchos consumidores.

El middleware de comunicaciones ICE

- Orientado a objetos.
- Interfaz de programación multi-lenguaje: C++, C#, Java, etc.
- Sigue el modelo cliente-servidor (existen peticionarios y servidores de peticiones).
- Clientes
  - entidades activas que generan peticiones de servicio.
- Servidores
  - entidades pasivas que ofrecen servicio en respuesta a peticiones.
- Ice proporciona un protocolo RPC que puede usar TCP/IP o UDP como transportes.
  - También permite usar SSL como transporte para encriptar las comunicaciones.

Tipos de invocación soportados

- Síncrona
- Asíncrona
- Unidireccional (*one way* method invocation)
  - Requieren TCP/IP o SSL
- Datagramas
  - Requieren UDP
- Lote unidireccional (*batched one way* method invocation)
- Lote datagrama (*batched datagram* method invocation)
- Adicionalmente Ice Storm ofrece publicación-suscripción

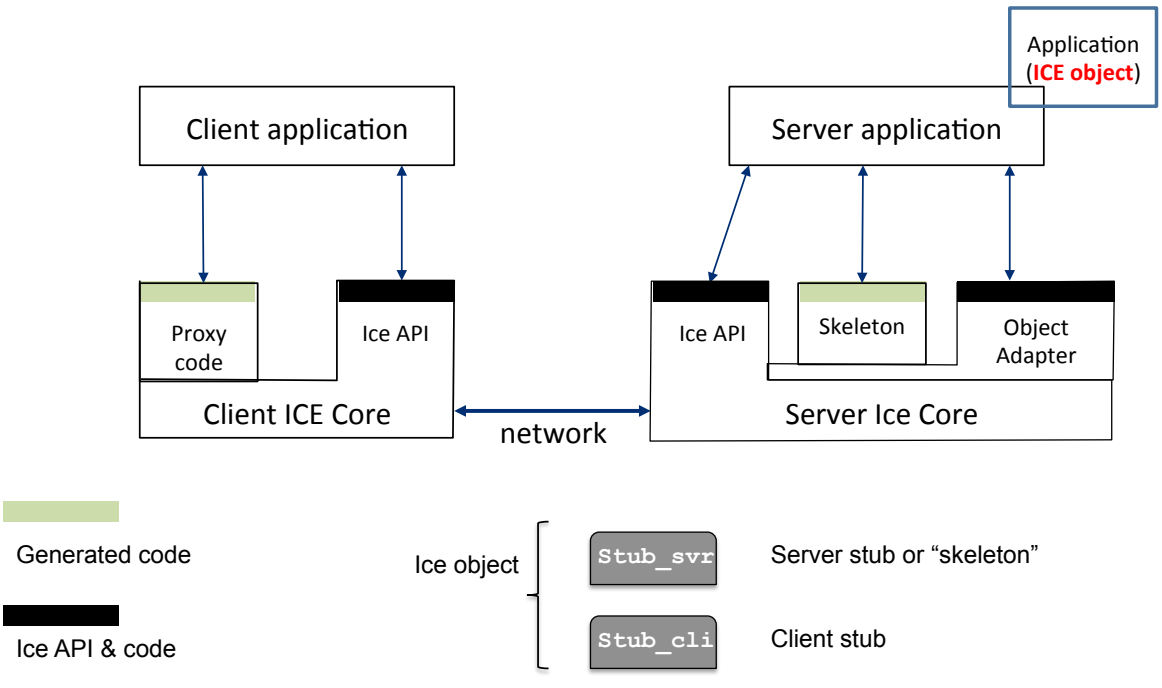
Conceptos

- Objectos ICE
  - Abstracción (local o remota) que responde a peticiones de clientes,
  - Pueden ser instanciados en uno o varios servidores,
  - Sus operaciones tienen de 0 a n parámetros (de entrada se inicializan en cliente y se pasan al servidor; los parámetros de salida se inicializan en el servidor)
  - Cada objeto ICE tiene una identidad única.

- Proxy
  - Representante de un objecto Ice en el lado del cliente.

- Sirviente (*servant*)
  - Encarna un objeto Ice (o varios)
  - Instancia de una clase escrita por el desarrollador del servidor
  - Implementa las operaciones remotas definidas en alguna interfaz
  - Registrado en el lado servidor para uno o más objetos Ice.

Estructura cliente-servidor



Beneficios arquitectónicos

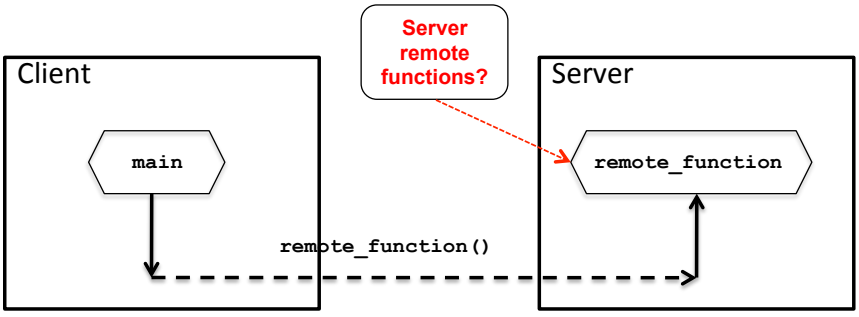
- Orientación a objetos
- Comunicación síncrona y asíncrona
- Soporte para múltiples interfaces
- Independencia del lenguaje
- Independencia de la máquina
- Independencia de la implementación
- Independencia del sistema operativo
- Soporte para concurrencia
- Independencia del transporte
- Transparencia de servidor y de ubicación
- Seguridad
- Persistencia inherente
- Disponibilidad del código fuente

La interfaz remota

- Un objeto Ice tiene funciones que pueden ser invocadas remotamente.
  - También puede tener otras funciones no remotas.

- ¿**Cómo indica el servidor** cuáles de sus **funciones** son **remotas**?

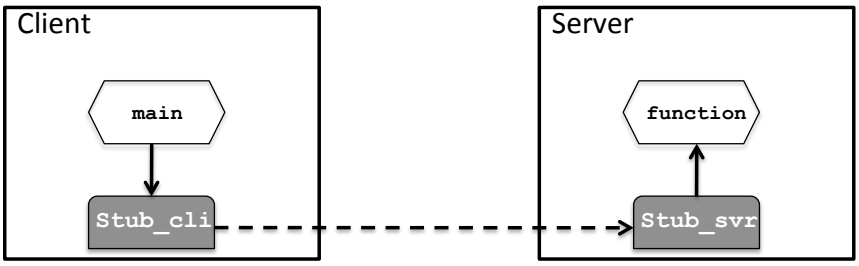
- Las funciones remotas se deben indicar en una interfaz.
- La interfaz se programa en un lenguaje independiente, **IDL** (Interface Definition Language).



IDL – Interface definition language

- Un IDL es un lenguaje que define la comunicación entre clientes y servidores que usan middleware basado en RPC o invocaciones a métodos.

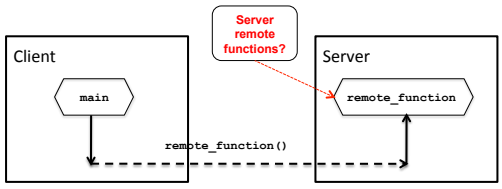
- Con IDL se especifica la interfaz entre cliente y servidor.
- Luego el middleware (compilador de ICE) crea los representantes (stubs).
- Los stubs ejecutan las llamadas remotas, lo que incluye los pasos de [de-]serialización o [un]marshaling.



Slice – Lenguaje IDL de Ice

- El primer paso para implementar una aplicación Ice es escribir la **definición Slice**.
- La definición Slice contiene las interfaces que son utilizadas por la aplicación.

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



- La interfaz se guarda como **`Printer.ice`** .

- En este caso la definición Slice consiste en:
  - El módulo **`Demo`** que contiene una sola interfaz llamada **`Printer`**.
  - La interfaz **`Printer`** es muy simple y contiene una sola función de nombre **`printString`**.

Compilación de la interfaz Slice

- La compilación:

```
$ slice2cpp Printer.ice
```

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

- El compilador **slice2cpp** produce dos ficheros a partir de esta definición:

- **Printer.h** con las definiciones de tipo C++ a partir del código Slice para la interfaz Printer.
  - Se debe incluir este fichero de cabecera en cliente y servidor.

- **Printer.cpp** con el **código fuente** para la interfaz Printer.
  - Es código específico para el *run-time* de cliente y servidor (información/datos de serialización de parámetros, y deserialización).
  - Debe ser compilado y enlazado en el cliente y servidor.

Aplicación Ice con C++

- Una aplicación Ice consta de una parte servidor, otra cliente y la interfaz Slice.
- Ambas pueden ser escritas en el mismo lenguaje de programación o en lenguajes diferentes.
  - La interfaz Slice define el tipo de invocación y parámetros para la comunicación.

servidor

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI: public Printer {

…
};

//implementar función printString

int main(…){
 //crear e inicializar objeto Ice
 // …

}
```

cliente

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

int main(…){
 //conseguir referencia a objeto Ice remoto
 //invocar función remota

}
```

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

interfaz

Servidor (`server.cpp`)

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI : public Printer {
public:
    virtual void printString(const string& s, const Ice::Current&);
};

void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =

ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -
p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();                          Cuerpo principal
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
```

- La mayoría es código común.
  - Definiciones del *run-time* de Ice.
  - Definiciones C++ para la interfaz Printer.
- Importar contenidos del espacio de nombres de `std` y `Demo`.
  - Evita código como `std::cout << s << end;`
- Se implementa un sirviente (*servant*) de tipo `PrinterI` que hereda de `Printer`.
  - Convención: usar "`I`" al final del nombre
  - Indica que esa clase implementa una interfaz (el esqueleto generado por el compilador de Ice)
- La clase sirviente debe implementar la función del interfaz: `printString`.
  - Es virtual dentro del esqueleto generado por Ice.

Servidor (`server.cpp`)

```
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =

ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -
p 10000");                                              Cuerpo principal
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

- `status`, estado de salida del programa.
- `ic`, referencia al run-time de Ice (`ice communicator`)
- Bloque `try`, código del servidor.
- `catch` para detectar dos tipos de excepciones y así no dañar la pila hasta alcanzar el main:
  - Se devuelve un error al sistema operativo.
- Código de limpieza
  - `destroy` del *communicator*, si éste fue inicializado.

Servidor (`server.cpp`)

```
int status = 0;
Ice::CommunicatorPtr ic;
try {
    ic = Ice::initialize(argc, argv);
    Ice::ObjectAdapterPtr adapter =
ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -
p 10000");
    Ice::ObjectPtr object = new PrinterI;
    adapter->add(object, ic->stringToIdentity("SimplePrinter"));
    adapter->activate();
    ic->waitForShutdown();
```

- Inicialización del run-time (`initialize`).
  - Devuelve un puntero al run-time para manejarlo.
- Crear un objeto adaptador (`createObjectAdapterWithEndpoints`).
- Argumentos del adaptador:
  - Nombre (`SimplePrinterAdapter`)
  - Escucha peticiones entrantes usando el protocolo por defecto (TCP/IP) en el puerto número `10000`.
- El run-time del lado servidor está inicializado
  - Se puede crear ya un sirviente.
- Se crea un sirviente para la interfaz `Printer` instanciando un objeto `PrinterI`.
- Informar al adaptador de la presencia de un nuevo sirviente (`add`).
  - El sirviente se llama "`SimplePrinter`"
  - Si hubiera más sirvientes para un adaptador, cada uno debe tener un nombre diferente.
- Se activa el adaptador (`activate`).
- Se suspende la hebra que lo invoca hasta que la implementación del servidor termina
  - Por una llamada a parar el run-time o por una señal.

Cliente (`client.cpp`)

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
    Ice::ObjectPrx base = ic->stringToProxy("SimplePrinter:default
-p 10000");
    PrinterPrx printer = PrinterPrx::checkedCast(base);
    if (!printer)
        throw "Invalid proxy";
        printer->printString("Hello World!");
    } catch (const Ice::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Implementación del cliente

- La mayoría es código común al servidor.
  - Definiciones del *run-time* de Ice.
  - Definiciones C++ para la interfaz Printer.
  - Bloques `try` y `catch` capturan las mismas excepciones.
- Importar contenidos del espacio de nombres de `std` y `Demo`.
  - Evita código como `std::cout << s << end;`
- Inicializar el run-time de Ice en el cliente.
- Obtener un proxy al objeto Ice remoto con función `stringToProxy` sobre el comunicator.
  - El string contiene la identidad y el número de puerto utilizados por el servidor "`SimplePrinter:default -p 10000`"
- El proxy devuelto por `stringToProxy` es de tipo `Ice::=ObjectPrx`.
  - `Ice::ObjectPrx` es el tipo raíz en el árbol de herencia para interfaces y clases.
- Casting es necesario con `PrinterPrx:checkedCast`
  - Necesitamos un proxy a una interfaz `Printer` y no `Object`.
  - Pregunta al servidor "¿es una interfaz Printer?".
  - Si lo es, devuelve un proxy de ese tipo, si no lo es devuelve un proxy nulo.
- Se comprueba si el casting es exitoso.
- Invocación de la función remota.

Compilación y ejecución

Servidor: compilación y enlazado:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Server.cpp
$ c++ -o server Printer.o Server.o -L$ICE_HOME/lib -lIce -lIceUtil
```

Cliente: compilación y enlazado:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp
$ c++ -o client Printer.o Client.o -L$ICE_HOME/lib -lIce -lIceUtil
```

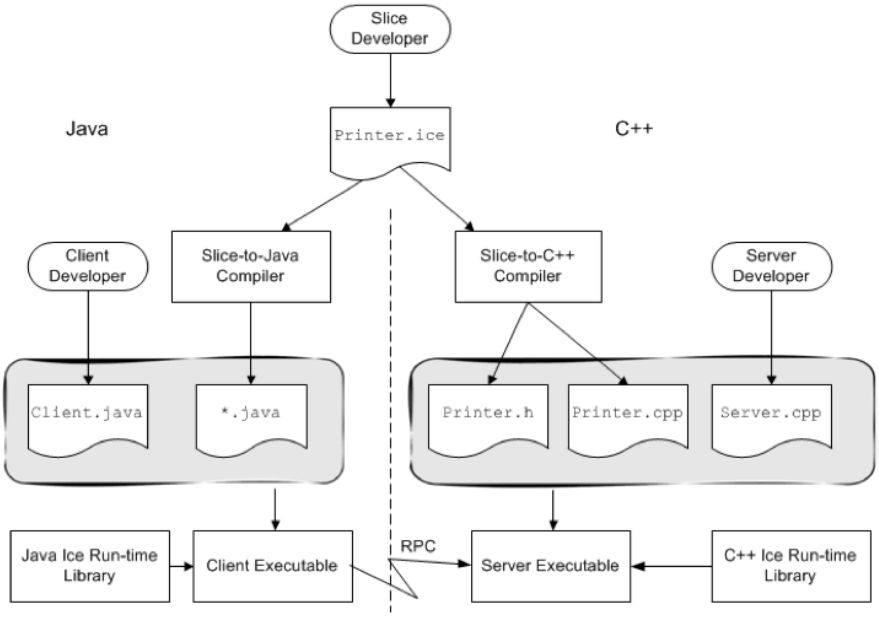
Arranque ordenado (terminales separados):

```
$ ./server
$ ./client
```

Ejemplo de error (cliente lanzado sin previa ejecución del servidor):

```
Network.cpp:471: Ice::ConnectFailedException:
connect failed: Connection refused
```


Ice – Different development environments



Ice::Current object

- Se pasa a cada operación del skeleton generado en el lado servidor.
- Su definición es:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Idempotent };

    local struct Current {
        ObjectAdapter   adapter;
        Connection      con;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
        EncodingVersion encoding;
    };
};
```

- Permite que la petición actual pase información a la implementación de la operación que la sirve en el servidor.

Servidor (server.cpp)

```
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter =

ic->createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -
p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

- `status`, estado de salida del programa.
- `ic`, referencia al run-time de Ice (`ice communicator`)

Cuerpo principal

- Bloque `try`, código del servidor.
- `catch` para detectar dos tipos de excepciones y así no dañar la pila hasta alcanzar el main:
  - Se devuelve un error al sistema operativo.
- Código de limpieza
  - `destroy` del *communicator*, si éste fue inicializado.