Tema 4: INVOCACIÓN REMOTA

E. U. Informática en Segovia
Departamento de Informática
Universidad de Valladolid

4.1 Introducción

Invocación remota: integra programas con arquitectura cliente/servidor y lenguajes de programación

Invocación remota (IR) modelada a imagen de invocación local (IL). Diferencia algún módulo ejecutado remotamente

Módulo = servicio

- Exporta procedimientos y métodos que
- Proveen un conjunto de operaciones sobre
- Una clase de recursos con
- Interfaz que especifica accesibilidad de operaciones y variables

IR proporciona a aplicaciones:

- Un modelo de programación a alto nivel
- Transparencia de ubicación
- Independencia de protocolos, sistemas operativos, hardware...

4.1.1. Interfaz de la invocación remota

Diferencias IR - IL:

- Argumentos de entrada siempre por valor (referencia requiere IDL)
- Sin paso de punteros (espacios de direcciones no coincidentes)
- Un módulo de un proceso no accede a las variables de otro módulo de otro proceso (encapsulamiento)

Definición de la interfaz:

- Integrar el interfaz en el lenguaje: Java RMI
- Utilizar un lenguaje de definición de interfaz (IDL):
  - Sun RPC, que define un IDL para RPC (*Remote Procedure Call*)
  - Corba IDL, que define un IDL para RMI (*Remote Method Invocation*)

La interfaz define:

- Los nombres de los métodos
- Argumentos: tipos y movimientos (de entrada, salida o entrada/salida)

4.1.2. Diseño de la invocación remota

En IL tiempo de espera por respuesta ilimitado ⇒ inaceptable en IR

Semántica basada en un protocolo petición-respuesta. Diferentes fuentes de error (el mensaje de petición no llega, el mensaje de respuesta no se recibe, el resultado de la operación se modifica si se reejecuta, ...)

Diferentes alternativas para tratar los posibles errores:

- Uso de temporizadores + reenvíos (en origen), para evitar pérdida de mensajes petición
- Reejecución (en destino), para evitar pérdida de mensajes de petición (relativas a operaciones idempotentes, es decir, aquellas en las que no se altera el resultado si se reejecutan)
- Mantenimiento historial peticiones + filtrado + retransmisión (en destino), para evitar reejecución de operaciones no idempotentes

Tres tipos de semántica de IR:

Semántica de IR	Reintento	Filtrado	Reejecución
tal-vez	No	-	-
al-menos-una-vez	Sí	No	Reejecución
como-mucho-una-vez	Sí	Sí	Retransmisión

- Semántica tal-vez (maybe): el cliente espera un rato y sigue con su procesamiento normal al cabo de un tiempo sin saber qué ocurrió:
  - Se cayó el servidor
  - No llegó la solicitud
  - Se perdió la respuesta

  No es aceptable

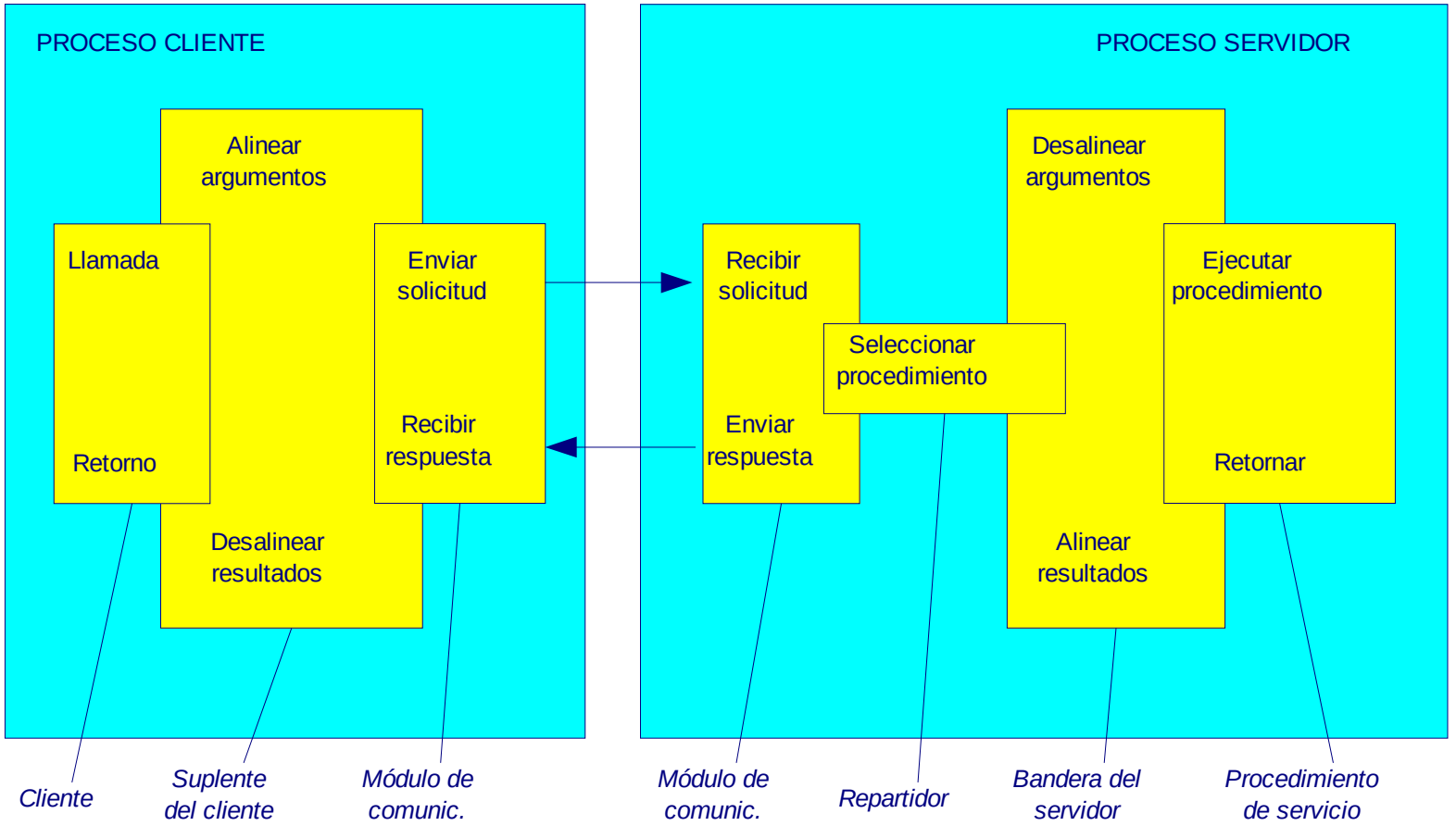
- Semántica al-menos-una-vez (at-least-once): el cliente reintenta tras temporización. El servidor no filtra duplicados. El cliente recibe una o más respuestas. Sólo válido para operaciones idempotentes (la reiteración de la misma operación no altera el resultado final)
  - Ejemplo operaciones idempotentes: Sumar(a, b); UnionConjuntos(A,B)
  - Ejemplo operaciones no idempotentes: Concatenar(lista1, lista2)

- Semántica como-mucho-una-vez (at-most-once): cliente reintenta tras temporización. El servidor filtra duplicados servicio de historial en servidor

  El proceso en el servidor se puede ejecutar una o cero veces:
  - Una vez: el servidor no cayó y el cliente recibió una respuesta
  - Cero veces: el cliente recibe algún tipo de informe de error

4.2. Llamada a Procedimientos Remotos

4.2.1. Características generales



Esquema general de Llamada a Procedimientos Remotos (RPC)

Componentes necesarios:

- Procedimiento suplente (*stub* del cliente):
  - Alinear los argumentos de entrada
  - Enviar la solicitud
  - Esperar la respuesta
  - Desalinear los argumentos de salida
- Procedimiento bandera (*stub* del servidor):
  - Desalinear los argumentos de entrada
  - Llamar al procedimiento correspondiente
  - Alinear los argumentos de salida
  - Enviar la respuesta
- Procedimiento repartidor (*dispatcher*):
  - Averigua la identidad del procedimiento remoto
  - Llama al *stub* que corresponda

Para esto: compilador de IDL (*Interface Definition Language*)

Compilador de IDL: a partir de la definición de interfaz se consiguen

- *Stubs* de cliente y servidor
- Repartidor
- Cabeceras de los procedimientos del servidor para ser usadas en el cliente
- Filtro para los argumentos que se pueden enviar (en principio, cliente y servidor pueden ser máquinas con hardware distinto)

Binding (registro): servicio que establece correspondencia nombre - identificador de comunicación

Funciones del *binder*:

- Asociar, a petición del servidor, el nombre del servicio, su puerto y su número de versión
- Buscar, a solicitud del cliente, el puerto de un servidor cuya interfaz sea de la misma versión que la del cliente

Migración del servidor ⇒ nuevo registro ⇒ localizable por clientes

Nota: migración = transferencia de un proceso en ejecución entre dos nodos

Alternativas al *binding*:

- Servidor en un puerto fijo y conocido: Dado que el cliente conoce la localización del servidor, la migración de éste requiere la recompilación del cliente
- El cliente envía solicitud por difusión: Dado que el cliente "descubre" la localización del servidor, la migración de éste no requiere ningún cambio en el cliente

Diferencias entre IL y IR en la práctica:

- Llamadas diferentes: argumentos adicionales y los de siempre hay que usarlos a través de punteros
- Notación ligeramente distinta (se verán ejemplos en Sun RPC)

Hay paquetes de usuarios que gestionan llamadas y retornos de forma análoga a las librerías de entrada/salida de Unix:

- Se ejecutan en el cliente,
- Es una capa adicional disminuye capacidad de control

4.2.2. Sun RPC

Características generales:

- Muy conocido y extendido
- En algunas cosas no sigue al pie de la letra el modelo de RPC previamente expuesto
- Diseñado para la comunicación cliente/servidor en Sun NFS, y admite llamadas a procedimientos remotos tanto sobre TCP como sobre UDP
- Dos componentes fundamentales:
  - Un lenguaje de interfaz: XDR (eXternal Data Representation)
  - Un compilador de interfaz: rpcgen

Sun XDR

Interfaz definida por:

- Un número de programa y un número de versión.
- Las definiciones de procedimientos y de los tipos asociados. Cada procedimiento:
  - Una cabecera
  - Un número
- Argumentos: varios de entrada, uno de salida

Número 0 de procedimiento reservado para comprobaciones

Admite (con sintaxis de C): constantes, estructuras, uniones, enumeraciones, typedefs y programas

```
/* calc.x */
program CALCULADORA
   version VERCALC
      long sum(long, long) = 1;
      long res(long, long) = 2;
      long mul(long, long) = 3;
      long div(long, long) = 4;
    = 1;
 = 0x20000001;
```

Vemos un ejemplo sencillo: mostrar un número en el servidor. Interfaz:

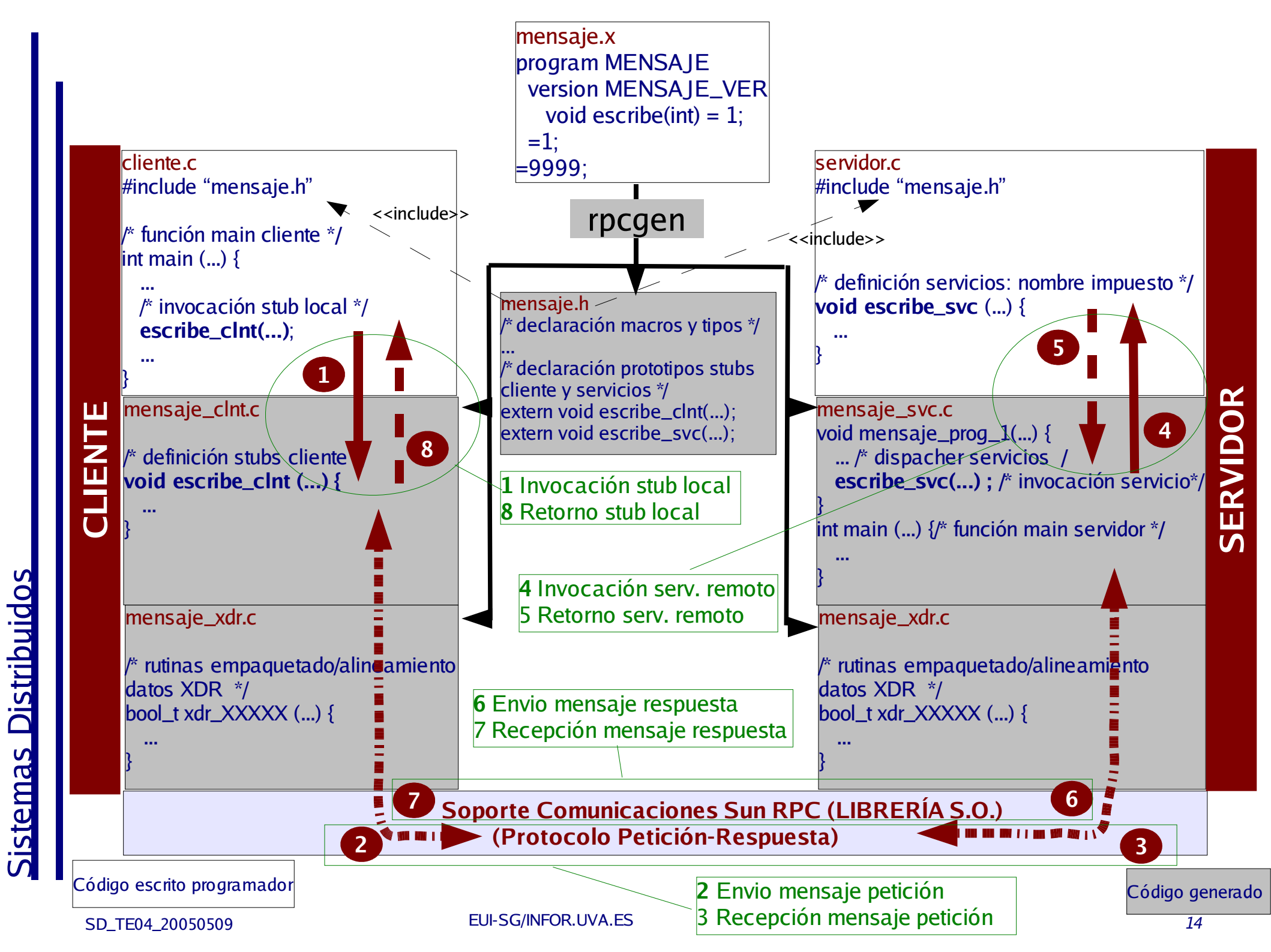
```
/* mensaje.x */
program MENSAJE
   version MENSAJE_VER
      void escribe(int) = 1;
    = 1;
 = 9999;
```

rpcgen

rpcgen compila el fichero mensaje.x y genera:

- Stub del cliente: mensaje_clnt.c
- El procedimiento main con todo el binder y el repartidor del servidor: mensaje_svc.c
- Los procedimientos de alineación y desalineación XDR: mensaje_xdr.c, de ser necesario (aquí no hace falta)
- Un fichero de cabecera mensaje.h que contiene:
  - Definiciones de constantes y tipos comunes. En este caso MENSAJE y MENSAJE_VER con el valor numérico dado
  - Cabeceras de los procedimientos a implementar

Opcionalmente, el rpcgen puede crear plantillas para el programa servidor y el cliente e incluso un Makefile


mensaje.x
program MENSAJE
version MENSAJE_VER
void escribe(int) = 1;
=1;
=9999;
rpcgen
CLIENTE
SERVIDOR
cliente.c
#include "mensaje.h"
<<include>>
/* función main cliente */
int main (...) {
...
/* invocación stub local */
escribe_clnt(...);
...
}
servidor.c
#include "mensaje.h"
<<include>>
/* definición servicios: nombre impuesto */
void escribe_svc (...) {
...
}
mensaje.h
/* declaración macros y tipos */
...
/* declaración prototipos stubs cliente y servicios */
extern void escribe_clnt(...);
extern void escribe_svc(...);
mensaje_clnt.c
/* definición stubs cliente */
void escribe_clnt (...) {
...
}
mensaje_svc.c
void mensaje_prog_1(...) {
... /* dispacher servicios */
escribe_svc(...) ; /* invocación servicio*/
}
int main (...) {/* función main servidor */
...
}
mensaje_xdr.c
/* rutinas empaquetado/alineamiento datos XDR */
bool_t xdr_XXXXX (...) {
...
}
mensaje_xdr.c
/* rutinas empaquetado/alineamiento datos XDR */
bool_t xdr_XXXXX (...) {
...
}
1
2
3
4
5
6
7
8
Soporte Comunicaciones Sun RPC (LIBRERÍA S.O.)
(Protocolo Petición-Respuesta)
1 Invocación stub local
8 Retorno stub local
4 Invocación serv. remoto
5 Retorno serv. remoto
6 Envio mensaje respuesta
7 Recepción mensaje respuesta
2 Envio mensaje petición
3 Recepción mensaje petición
Código escrito programador
Código generado

El programa cliente

El cliente importa la interfaz de servicio y llama a sus procedimientos. El binding no se hace a nivel de red: los servicios se registran en sus máquinas locales utilizando el portmapper:

- Hay un portmapper en cada ordenador
- Registra el puerto de cada servicio local a su máquina

El cliente debe especificar:

- El ordenador del servidor
- El número de programa y la versión del mismo

Forma de realizar la invocación:

- Se llama al procedimiento número x del programa y versión w de la máquina z
- Previamente se consulta al portmapper del servidor el puerto del servicio

Ejemplo de cliente:

```
#include "mensaje.h"

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

   char *host;
   CLIENT *clnt;
   void  *result_1;
   int  escribe_1_arg;

   if (argc < 2) {
      printf ("usage: %s server_hostverb++n", argv[0]);
      exit (1); }
   host = argv[1];

   clnt = clnt_create (host, MENSAJE, MENSAJE_VER, "udp");
   if (clnt == NULL)  {
      clnt_pcreateerror (host);
      exit (1); }

   do {
      printf("Número: ");
      scanf("%d",&escribe_1_arg);
      result_1 = escribe_1(&escribe_1_arg, clnt);
      if (result_1 == (void *) NULL)  {
         clnt_perror (clnt, "call failed"); }

   } while (escribe_1_arg != 0);

   clnt_destroy (clnt);
   exit (0);
}
```

La llamada se realiza a través del procedimiento suplente en el cliente:

suplente = procedimiento + "_" número

El suplente recibe el handle del cliente, obtenido mediante clnt_create
En su creación se especifica el protocolo (udp o tcp)

La llamada a clnt_create consulta al portmapper

El suplente invoca a clnt_call, que hace la LPR:

- Utiliza semántica *al-menos-una-vez*
- Otras funciones fijan la temporización entre reintentos y el tiempo de espera total
- Si acaba el tiempo total y no hay respuesta ⇒ error

Prototipo de clnt_call:

```
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
```

- CLIENT *clnt es el handle del cliente
- u_long procnum es el identificador del procedimiento
- xdrproc_t inproc, outproc son los procedimientos de alineación y desalineación de argumentos
- char *in, *out son los argumentos de entrada y salida
- struct timeval tout es el tiempo total que puede durar la llamada remota

Esto se compila:

gcc mensaje_client.c mensaje_clnt.c [mensaje_xdr.c] -o cliente

El programa servidor

La implementación del servicio sigue el prototipo suministrado por rpcgen

```
#include "mensaje.h"

void * escribe_1_svc(int *argp, struct svc_req *rqstp)
{
   static char * result;
   printf("Número: %d\n",*argp);
   return (void *) &result;
}
```

- Argumentos: punteros a argumentos de entrada + una estructura con información sobre la invocación
- Resultados: puntero al argumento de salida (este caso no hay)

En el stub del servidor hay un procedimiento main:

- Establece una comunicación (socket) para recibir las solicitudes
- Exporta la interfaz de servicio, comunicando al portmapper local el número del programa, y su versión. El portmapper asigna un puerto

El repartidor es invocado cuando se lanza el servidor como un proceso ya dado por el rpc.

Funciones del repartidor:

- Aceptar las solicitudes de llamadas a procedimientos remotos
- Comprobar el número de programa y de versión
- Desalinear los argumentos
- Llamar al procedimiento correspondiente
- Alinear el resultado del procedimiento llamado
- Transmitir el mensaje de vuelta al cliente

Para compilar el servidor:

```
gcc mensaje_server.c mensaje_svc.c [mensaje_xdr.c] -o servidor
```

El enlace

- El servidor arranca y registra el número de programa y su versión en el portmapper local.
- El portmapper deberá estar ejecutándose en el servidor, en Linux es un ejecutable llamado portmap
- El cliente arranca y cliente averigua el número de puerto del servidor enviando una solicitud al portmapper.
- El enlace se establece por sockets ⇒ hay que saber el puerto

Facilidades de nivel inferior

RPC provee gran número de funcionalidades a bajo nivel:

- Para el cliente:
  - En la llamada a clnt_create() se especifica el protocolo de comunicación ("tcp" y "udp")
  - clnt_control permite especificar el tiempo total permitido para que el procedimiento remoto termine y el tiempo entre reintentos
  - Es posible utilizar un socket creado previamente para hacer la llamada
- Para el servidor:
  - Se pueden registrar procedimientos "a mano"
  - Se puede especificar el protocolo que usan o usar un socket previamente definido
- Para el cliente y el servidor:
  - Hay utilidades para crear rutinas de alineamiento y desalineamiento de los datos y especificar que son esas las rutinas a utilizar

Un ejemplo completo

Funcionalidad: aplicación que permite ver la primera línea de un fichero remoto

El fichero rmore.x define la interfaz

```
const MAX_CHAR=255;

struct linea {
   char cadena[MAX_CHAR]; };

struct lineaint {
   int error;
   char cadena[MAX_CHAR]; };

program RMORE_PROG {
   version RMORE_VER {
      struct lineaint rmore(struct linea) = 1;
   } = 1;
} = 9999;
```

- El formato de fichero para rpcgen no admite líneas #define ⇒ hay que recurrir a constantes
- Al compilar se genera un fichero .h en el que estarán los #define
- Las matrices son problemáticas uso de estructuras

Servidor:

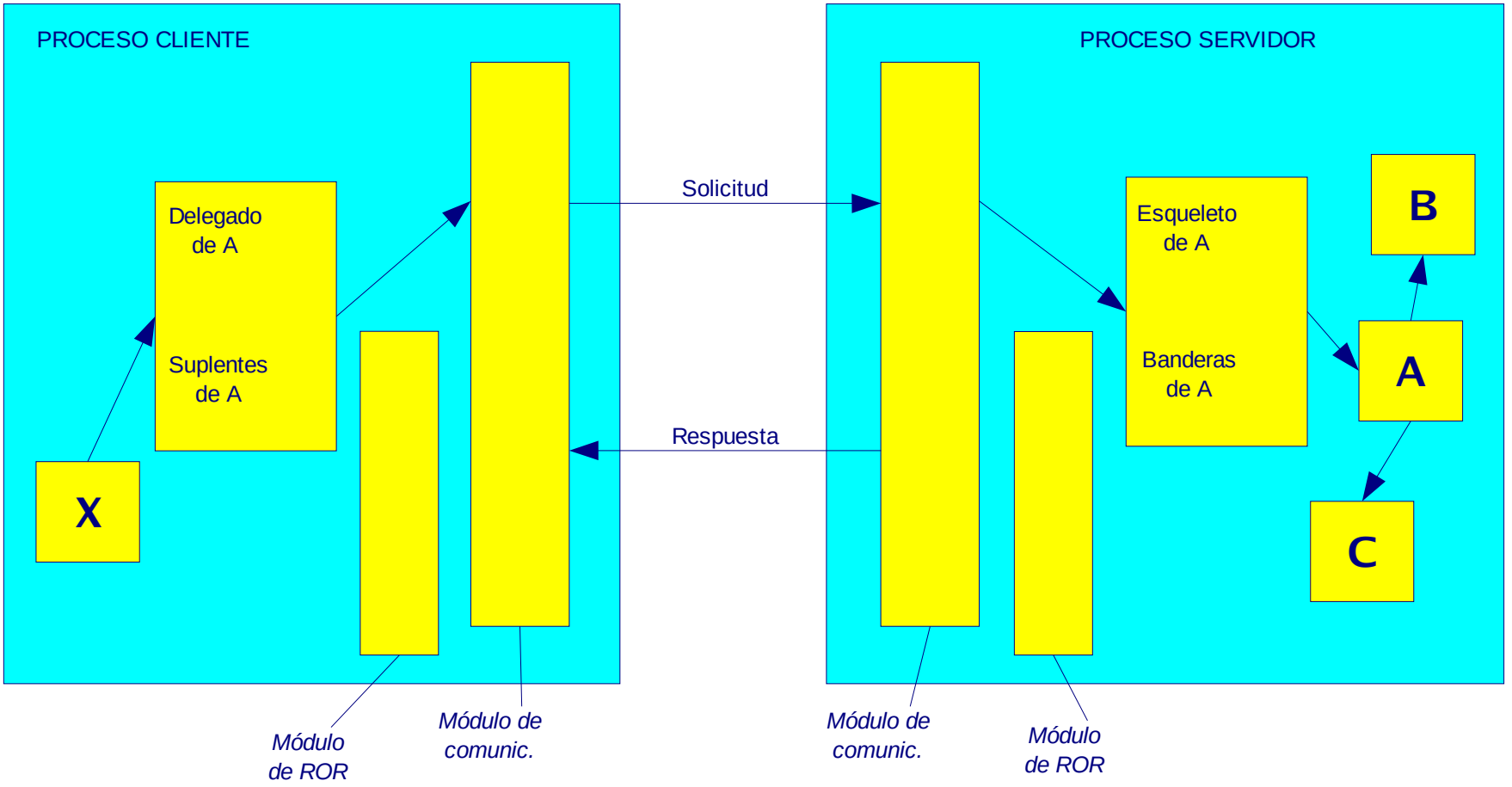
```
#include "rmore.h"

struct lineaint * rmore_1_svc(struct linea *argp, struct svc_req *rqstp)
{
   static struct lineaint result;
   FILE *f;

   printf("Voy a abrir fichero %s\n",argp->cadena);

   f = fopen(argp->cadena,"r");
   if (f == NULL) {

      printf("Error abriendo fichero %s\n",argp->cadena);
      result.error = 1;
   }
   else {
      fgets(result.cadena,MAX_CHAR,f);
      result.error = 0;
      fclose(f);
   }

   if (result.error == 0)
      printf("Resultado: %s",result.cadena);

   return &result;
}
```

Cliente:

```
#include "rmore.h"

void rmore_prog_1(char *host, char *fichero) {
   CLIENT *clnt;
   struct lineaint *result_1;
   struct linea rmore_1_arg;

   strcpy(rmore_1_arg.cadena,fichero);
   printf("Fichero remoto: %s\n",rmore_1_arg.cadena);

   clnt = clnt_create (host, RMORE_PROG, RMORE_VER, "udp");
   if (clnt == NULL) {
      clnt_pcreateerror (host);
      exit (1); }

   result_1 = rmore_1(&rmore_1_arg, clnt);
   if (result_1 == (struct lineaint *) NULL)
      clnt_perror (clnt, "call failed");
   clnt_destroy (clnt);

   if (result_1->error)  printf("Hay algún problema en el servidor\n");
   else  printf("Resultado: %s",result_1->cadena); }


int main (int argc, char *argv[]) {
   char *host;
   if (argc != 3) {
      printf ("uso: %s servidor fichero\n", argv[0]);
      exit (1); }
   host = argv[1];
   rmore_prog_1 (host, argv[2]); }
```

4.3. Invocación de Métodos Remotos

4.3.1. Características generales



Esquema general de Invocación de Métodos Remotos (RMI, *Remote Method Invocation*)

Módulo de comunicaciones: protocolo de comunicación y semántica de invocación ⇒ invocación remota igual que completamente local

Para cada objeto remoto invocable:

- Se crea un delegado o proxy (para el cliente es como un objeto local)
- Invocación ⇒ mensaje pasado al objeto remoto
- El objeto remoto ejecuta y responde sin saber que la respuesta va a ir a un cliente remoto ⇒ la distancia cliente/servidor es transparente
- En el delegado se implementan los métodos como métodos suplentes

Una clase que representa a objetos accesibles de forma remota tiene:

- Un esqueleto (skeleton): implementa los métodos del objeto como métodos bandera
- Un repartidor (dispatcher) que invoca el método bandera adecuado basándose en la identificación de éste método

Forma de hacer las cosas:

Interfaz → delegado, repartidor y esqueleto

En Java RMI, la interfaz se define como una interfaz Java estándar

Pasos de la invocación remota

- En el cliente: invocación de un método de un objeto remoto (OR) ⇒ se invoca un método suplente en el delegado del OR
- El suplente construye y envía una solicitud con identificación del OR, la identificación del método y argumentos
- El módulo de comunic. del servidor (del objeto remoto) pasa la solicitud al repartidor del OR
- El repartidor del OR invoca el método bandera en el esqueleto
- El método bandera desalinea argumentos e invoca
- El método bandera alinea el resultado y envía respuesta al delegado
- El método suplente desalinea respuesta y la pasa al emisor

El programa del servidor incluirá la clase de los repartidores y los esqueletos y las secciones de inicialización

El programa del cliente incluirá la clase de los delegados. Para crear los métodos delegado, no se puede utilizar un constructor convencional, sino que se usan los denominados métodos factoría

Módulo de ROR

- El módulo de referencias de objetos remotas (ROR) gestiona los delegados y las RORs
- La mayoría de las ROR se obtienen por RMI: se busca por su nombre un objeto remoto se crea un delegado que asocia nombre textual a la referencia del objeto remoto
- Los delegados se crean "bajo demanda":
  - Se invoca un método de un OR que no tiene delegado
  - Si se incluye una referencia a un OR en una respuesta ⇒ el método bandera pide al módulo de ROR la referencia y la incluye en la respuesta
- Módulo de ROR = tabla de correspondencias identificadores locales ↔ identificadores remotos
  - Para objetos locales accesibles por remotos: correspondencia referencia de acceso local ↔ ROR con la que se accede remotamente a dicho método
  - Para objetos remotos accesibles por locales: correspondencia ROR del objeto remoto ← delegado local

  De esto, en el caso de Java se encarga el *RMIregistry*, mientras que en caso de CORBA se encarga el *Naming Service*

Recolección de basura

- En IL el recolector de basura (*garbage collector*) elimina de la memoria los objetos que ya no se usan
- En IR:
  - Mantener un objeto mientras exista una ROR al mismo
  - Cuando nadie mantenga la ROR, el objeto será liberado (siempre y cuando no haya referencias locales)
- Forma de realizar esto:
  - Cuando llega una ROR a un cliente, se informará de esta ROR al servidor donde reside el OR
  - Cuando la ROR se deja de usar, deberá informarse al servidor
  - En un servidor, el módulo de ROR registra cuántos clientes tienen RORs de los objetos del servidor
  - En un cliente, el módulo de ROR informa a los servidores de las incorporaciones o eliminaciones de las RORs de los objetos de dichos servidores

En Java RMI el módulo de ROR de la JVM colabora con la recolección local para proporcionar recolección distribuida

Objetos persistentes

Objeto persistente: debe sobrevivir entre activaciones de los procesos

Se construyen almacenes de objetos persistentes que:

- Cuando dejan de ser necesarios, se desactivan (se guardan en disco de forma aplanada)
- Cuando se vuelven a invocar se activan (el llamante no debe percibir si ya estaba en memoria o si se ha activado)

4.3.2. Java RMI

Java RMI extiende el modelo de objeto de Java a objetos distribuidos. Permite IMR de forma casi transparente:

- Con la misma sintaxis
- Se debe manejar la excepción RemoteException
- El OR debe implementar la interfaz Remote
- No es necesario un IDL, pues todo se hace definiendo una interfaz de Java estándar
- Si la IMR implica comunicación de objetos entre cliente y servidor, éstos deberán extender la interfaz Serializable ⇒ las instancias serán alineables
- Instancias alineables ⇒ almacenables en disco

Definición de la interfaz

Interfaces en Java RMI como una interfaz normal:

- Extendiendo la interfaz java.rmi.Remote
- Lanzando la excepción RemoteException
- Argumentos y resultados: tipos primitivos, objetos normales y objetos remotos (denotados por el nombre de su interfaz remota)

Comunicación entre cliente y servidor

Todos los argumentos de entrada salvo el resultado. Entre los argumentos: datos de tipos primitivos (alineables, por valor), objetos locales (alineables, por valor), objetos remotos (como RORs)

Al alinear, cualquier objeto que implemente Remote es sustituido por su ROR y el receptor la descarga por su URL ⇒ la máquina en la cual resida deberá correr algún de servidor HTTP

Cuando se pasa una clase de una JVM a otra, se descarga el código:

- Si se pasa por valor y el receptor no posee la clase
- Si es remoto (referencia) y el receptor no posee la clase del delegado

RMIRegistry

- El RMIRegistry es el *binder* de la JVM: mantiene una tabla de correspondencias entre la ROR y el objeto local
- Deberá haber un registro en cualquier máquina que aloje objetos que sean accesibles de forma remota
- Esto no da servicios a través de red, solamente actúa de binder
- El RMIRegistry es accedido a través de métodos de la clase Naming, con la siguiente sintaxis: rmi://maquina:puerto/objeto donde maquina y puerto especifican la ubicación del RMIRegistry

Un ejemplo completo

La construcción de la aplicación involucra los siguientes pasos:

1. Escribir y compilar el código Java para las interfaces

2. Escribir y compilar el código Java para la implementación de las clases accesibles remotamente

3. Generar las clases delegado y esqueleto (con los métodos suplente ybandera, respectivamente) a partir de la implementación

4. Escribir el código Java para el programa servidor

5. Escribir el código Java para el programa cliente

6. Instalar y ejecutar el sistema

Ejemplo: un servidor que imprime por pantalla los mensajes que le van llegando desde el cliente

1. Escribir y compilar el código Java para las interfaces

Definición de la interfaz:

```
/* Mensajes.java */
public interface Mensajes
   extends java.rmi.Remote {
      public void escribe(String mensaje)
      throws java.rmi.RemoteException;
   }
```

Observaciones:

- Se extiende la interfaz Remote
- Los métodos accesibles remotamente deberán lanzar la excepción RemoteException

El interfaz se compila mediante

```
$javac Mensajes.java
```

y deberá estar disponible tanto en el servidor como en el cliente

2. Código Java para la implem. de las clases accesibles remotamente

Esta es la implementación que habrá en el servidor:

```
/* MensajesImpl.java */
public class MensajesImpl
extends java.rmi.server.UnicastRemoteObject
implements Mensajes
{
   public MensajesImpl()
   throws java.rmi.RemoteException {
      super();
   }
   public void escribe(String s)
   throws java.rmi.RemoteException {
      System.out.println(s);
   }
}
```

- Se usa UnicastRemoteObject para enlace con el sistema RMI
- Debe haber un constructor que invoque al constructor del padre para hacer el enlace RMI y la inicialización del objeto remotos (alternativa: no extender esta clase y usar exportObject())

Se compila mediante

```
$javac MensajesImpl.java
```

3. Generar las clases delegado y esqueleto (con los métodos suplente y bandera, respectivamente) a partir de la implementación

Esto se consigue utilizando el compilador RMI rmic sobre el fichero Java que incluye la implementación de las clases MensajesImpl.java:

```
$rmic MensajesImpl
```

Esto generará las clases:

- MensajesImpl_Skel (esqueleto del servidor)
- MensajesImpl_Stub (delegado del cliente y métodos suplente y bandera)

4. Escribir el código Java para el programa servidor

La clase MensajesServer proporciona la funcionalidad esencial:

```
/* MensajesServer.java */
import java.rmi.Naming;
public class MensajesServer {
   public MensajesServer() {
      try {
         Mensajes c = new MensajesImpl();
         Naming.rebind(
            "rmi://localhost:1099/MensajesService", c);
      } catch (Exception e) {
         System.out.println("Problema: " + e);
      }
   }
   public static void main(String args[]) {
      new MensajesServer();
   }
}
```

Se compila mediante

```
$javac MensajesServer.java
```

5. Escribir el código Java para el programa cliente

```
/* MensajesClient.java */
import java.io.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class MensajesClient {
   public static void main(String[] args) {
      String mensaje ="";
      try {
         Mensajes c = (Mensajes) Naming.lookup("rmi://localhost/MensajesService");
         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
         do {
            mensaje = in.readLine();
            c.escribe(mensaje);
         } while (!mensaje.startsWith("fin"));
      }
      catch (Exception e) {
         System.out.println("Problema: "+e);
      }
   }
}
```

Se compila mediante

```
$javac MensajesClient.java
```

6. Instalar y ejecutar el sistema

En primer lugar, se debe iniciar el registro RMI en el servidor:

```
$rmiregistry &
```

Y se ejecutan las clases del servidor y del cliente. En el servidor:

```
$java MensajesServer
```

y en el cliente:

```
$java MensajesCliente
```

NOTA: la clase MensajesImpl_Stub fue generada a partir de la implementación del servicio (en el servidor) e incluye los métodos suplentes, necesarios en el cliente ⇒ dos posibilidades:

- Llevar el fichero MensajesImpl_Stub.class al cliente
- Poner esta clase accesible en el servidor para los clientes ⇒ instalar algún tipo de servicio HTTP en el servidor