Tema 2: Llamadas a procedimientos remotos

Sistemas Distribuidos

2022-2023

Middleware: RPC

Concepto

Transparencia: Ocultar el paso de mensajes subyacente

> **IDEA**
>
> Poder llamar desde una máquina a una función que está en otra

Problemas:

- Codificación de la información
- Espacio de direcciones
- Son dos procesos independientes ⇒ Sincronización
- Errores en la red, servidor apagado...

Mecanismo de funcionamiento

Considera una llamada convencional (local al propio proceso):

```
int sumar(int a, int b) {
    int r = a + b;
    return(r);
}

int main() {
    int x=5, y=20, z;
    z = sumar(x,y);
    printf("%d", z);
}
```

Llamada convencional:

- El programa principal guarda los parámetros en una zona común (pila)
- Se transfiere el control a la rutina
- La rutina recoge los parámetros de la pila
- Ejecuta su código
- `return(r)` copia `r` a un lugar común (registro de la CPU)
- El control vuelve al programa principal
- Se extrae el resultado (del registro) y se asigna a `z`

Invocación remota

- El proceso *llamador* envía los parámetros en un mensaje por la red
- El proceso *llamador* se bloquea esperando el mensaje de respuesta
- El proceso *llamado* está esperando conexiones
- Recibe una conexión y lee el mensaje con los parámetros
- Ejecuta el código de la función solicitada, pasándole los parámetros
- Envía el resultado `r` en un mensaje de respuesta
- El proceso *llamador* recibe el mensaje de respuesta y se desbloquea
- Asigna el dato recibido a la variable `z`

Como vemos hay muchos paralelismos entre una llamada local y una remota, pero hay también diferencias:

- En la RPC hay dos procesos
- El problema de la localización de la máquina donde está el procedimiento remoto
- La comprobación de tipos en la llamada
- El paso por referencia (¡imposible!) (¿o no?)

Idea clave: los *extremos* (*stubs*)

Para posibilitar la máxima transparencia en la RPC, existen dos funciones encargadas de la interacción a nivel de mensajes:

- Extremo del cliente: Está en el mismo proceso que el *llamador*
- Extremo del servidor: Está en el mismo proceso que la función *llamada*

Supón el siguiente programa, pero ahora la función `sumar()` está en otro proceso (y quizás en otra máquina):

```
int main() {
    int x=5, y=20, z;
    z = sumar(x,y);
    printf("%d", z);
}
```

El extremo del cliente

- Tiene el mismo prototipo que la función remota
- Es quien codifica y envía los parámetros, junto con el nombre de la función remota a invocar
- Y queda esperando por la respuesta y la retorna al llamador

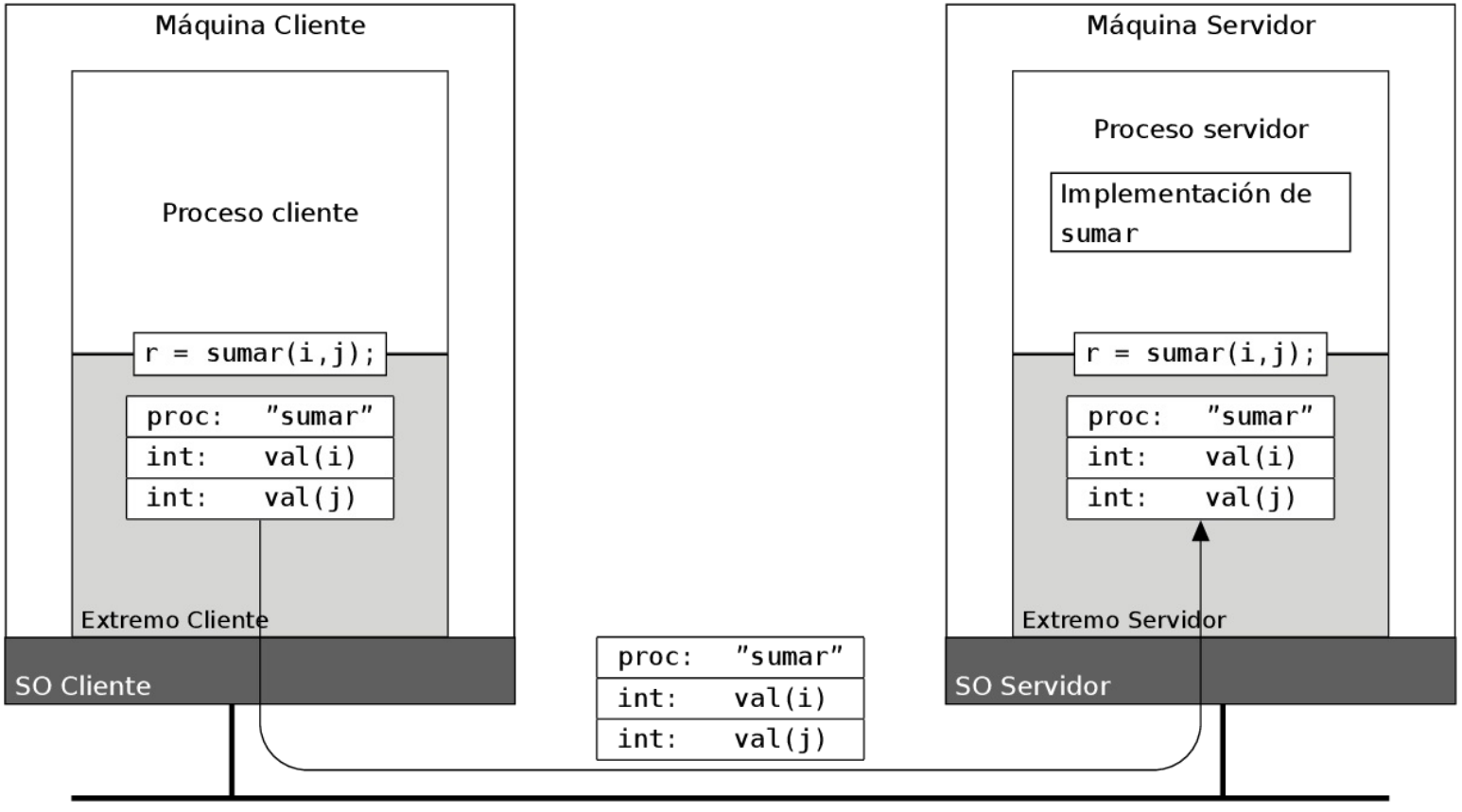
```
// Pseudocodigo
int sumar(int a, int b) {
    int r;
    localizar servidor
    codificar valores de a, b
    enviar mensaje al servidor <sumar, a, b>
    leer respuesta (bloqueante) sobre r
    return(r);
}
```

El extremo del servidor

- Está en un bucle infinito, esperando conexiones y mensajes
- Invoca a la verdadera función remota (que es local ahora)
- Envía al cliente el valor retornado por la función

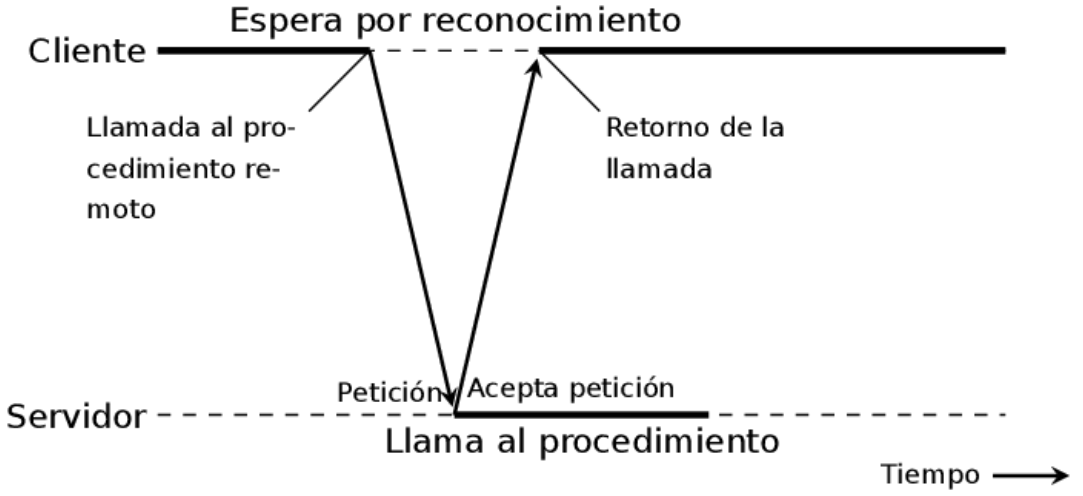
```
// Pseudocódigo
void extremo_del_servidor(){
    int resultado;
    bucle infinito:
        esperar por cliente (accept)
        leer mensaje del cliente <sumar, a, b>
        decodificar los valores de a y b
        resultado = sumar(a,b); // La función que
                                // realmente
                                // hace el trabajo
        enviar resultado al cliente
        cerrar conexión con ese cliente
}
```

Diagrama de la invocación remota


Máquina Cliente
Proceso cliente
r = sumar(i,j);
proc: "sumar"
int: val(i)
int: val(j)
Extremo Cliente
SO Cliente
Máquina Servidor
Proceso servidor
Implementación de sumar
r = sumar(i,j);
proc: "sumar"
int: val(i)
int: val(j)
Extremo Servidor
SO Servidor
proc: "sumar"
int: val(i)
int: val(j)

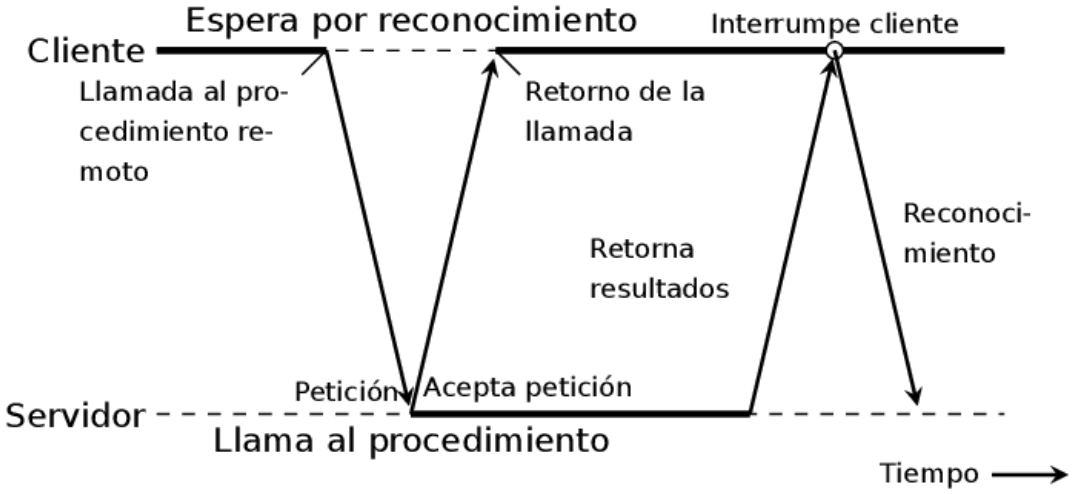
RPCs asíncronas

El cliente no espera por la respuesta (porque no la hay).



RPCs asíncronas (I)

- El cliente no espera por la respuesta (pero ésta llega más tarde)
- El servidor llama un *callback* del cliente (por RPC)



Implementaciones del concepto RPC

- ONC RPC
- DCE RPC
- Microsoft RPC (DCOM)
- CORBA
- Java RMI
- Web Services (SOAP)

ONC RPC

Problemas y soluciones en ONC RPC

Problema	Solucion
Codificacion de los parametros	XDR
Especificacion del interfaz (prototipo del procedimiento remoto)	IDL
Localizacion del servicio	*portmapper* (registro)
Generacion automatica del codigo de los extremos	rpcgen

Especificación del interfaz

Se especifica en un lenguaje específico:

- En un fichero de extensión `.x`
- Usa XDR para declarar los tipos del parámetro y resultado
- Y al final declara los servicios ofertados por RPC (**un solo parámetro**)
- Usa un esquema numérico para identificar:
  - El servidor que los ofrece (números "especiales")
  - El número de versión ofertado (números libremente elegidos)
  - Cada función específica (números libremente elegidos)

Ejemplo de especificación de interfaz

```
/* Fichero: calc.x */
/* Zona de declaración de tipos XDR */
struct Operandos {
    int a;
    int b;
};
typedef int Resultado;

/* Zona de declaración del interfaz */
program SERVIDOR_CALCULO {
    version BETA {
      Resultado sumar(Operandos ab) = 1;
    } = 17;
} = 0x40008001;
```

Explicación

- Funciones: sintaxis C, más asignación final de un número (será su "identificador" para el protocolo entre extremos)
  - `suma` es la función `1`
- Funciones agrupadas en *versiones* (tienen nombre y un número)
  - `BETA` es la versión `17`
- Versiones agrupadas en *programas*, (tienen nombre y un número)
  - `SERVIDOR_CALCULO` es el programa número `0x400080001` (hex)

Los números los elige libremente el programador, salvo el de programa

Existen números reservados. Pueden usarse los comprendidos entre `0x4000000` y `0x5FFFFFF`

Ejecución de rpcgen

Se generan dos ficheros relacionados con XDR

- `calc.h` para incluir en los programas a desarrollar
- `calc_xdr.c` para los filtros de los tipos `xdr_Operandos()` `xdr_Resultado()`

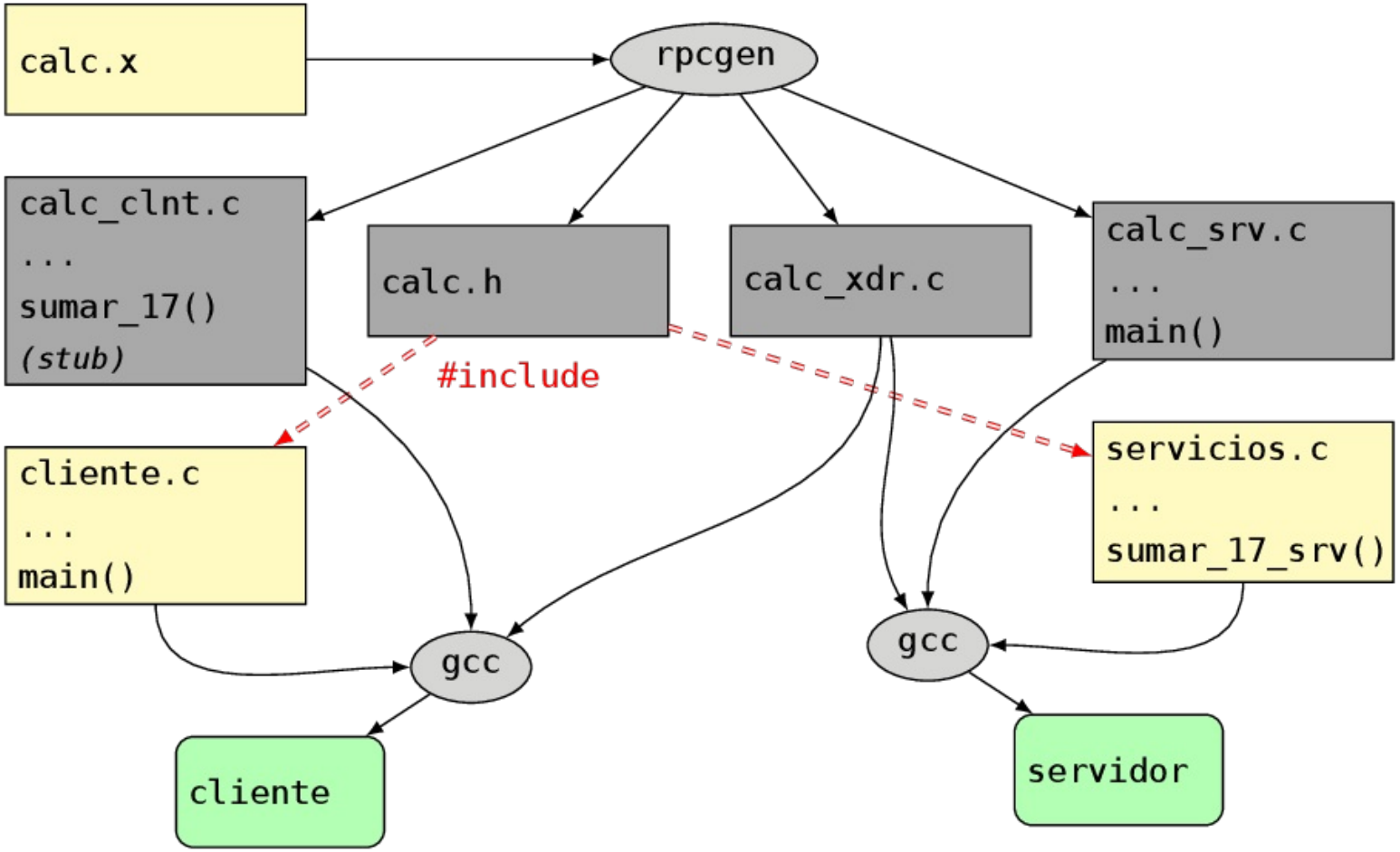
Ejecución de rpcgen

Y otros dos relacionados con RPC

- `calc_clnt.c` implementa el extremo del cliente.
  - Contiene la función `sumar_17()`
  - Que se ocupa de contactar con el servidor, transmitirle parámetros, esperar respuesta, etc
- `calc_svc.c` implementa el extremo del servidor
  - Contiene la función `main()`
  - Que inicializa todo y espera por clientes, recibe parámetros etc.
  - E invoca a la función `sumar_17_svc()`

> El programador debe escribir el código de `sumar_17_svc()` y del del posible cliente

Ejecución de rpcgen



Implementación de los servicios

- Nombre del servicio: incluye el número de version (`sumar_13()`)
- Parámetros:
  - Desde el cliente: sólo uno y por referencia
  - Otro parámetro adicional para aspectos de seguridad (no lo usaremos)
- Resultados: Por referencia también (variable local `static`)

```
// Fichero: servicios.c
#include "calc.h"
Resultado * sumar_17_svc(Operandos *datos, struct svc_req *r)
{
    static Resultado result;
    result = datos->a + datos->b;
    return (&result);       // Hay que retornar por referencia
}
```

Compilación del servidor

```
$ gcc -Wall -c servicios.c
$ gcc -c calc_svc.c
$ gcc -c calc_xdr.c
$ gcc -o servidor servicios.o calc_svc.o calc_xdr.o
$ ./servidor &
```

Implementación del cliente

El cliente es más complejo:

- Debe declarar una estructura `CLIENT`
- Inicializarla con:
  - Nombre o IP de la máquina servidora
  - Número del servidor (`0x40008001`)
  - Número de versión (`17`)
  - Protocolo (`"tcp"` o `"udp"`)
- Luego ya puede invocar a `sumar_17()` (su extremo local)

Para manejar esto hay funciones específicas:

```
clnt_create()    clnt_pcreateerror()
clnt_perror()    clnt_destroy()
```

Ejemplo de código de cliente

```c
#include "calc.h" // Faltarian mas includes...
int main() {
    CLIENT *clnt;
    Resultado *res;
    Operandos op;

    clnt = clnt_create("mi.servidor.com",
                       SERVIDOR_CALCULO, BETA, "udp");
    if (clnt==NULL) {
        clnt_pcreateerror("No puedo inicializar cliente");
        exit(1);
    }
    // Preparar parametros
    op.a = 5;    op.b = 7;
    // Realizar invocacion remota
    res = sumar_17(&op, clnt);
    // Mostrar resultado
    if (res == NULL) {
        clnt_perror(clnt, "Fallo en la invocacion remota");
        exit(1);
    }
    printf("Resultado: %d\n", *res);
    clnt_destroy(clnt);
    return 0;
}
```

Compilación del cliente

```
$ gcc -Wall -c cliente.c
$ gcc -c calc_clnt.c
$ gcc -c calc_xdr.c
$ gcc -o cliente cliente.o calc_svc.o calc_xdr.o
$ ./cliente &
```

Makefile

```
all: cliente servidor

cliente: cliente.o calc_xdr.o calc_clnt.o
    gcc -o $@ $^

servidor: servicios.o calc_xdr.o calc_svc.o
    gcc -o $@ $^

cliente.o: cliente.c calc.h

servidor.o: servidor.c calc.h

calc.h calc_xdr.c calc_svc.c calc_clnt.c: calc.x
    rpcgen $<
```

Más de ONC RPC (no cubierto en esta asignatura)

- El extremo de servidor creado por `rpcgen` se puede editar
  - Para incluir `fork()` o `pthread()` y hacerlo concurrente
- La función `clnt_control()` permite en el cliente:
  - Fijar un tiempo máximo de espera por la respuesta
  - Obtener IP y puerto del servidor
  - Reintentar llamadas fallidas
- Otras funciones permiten
  - Actuar directamente sobre el localizador
  - Llamadas "por lotes" (varias juntas, sin esperar respuesta)
  - LLamadas multidifusión (a varios servidores a la vez)
- Seguridad
  - El servidor puede verificar credenciales de un cliente
  - Y negarse a ejecutar el servicio

Java RMI

Los conceptos son prácticamente los mismos, pero orientados a objetos.

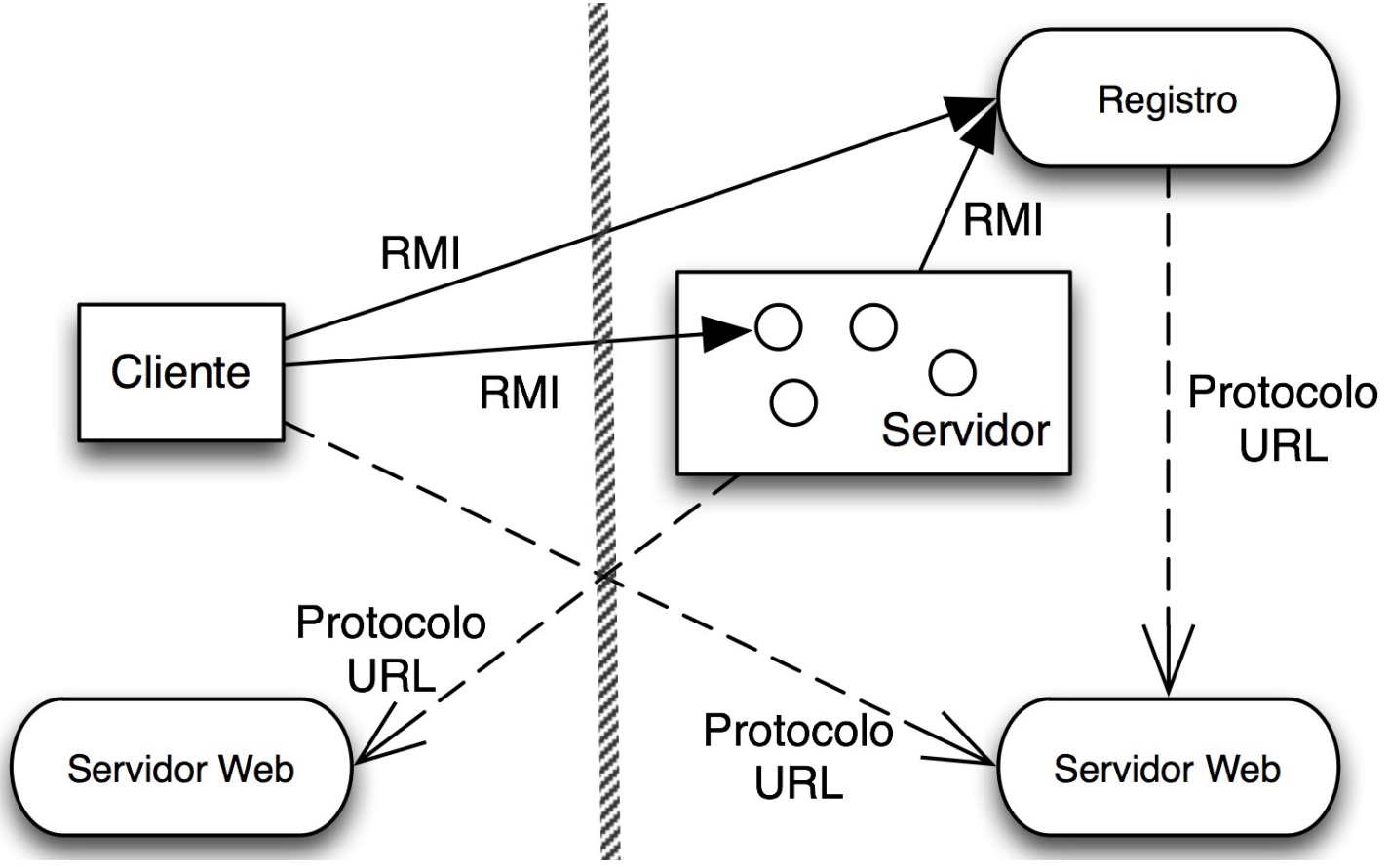
Puntos en común:

- Una clase remota implementa métodos que un cliente quiere invocar
- El cliente invoca en realidad un *stub* local
- Otra clase remota implementa el *esqueleto* con el que conecta el cliente
- Es necesario un "registro" que ponga en contacto el cliente con el servidor
- El cliente debe conocer la IP y puerto del registro

Java RMI. Características propias

- Los parámetros pueden ser cualquier tipo básico u objeto *serializable*
  - Se pueden transmitir clases enteras como parámetro
  - Incluso su código
- La especificación del interfaz no requiere lenguaje aparte
  - Usa `interfaz` de Java
- No requiere un "rpcgen" aparte:
  - Lo tuvo: `rmic` (no es necesario desde Java 1.5)
  - El esqueleto del servidor no se genera en código fuente, sino sino "al vuelo"
  - El *stub* del cliente no se genera en código fuente, sino "al vuelo"
- El cliente no necesita tener el *stub* localmente, lo obtiene del servidor
- El registro contiene también *urls* de las que el cliente puede descargar *stubs*

Arquitectura RMI



Implementación (lado servidor)

1. Escribir el interfaz (que *extienda* `Remote`)
   - Todos los métodos deben declarar `throws RemoteException`
2. Escribir la clase que *implementa* el interfaz
   - Debe *extender* `UnicastRemoteObject`
   - Contiene el código de los servicios
   - Y un constructor sin parámetros con `throws RemoteException`
3. Escribir un servidor que:
   - Instancie la clase implementación
   - Registre la instancia en el registro

Implementación (lado cliente)

1. Se necesita la clase interfaz (basta el `.class`)
2. El cliente obtiene una referencia a la clase remota
   - Para ello conecta con el `rmiregistry` remoto
   - En realidad obtiene un *stub*, pero "no lo sabe"
3. Invoca los métodos en ese objeto-referencia

Ejemplo

Interfaz:

```
import java.rmi.Remote;

public interface Calculadora extends Remote {
    int suma  (int o1, int o2) throws RemoteException;
    int resta (int o1, int o2) throws RemoteException;
}
```

Ejemplo

Implementación de los servicios:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculadoraImpl   extends UnicastRemoteObject
                               implements Calculadora {
    public CalculadoraImpl() throws RemoteException { // Constructor
        super();
    }

    public int suma (int op1, int op2) {    // Servicio
        return op1 + op2;
    }

    public int resta (int op1, int op2) {   // Servicio
        return op1 - op2;
    }
}
```

Ejemplo

Servidor (versión mínima)

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Servidor {
    public static void main(String[] args) throws Exception {
          CalculadoraImpl obj=new CalculadoraImpl();
          Naming.rebind("CalculadoraServer",obj);
     }
}
```

Ejemplo

Cliente (versión mínima)

```
import java.rmi.*;

public class Cliente {
    public static void main(String[] a) throws Exception {
        Calculadora obj =  (Calculadora)Naming.lookup(
                  "rmi://mi.servidor.com/CalculadoraServer");
        int op1=6, op2=8;
        int result;

        result = obj.sumar(x,y);
        System.out.println(
            String.format("%d + %d = %d", op1, op2, result));
    }
}
```

Despliegue

Máquina del servidor

1. Ejecutar `rmiregistry`

```
$ rmiregistry
```

2. Ejecutar el servidor

```
$ java -Djava.rmi.server.hostname=mi.servidor.com \
       -Djava.rmi.server.codebase=file:/carpeta/clases Servidor &
```

Máquina del cliente

1. Ejecutar cliente

```
$ java Cliente
```

Muchas más cosas en RMI

- ¿Seguridad?
  - Políticas de seguridad de Java (conexión, descarga, etc.)
  - Canal RMI no cifrado por defecto
  - Se puede usar sobre SSL, o en un *túnel*
- ¿*codebase*?
  - Es un almacén de código (`.class`)
  - Otra clase puede conectarse a él y descargarlo (por RMI)
  - Así es como llega el *stub* al cliente, por ejemplo

Muchas más cosas en RMI

- ¿Concurrencia?
  - SÍ: se crea un hilo para cada cliente
- ¿Carga del servicio bajo demanda?
  - Usando `UnicastRemoteObject` no (el objeto implementación debe existir en un servidor en ejecución)
  - Pero hay otros modelos (`Activatable`) que pueden cargar la clase del disco bajo demanda
- ¿El cliente debe conocer la IP del registro?
  - Existen tecnologías (Jini) para descubrirlos por *multicast*