Tema 6
Llamadas a procedimientos remotos

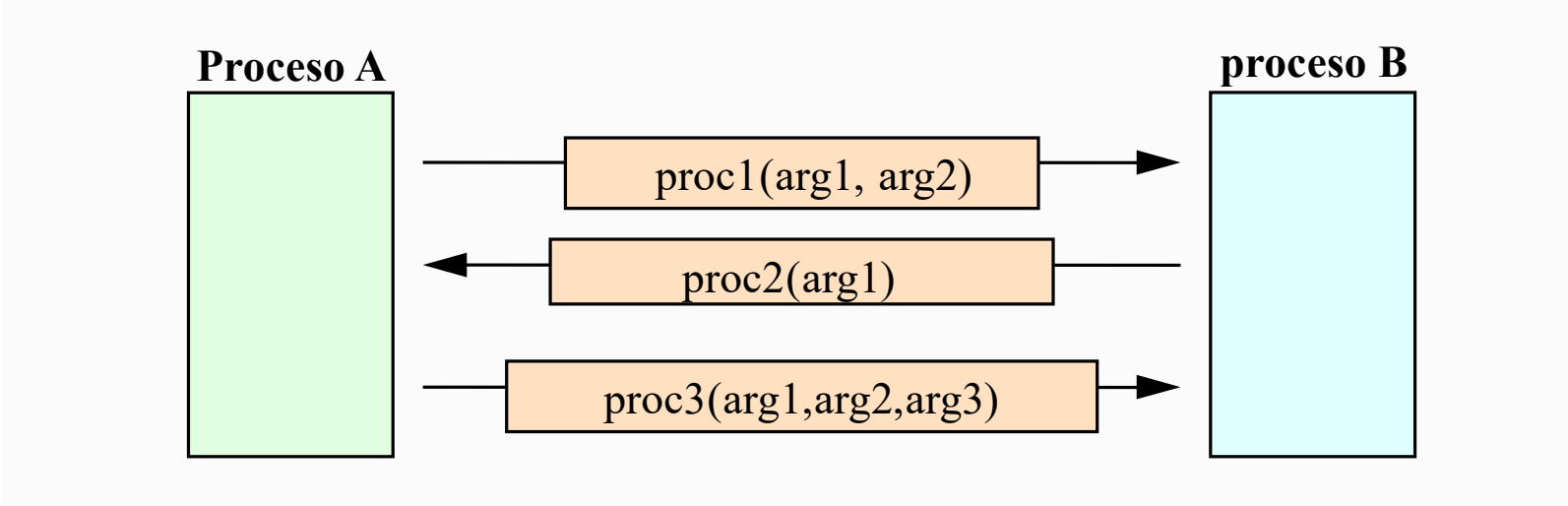
Sistemas Distribuidos
Grado en Ingeniería Informática
Universidad Carlos III de Madrid

Contenidos

- *Remote Procedure Call* (RPC)
  - Conceptos básicos
  - Aspectos relacionados con las RPCs
- RPC de Sun/ONC
  - Biblioteca de funciones de RPC
  - Desarrollo de programas
  - Ejemplos

Llamadas a procedimientos remotos

- **Objetivo:** hacer que el software distribuido se programe igual que una aplicación no distribuida
- Mediante el modelo **RPC** la comunicación se realiza conceptualmente igual que la invocación de un procedimiento local



Aplicación monolítica con llamadas locales

Aplicación

```
int sumar (int a, int b) {
   return (a + b);
}
int restar (int a, int b){
   return (a-b);
}
 . . . .
int main(…) {
    . . .
    r = sumar (a, b);
    . . . .
}
```


Ejecutable

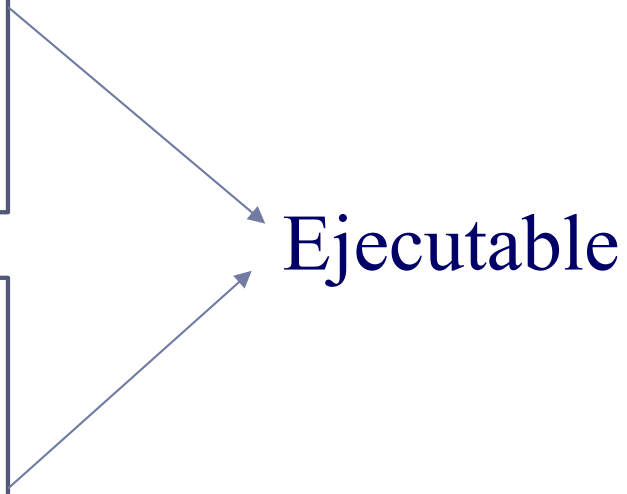
Provisión de servicios en un sistema no distribuido (llamada local)

Aplicación

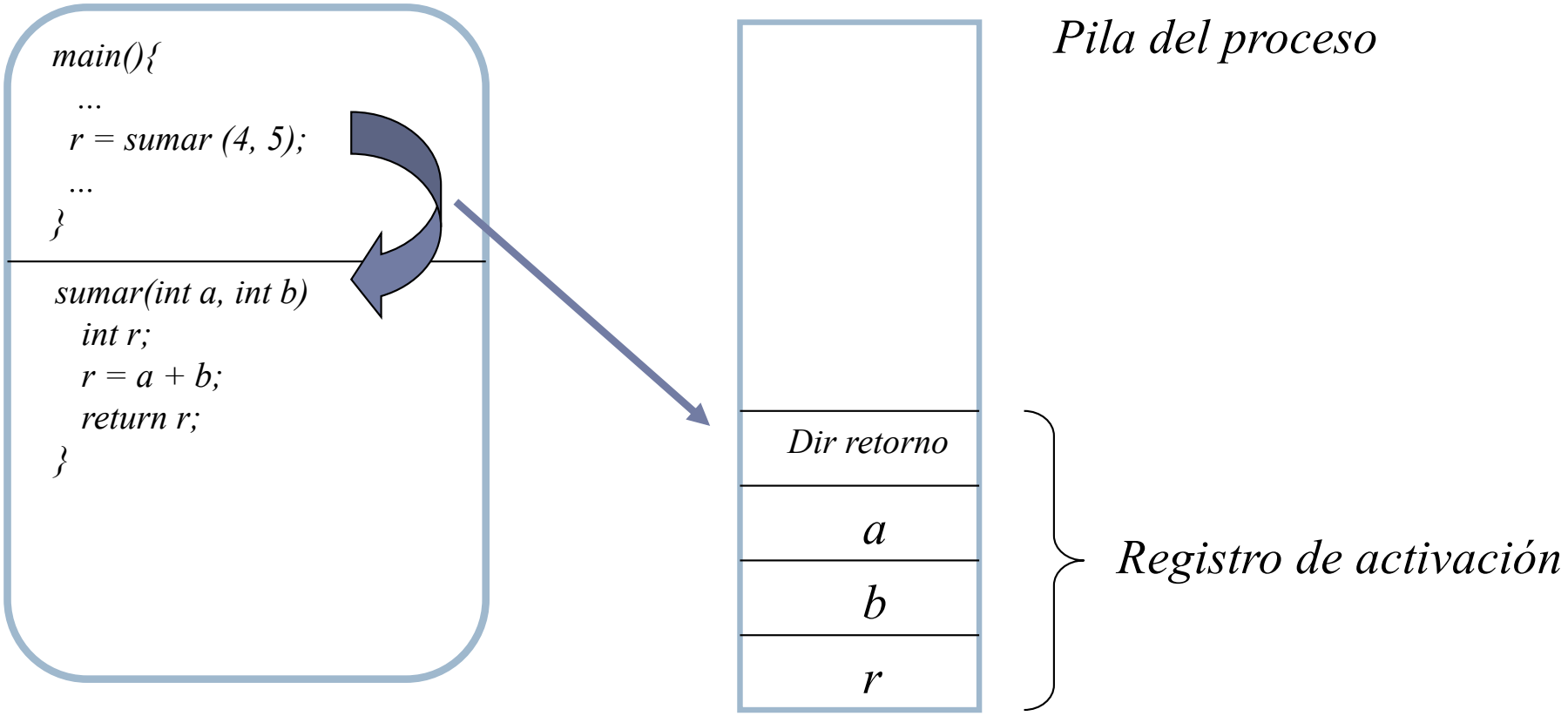
```
int main(…) {
    . . .
    r = sumar (a, b);
    . . . .
}
```

Biblioteca (lib_suma)

```
int sumar (int a, int b) {
   return (a + b);
}
int restar (int a, int b){
   return (a-b);
}
 . . . .
```



Invocación de la llamada local



Provisión de servicios en un sistema distribuido

Aplicación (Cliente)

```
int main(…) {
    . . .
    r = sumar (a, b);
    . . . .
}
```

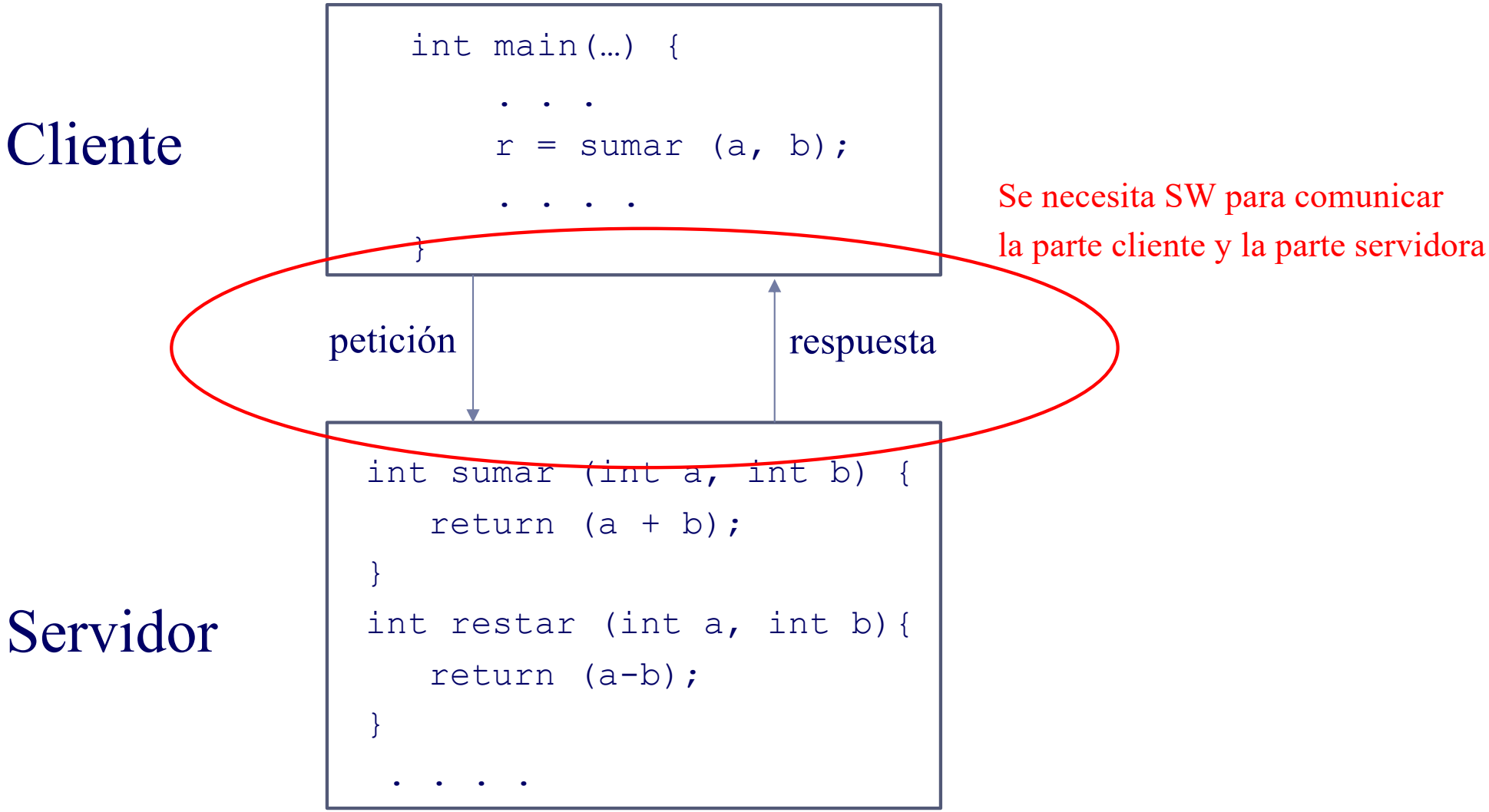
petición

respuesta

Servidor (implementación)

```
int sumar (int a, int b) {
   return (a + b);}
}
int restar (int a, int b){
   return (a-b);
}
 . . . .
```

Provisión de servicios en un sistema distribuido



Provisión de servicios en un sistema distribuido con colas de mensajes

App.

```
int main(…) {
   . . .
   err = sumar a, b, &res);
   . . . .
}
```

Impl.

```
int sumar (int a, int b) {
   return (a + b);
}
int restar (int a, int b){
   return (a-b);
}
 . . . .
```

lib_suma

```
int sumar(int a, int b, int *res) {
    cs = mq_open("Cola Servidor"…)
    cq = mq_open("Cola Cliente" CREAT);
    peticion.cod_op = SUMAR;
    peticion.colaCli= "Cola Cliente");
    peticion.a = a;
    peticion.b = b;

    mq_send(cs, &peticion);
    mq_recive(cc, &respuesta);
    res = respuesta.resultado;
    error = respuesta.error;

    *res = res;
    mq_close("Cola Servidor");
    mq_close("Cola cliente");
    mq_unlink("Cola Cliente");
    return(error);
}
```

Provisión de servicios en un sistema distribuido con colas de mensajes

servidor

App.

```
int main(…) {
   . . .
   err = sumar a, b, &res);
   . . . .
}
```

```
int main() {
  cs=mq_open("Cola Servidor", CREAR);
  while (true){
    mq_receive(&peticion);
    if (petcion.cod_op == SUMAR) {
      res = sumar(peticion.a,
                  peticion.b);
      cc= mq_open("peticion.colaCli");
      respuesta.res = res;
      respuesta.cod_err = err;
      mq_send(&respuesta);
      mq_close(cc);
    }
    if (peticion.cod_op == RESTAR)
      .
      .
      .
}
```

Impl.

```
int sumar (int a, int b) {
   return (a + b);
}
int restar (int a, int b){
   return (a-b);
}
 . . . .
```

Idea de las RPC

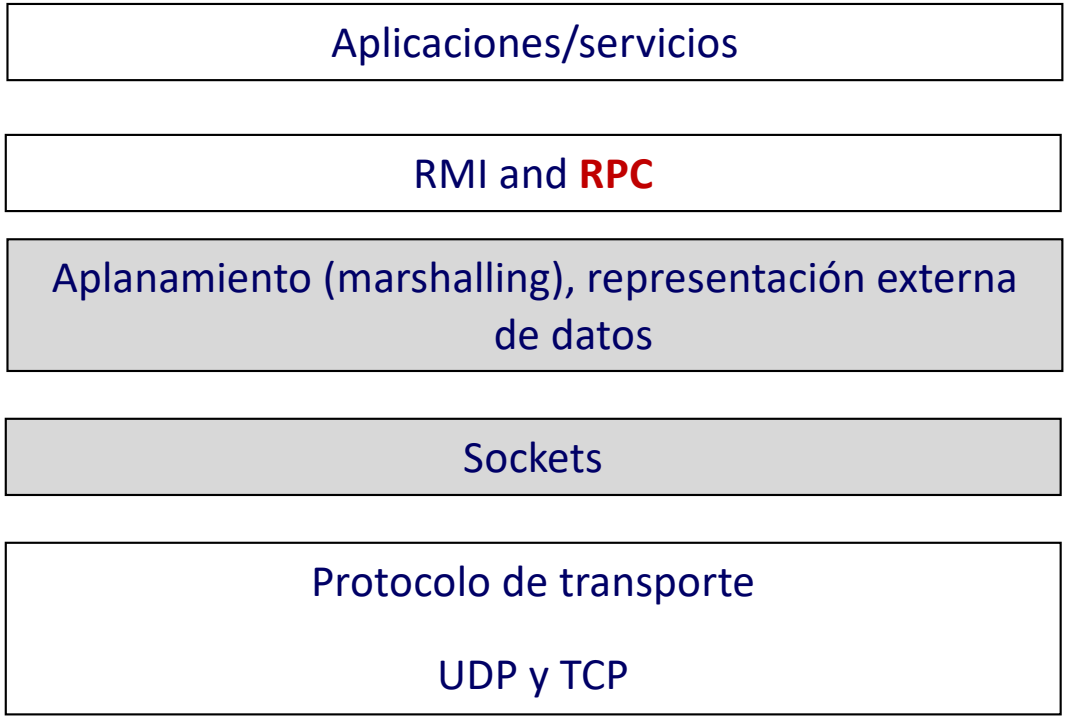
- Que el código del servidor y de lib_cliente se obtenga de forma automática a partir de la definición de la interfaz de las funciones sumar y restar (para el ejemplo anterior)

RPC

- **RPC** (*Remote Procedure Call*): llamadas a procedimientos remotos
- Por Birrel y Nelson (**1985**) en "Implementing Procedure Calls"
- Híbrido:
  - Llamadas a procedimientos
  - Paso de mensajes
- Las RPC constituyen el núcleo de muchos SSDD
- Llegaron a su culminación con DCE (*Distributed Computing Environment*)
- Han evolucionado a invocación de métodos remotos (CORBA, RMI) e invocación de servicios (Web Services)

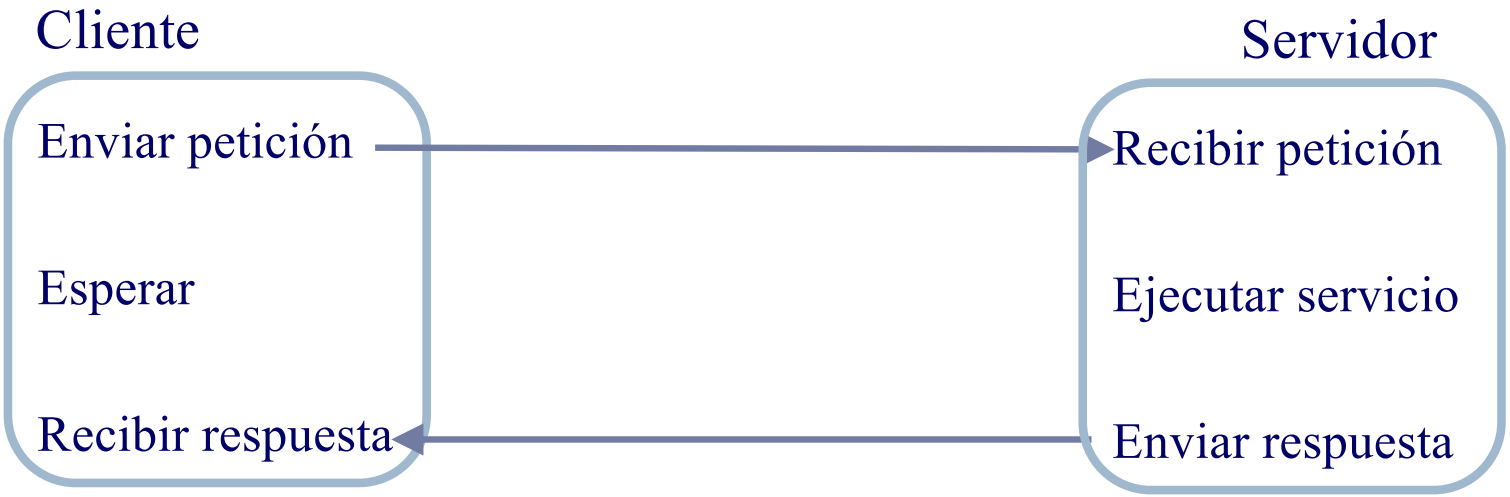
RPC

- Las **RPCs** ofrecen una interfaz sencilla para construir aplicaciones distribuidas sobre TCP/IP

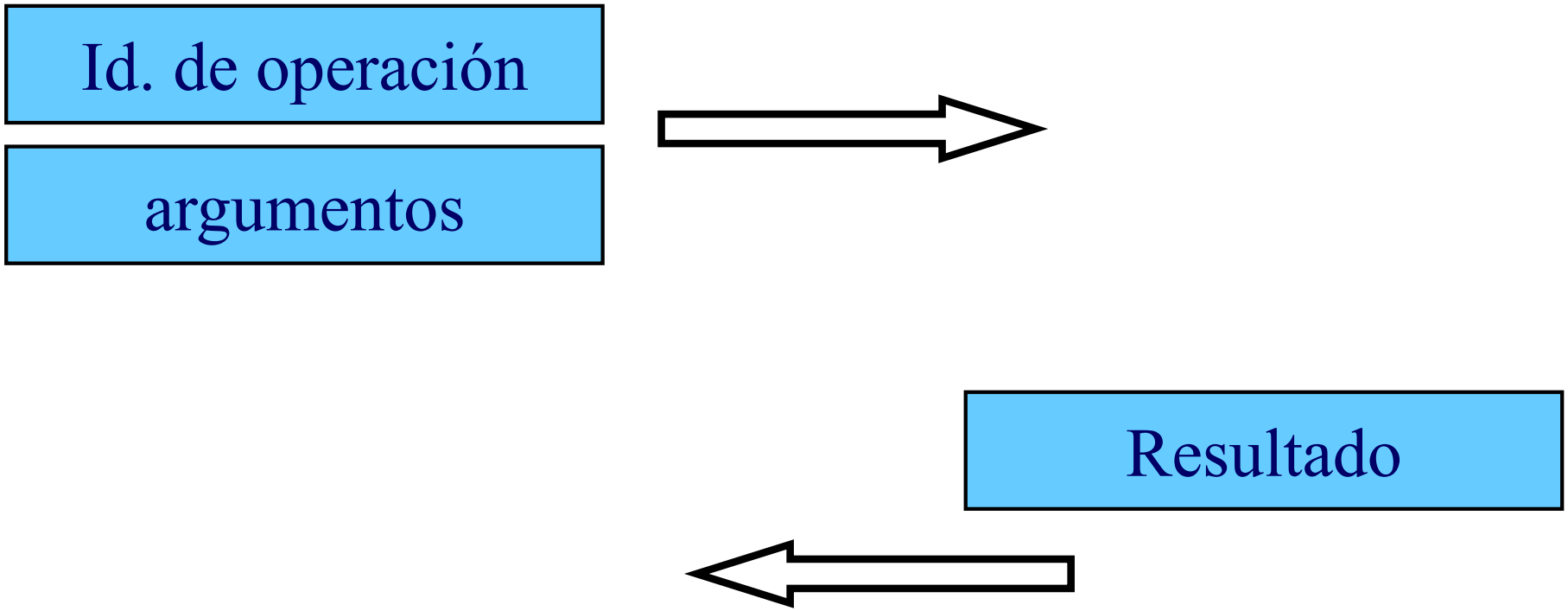


Comunicación cliente-servidor

- Protocolo petición-respuesta



Estructura de los mensajes petición-respuesta

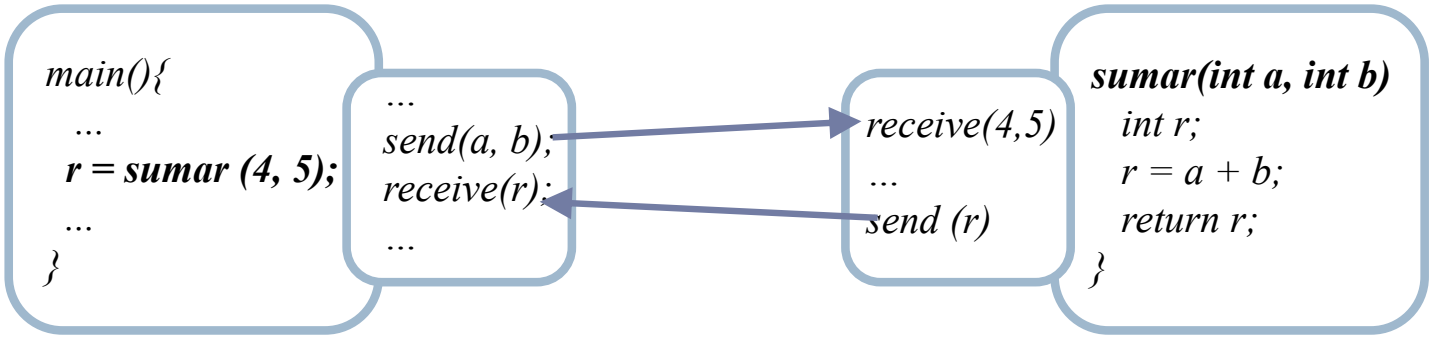


Funcionamiento de las RPC

- El proceso que realiza la llamada empaqueta los argumentos en un mensaje, se los envía a otro proceso y espera el resultado

- El proceso que ejecuta el procedimiento extrae los argumentos del mensaje, realiza la llamada de forma **local**, obtiene el resultado y se lo envía de vuelta al proceso que realizó la llamada

- **Objetivo:** acercar la semántica de las llamadas a procedimiento convencional a un entorno distribuido (**transparencia**)



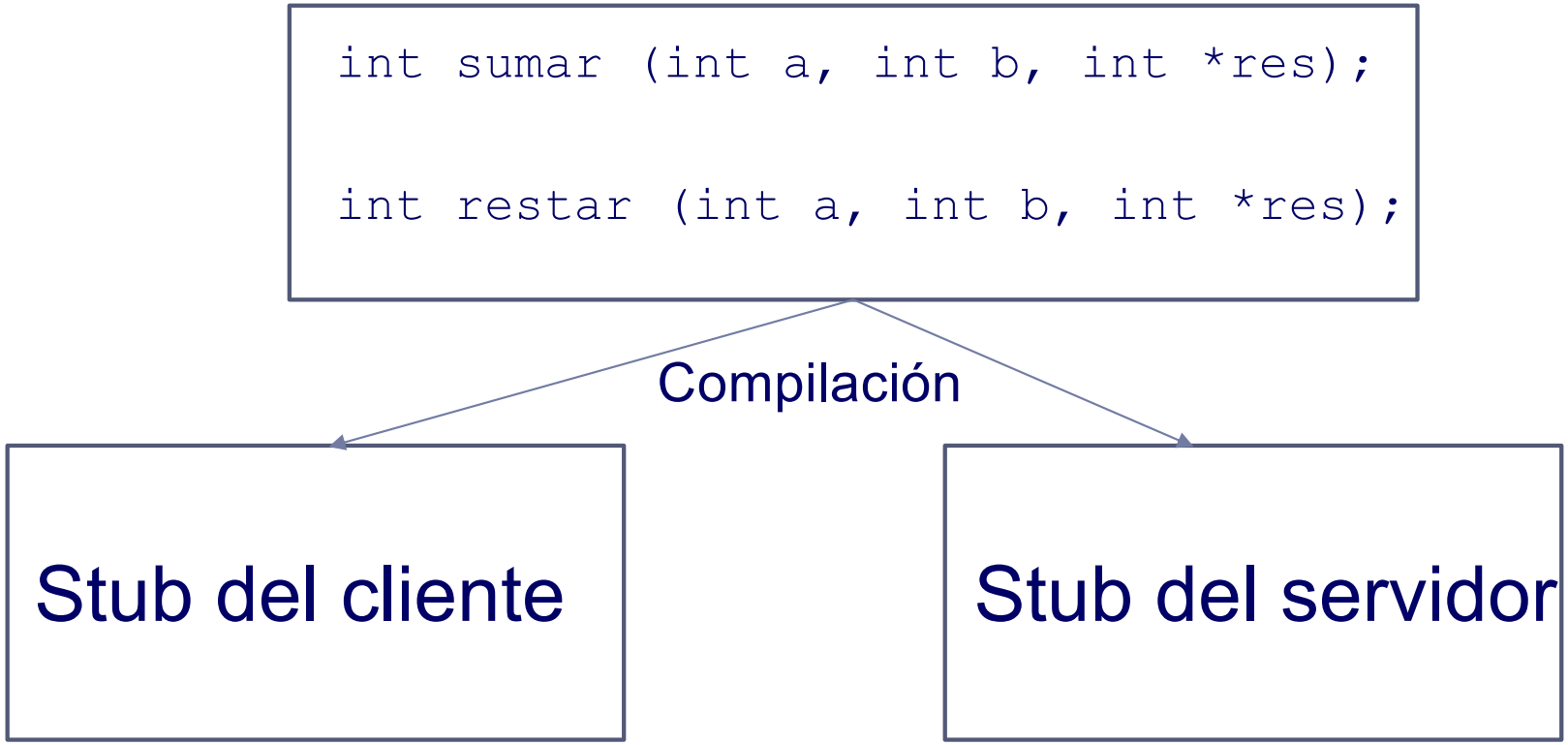
Funcionamiento de las RPC: Definición de la interfaz del servicio

- Definir la interfaz del servicio
  - Operaciones
  - Argumentos de entrada
  - Argumentos de salida
  - Resultados
- Se usa un lenguaje de definición de interfaces
- En el ejemplo anterior una hipotética interfaz podría ser:

```
int sumar (int a, int b, int *res);

int restar(int a, int b, int *res);
```

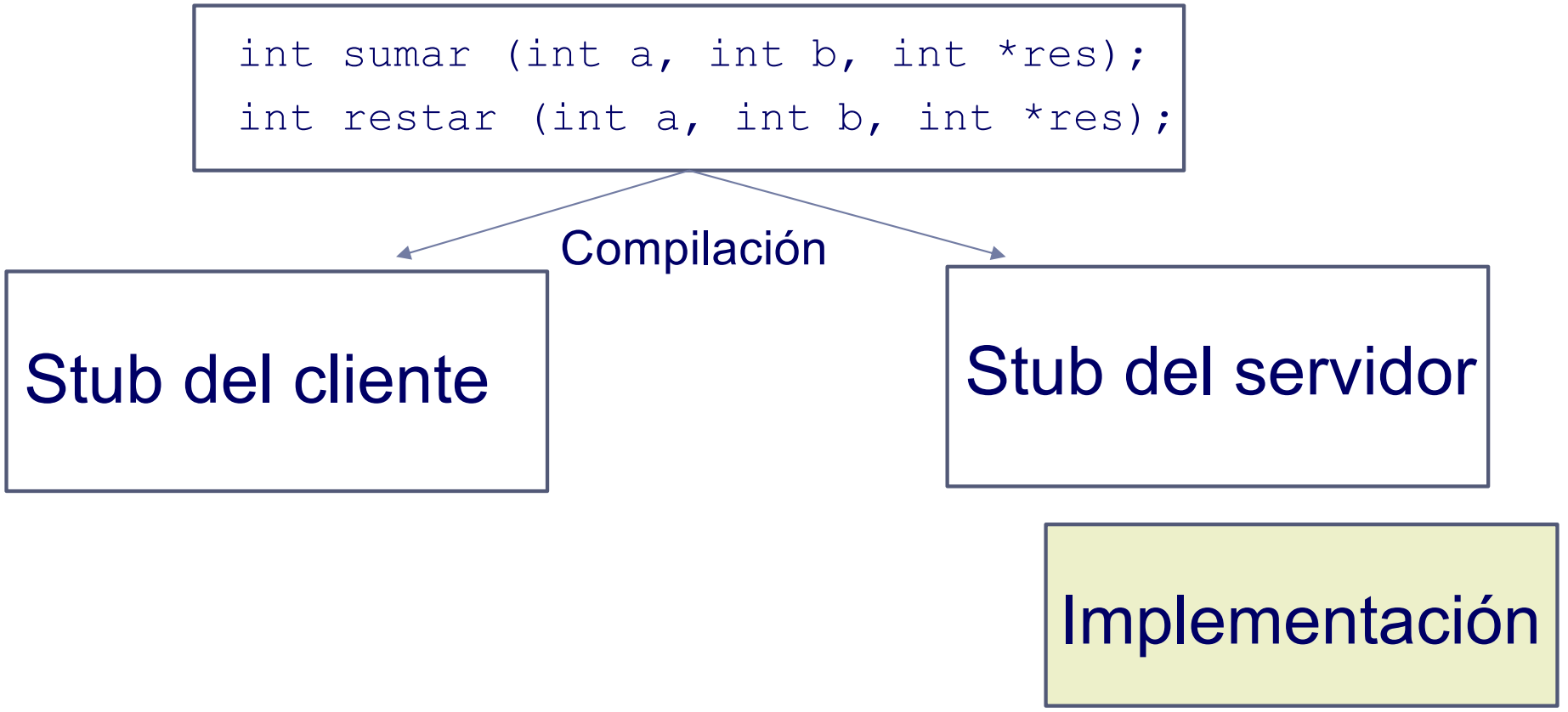
Funcionamiento de las RPC: Compilación de la interfaz

- Se utiliza un compilador de interfaces para generar los archivos del lado del cliente y del servidor que se encargan de las comunicaciones.



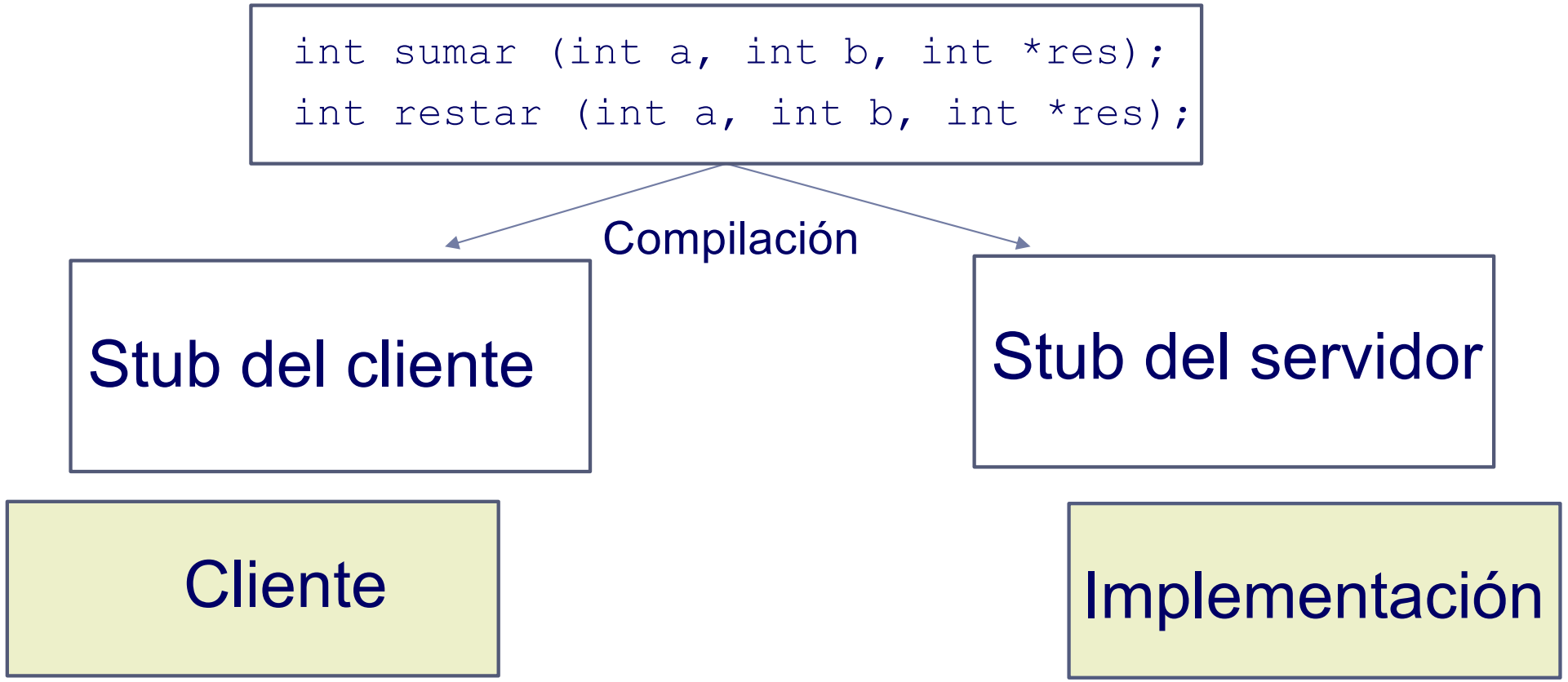
Funcionamiento de las RPC: Implementación del servicio

- Se implementa el servicio en el lado del servidor.



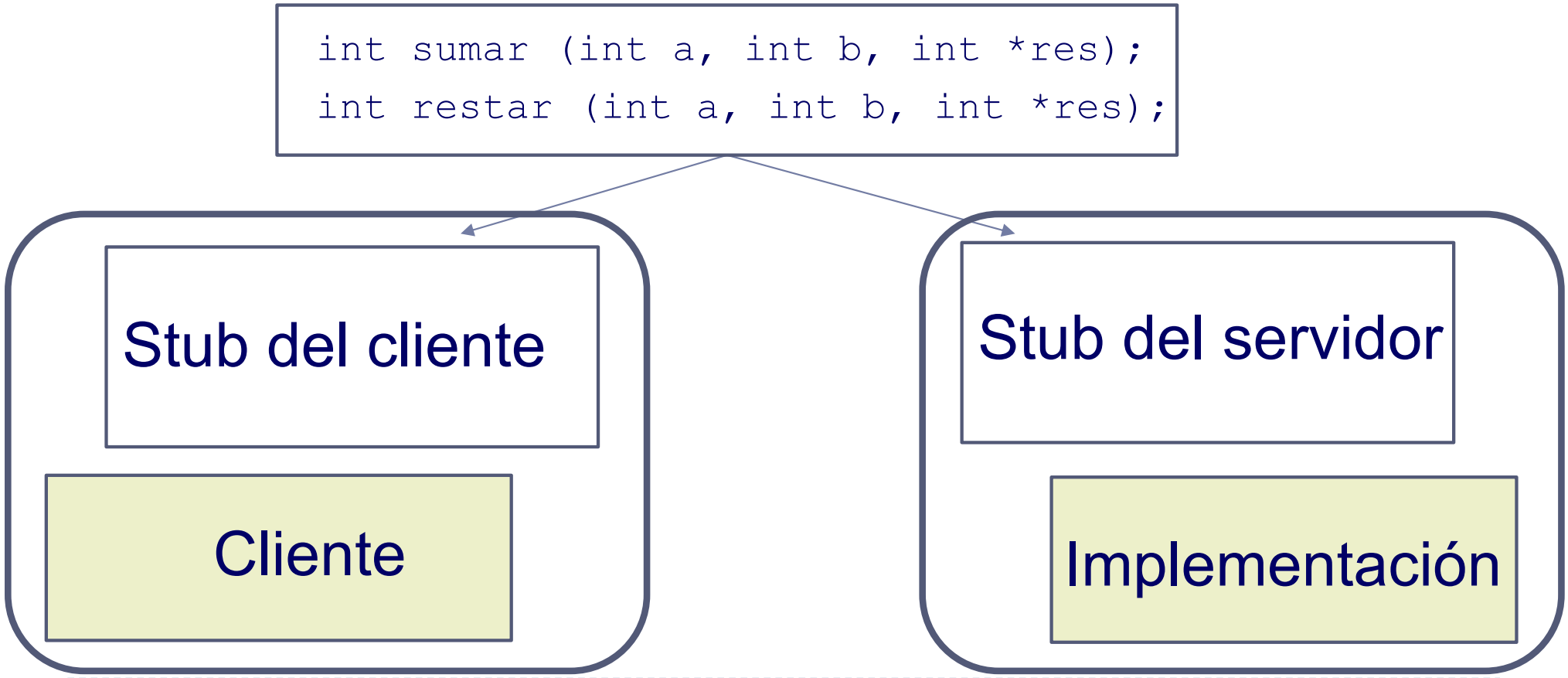
Funcionamiento de las RPC: Implementación del cliente

- Se implementa el cliente que hace uso de las funciones en el lado del cliente



Funcionamiento de las RPC: Generación de los ejecutables

- Se genera el ejecutable del lado del cliente y el ejecutable del lado del servidor



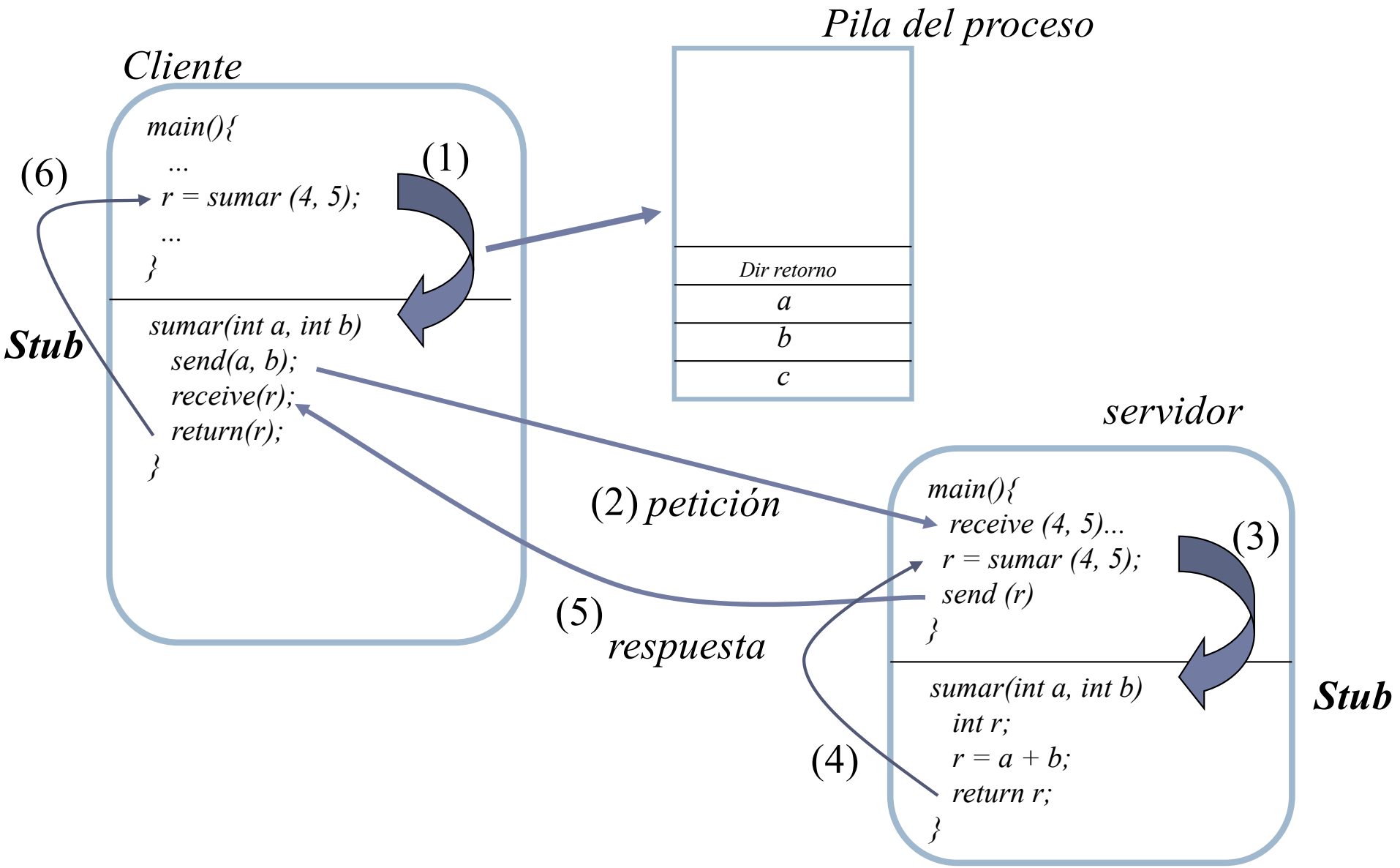
Stub del cliente. Funciones genéricas

```
int sumar (int a, int b, int *res) {
        obtener dirección (IP, puerto) del servidor (Nombre de servicio) => dir;
        convertir formato de argumentos (a,b) => (a', b') ;
        construir mensaje de petición (Cod Op, a', b') => petición;
        send(dir, petición);
        receive(dir, respuesta);
        extraer resultado del mensaje de respuesta => resultado;
        convertir resultado a formato del cliente => resultado';
        if (no hay error){
            * res = resultado';
            return NO_ERROR;
        }
        else
            return ERROR;
}
```

Stub del servidor. Funciones genéricas

```
int main(void){
        dir = obtener dirección (IP, puerto);
        Publicar dirección (nombre de servicio, IP, puerto);
        while (true) {
          dir_cli = receive(ANY, petición);
          extraer código de operación ;
          if (código de operación == SUMAR) {
                  extraer argumentos de petición => (a, b)
                  convertir a formato del servidor (a, b) => (a'b');
                  res = sumar (a', b');
                  convertir formato del resultado => res'
                  construir mensaje respuesta (res', código de error);
                  send(dir_cli, respuesta);
        }
        …..
}
```

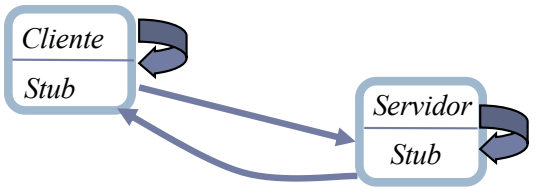
Llamada a procedimiento remoto



Conceptos básicos

- Una RPC tiene dos **participantes**:
  - Un cliente activo, que envía una RPC al servidor
  - Un servidor pasivo, que calcula un resultado y lo devuelve al cliente
- Un **servicio de red** es una colección de uno o más programas remotos
- Un **programa remoto** implementa uno o más procedimientos remotos
- Un servidor puede soportar más de una **versión** de un programa remoto
  - Permite al servidor ser compatible con las actualizaciones de protocolos
- Un **procedimiento remoto,** sus parámetros y sus resultados se especifican en un fichero de especificación del protocolo escrito en un lenguaje de especificación de interfaces (IDL)

Suplentes (*stubs*)

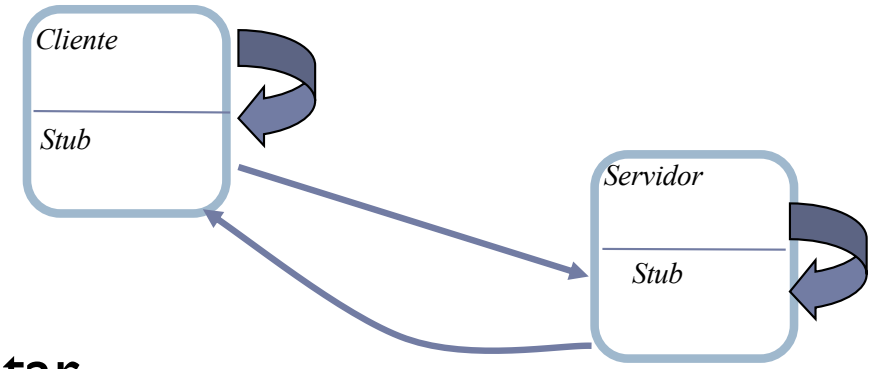


- Se generan **automáticamente** por el software de RPC
  - Stub del cliente
  - Stub del sevidor
- Un **stub (suplente, proxy)** es una pieza de código usada en el cliente y el servidor

- Responsable de convertir los parámetros de la aplicación cliente/servidor durante una llamada a procedimiento remoto
  - Espacio de direcciones independiente del cliente y servidor
  - Representaciones de datos diferentes (big-endian, little-endian)

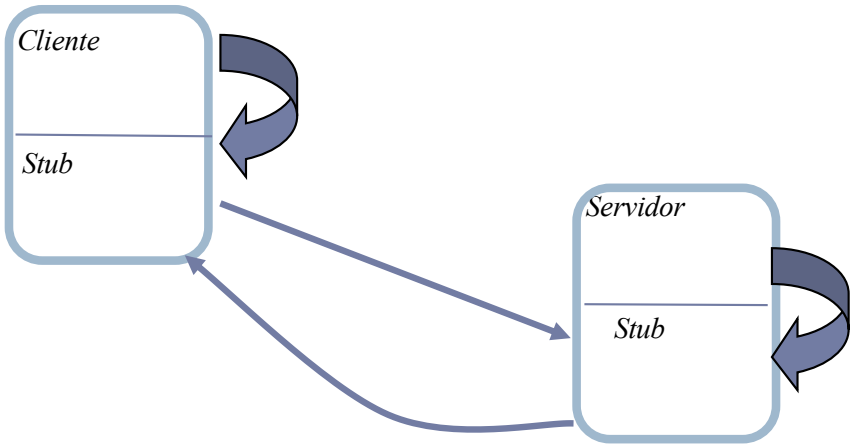
- Los suplentes son **independientes** de la implementación que se haga del cliente y del servidor.
  - Sólo dependen de la interfaz

Suplentes (*stubs*)



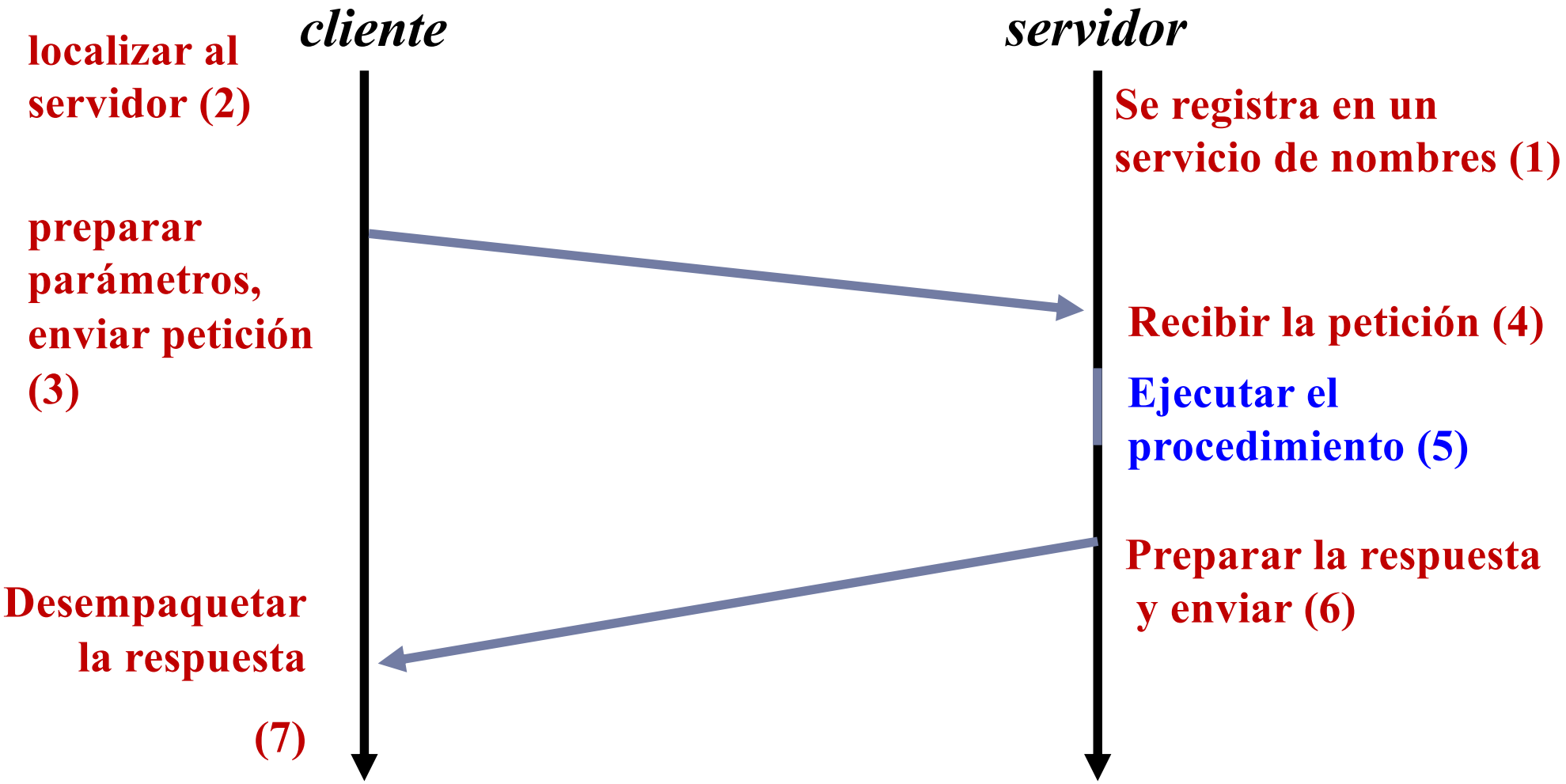
- Funciones en el cliente:
  - Suplantar al procedimiento a ejecutar
  - Localizar al servidor
  - Empaquetar los parámetros y construir los mensajes
  - Enviar el mensaje al servidor
  - Recibir el mensaje de respuesta
  - Desempaquetar los resultados del mensaje de respuesta
  - Devolver los resultados a la función llamante

Suplentes (*stubs*)

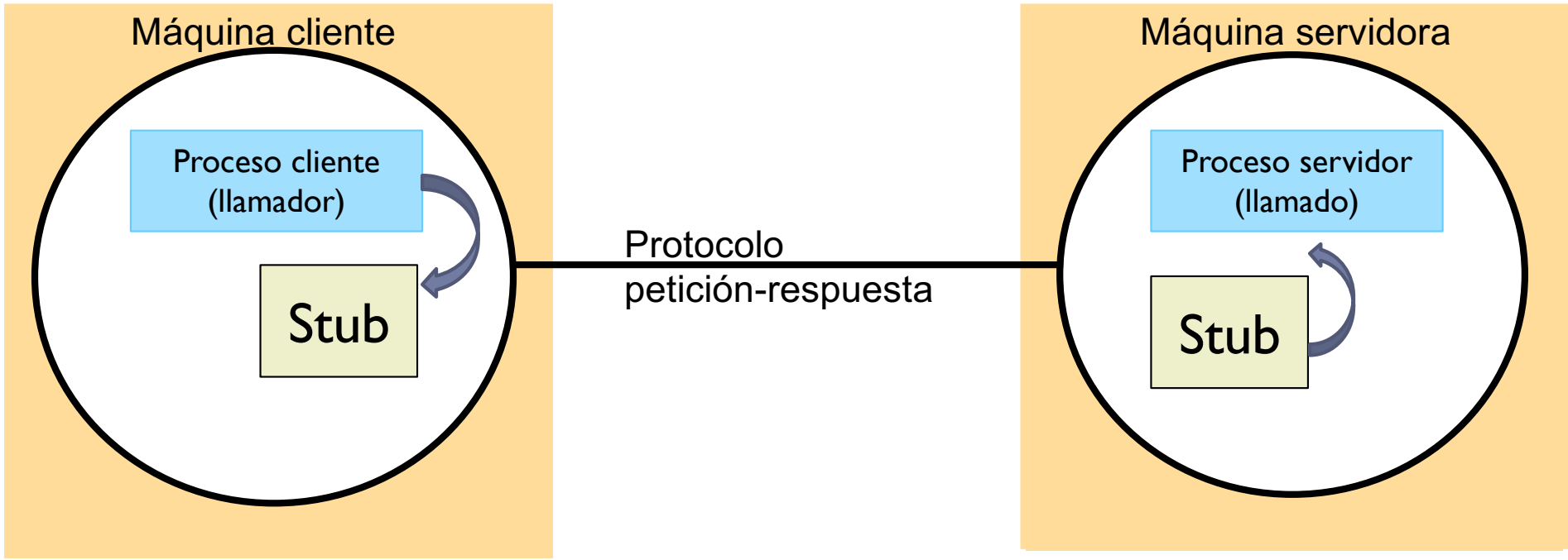


- Funciones en el servidor:
  - Recibir los mensajes de petición
  - Desempaquetar los parámetros
  - Invocar al procedimiento local (procedimiento que implementa el servicio) con los parámetros
  - Recoger los resultados de la ejecución del procedimiento local
  - Empaquetar el resultado en un mensaje de respuesta
  - Enviar el mensaje al *stub* del cliente

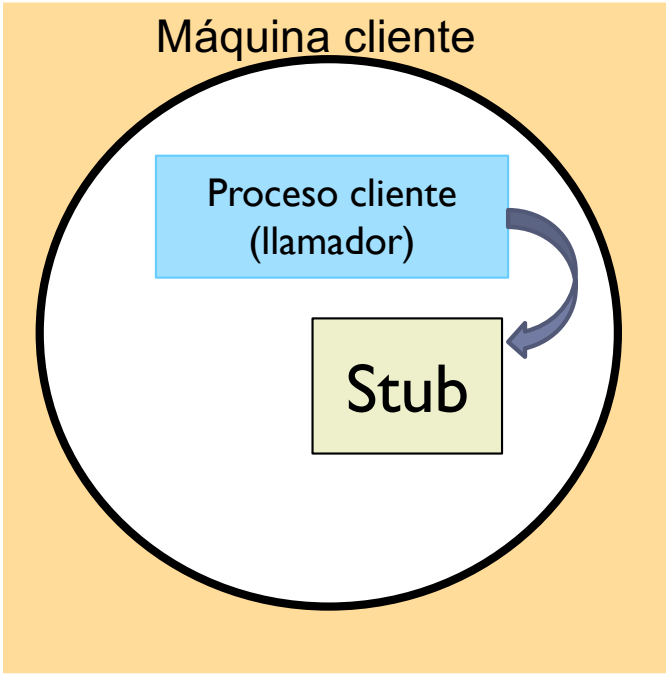
RPC: protocolo básico



RPC: protocolo básico



RPC: protocolo básico



Proceso cliente (llamador)

- **Conectar** al servidor
- **Invocar** una llamada a procedimiento remoto

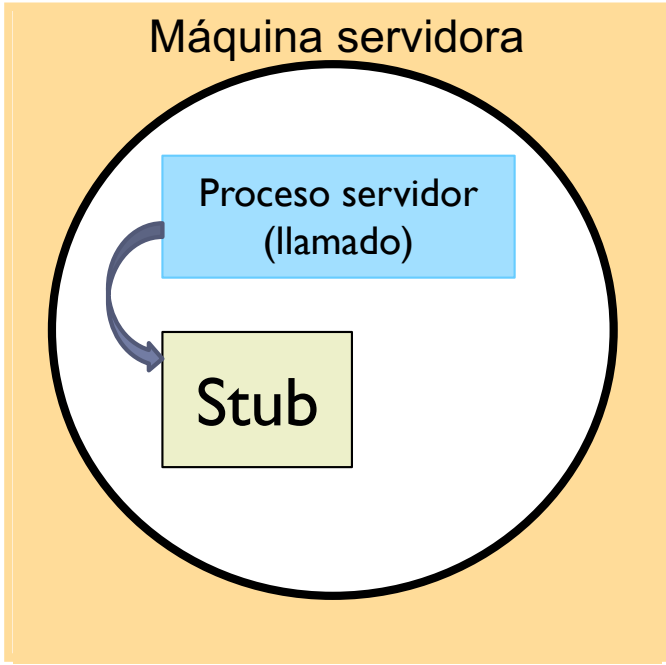
  Stub o suplente del cliente:
  - Localizar al servidor
  - Empaquetar los parámetros y construir los mensajes
  - Enviar los mensajes al servidor
  - Bloquearse hasta esperar la respuesta
- **Obtener** la respuesta

RPC: protocolo básico

Proceso servidor (llamado)

- **Registrar** las RPCs
- **Implementar** los procedimientos

  Stub o suplente del servidor:
  - Recibir la petición del cliente
  - Desempaquetar los parámetros
  - Invocar el procedimiento de manera local
- **Obtener** la respuesta y enviarla al cliente

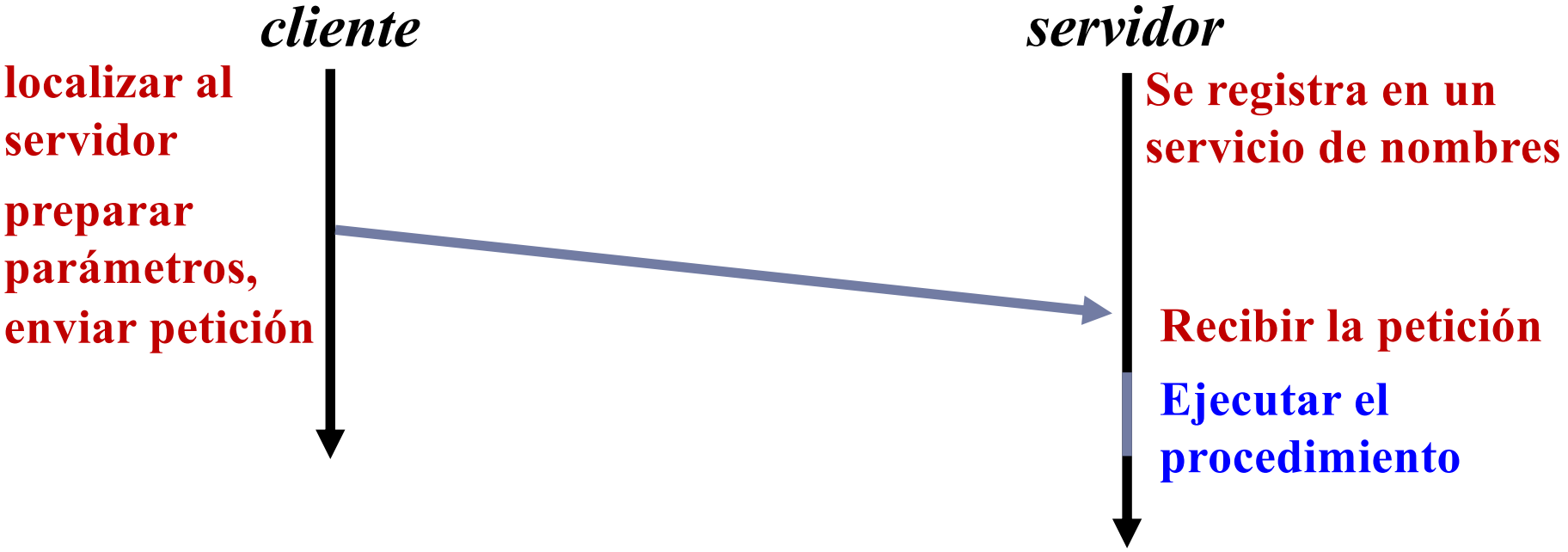


Modelos de RPCs

- RPCs síncronas
  - petición
  - respuesta
- RPCs asíncronas
  - solo petición

RPC asíncronas

- El cliente **no** espera la respuesta
- No admite parámetros de salida
- Ejemplo en CORBA:
  - Se corresponden con métodos especificados como ***oneway***



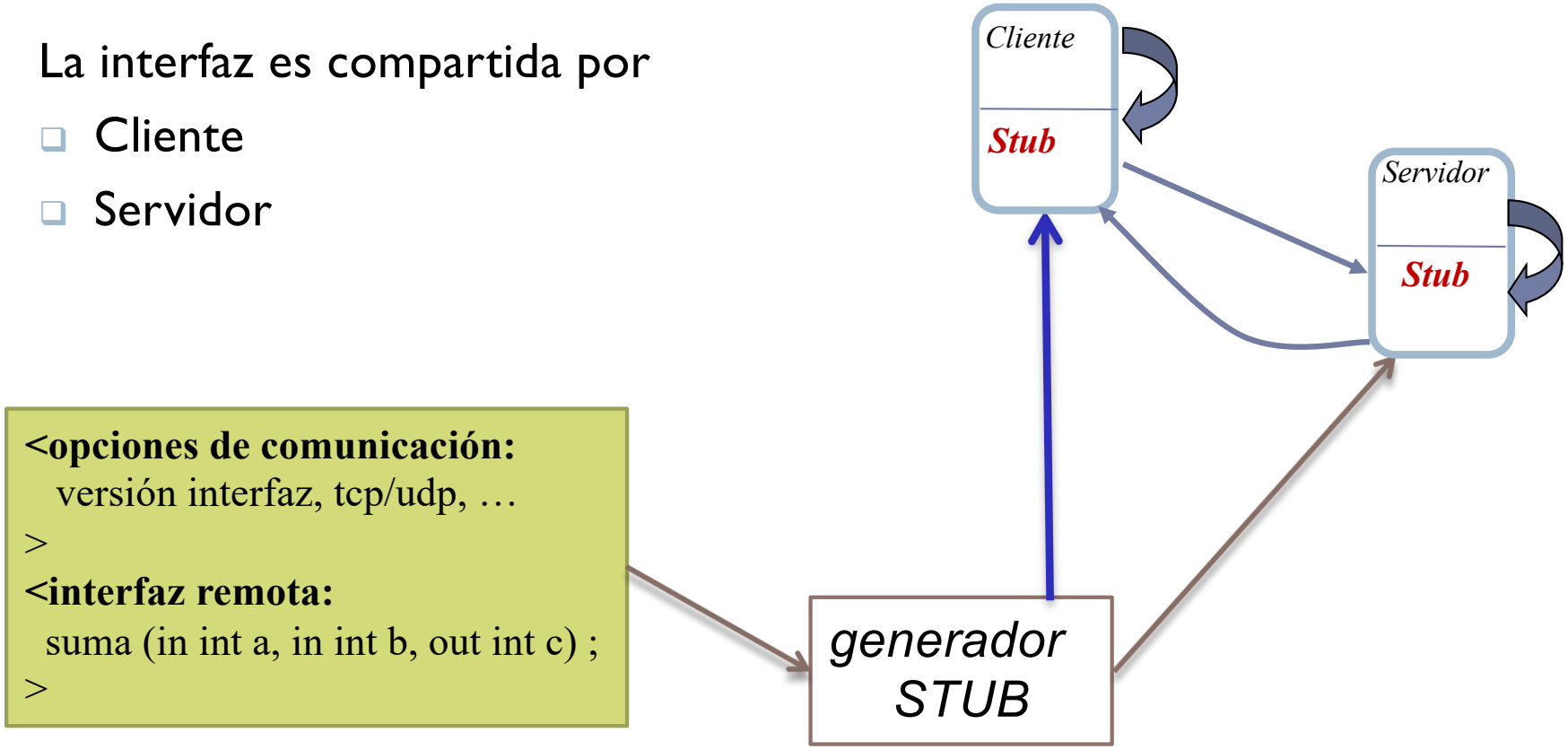
Aspectos relacionados con las RPC

- Lenguaje de definición de interfaces (**IDL**)
- Generación de **stubs**
- Tipos de **parámetros**
- Transferencia de parámetros
- **Protocolo** de comunicación
- **Enlace** entre el cliente y el servidor (**binding**)
- Semántica de las RPC en presencia de fallos
- Autenticación

Lenguaje de Definición de Interfaces

- Un **Lenguaje de Definición de Interfaz** (IDL) permite especificar el formato de los procedimientos remotos y otras opciones de comunicación

- La interfaz es compartida por
  - Cliente
  - Servidor



Lenguaje de Definición de Interfaces

- Una **interfaz** especifica:
  - Nombre de servicio que utilizan los clientes y servidores
  - Nombres de procedimientos
  - Parámetros de los procedimientos
    - Entrada
    - Salida
  - Tipos de datos de los argumentos

- Los compiladores pueden diseñarse para que los clientes y servidores se escriban en lenguajes diferentes
- Tipos de IDL
  - Integrado con un lenguaje de programación (Cedar, Argus, RMI de Java)
  - Lenguaje de definición de interfaces específico para describir las interfaces entre los clientes y los servidores (RPC de Sun/ONC, RPC de DCE, CORBA, Google RPC)

Tipos de parámetros

- Parámetro **de entrada** (*in*)
  - El parámetro se envía del cliente al servidor
- Parámetro **de salida** (*out*)
  - El parámetro se envía del servidor al cliente
- Parámetro de **entrada/salida** (*inout*)

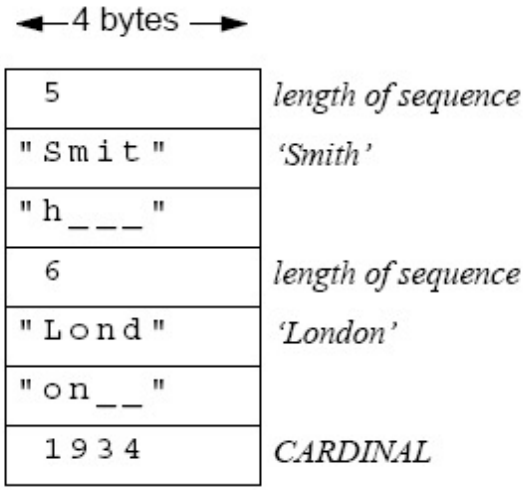
Transferencia de parámetros

- Una de las funciones de los suplentes es empaquetar los parámetros en un mensaje: **aplanamiento** (*marshalling*)
- **Problemas** en la representación de los datos:
  - Servidor y cliente pueden ejecutar en máquinas con arquitecturas distintas (ordenamiento de bytes)
  - Problemas con los punteros
    - Una dirección sólo tiene sentido en un espacio de direcciones
- Ejemplos de esquemas de representación de datos:
  - **XDR** (e***X****ternal* ***D****ata* ***R****epresentation*) es un estándar que define la representación de tipos de datos (**RFC 1832**)
  - Representación común de datos de CORBA (**CDR**)
  - Serialización de objetos de Java
  - **XML** (e***X****tensible* ***M****arkup* ***L****anguage*) es un metalenguaje basado en etiquetas definida por W3C
  - **JSON** (*JavaScript Object Notation)* formato de texto ligero para el intercambio de datos

Ejemplo de representación de datos

- Mensaje: 'Smith', 'London', 1934

XDR



CDR

index in sequence of bytes	4 bytes	*notes*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

XML

```
<person>
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
</person >
```

JSON

{"Name":"Smith", "Place":"London, "Year":1934}

Protocolo

- Entre cliente y servidor debe establecerse un **protocolo**:
  - Formato de los mensajes
  - Formato de representación

Conversión de datos

- Conversión de datos asimétrica
  - El cliente convierte a la representación de datos del servidor
- Conversión de datos simétrica
  - El cliente y el servidor convierten a una representación de datos estándar
- ONC-RPC utiliza representación de datos simétrica

Tipado

- Tipado explícito
  - Cada dato enviado incluye el valor y su tipo
- Tipado implícito
  - Cada dato enviado solo incluye el valor, no se incluye el tipo
- ONC-RPC utiliza tipado implícito

Servicio de enlace *(Binding)*

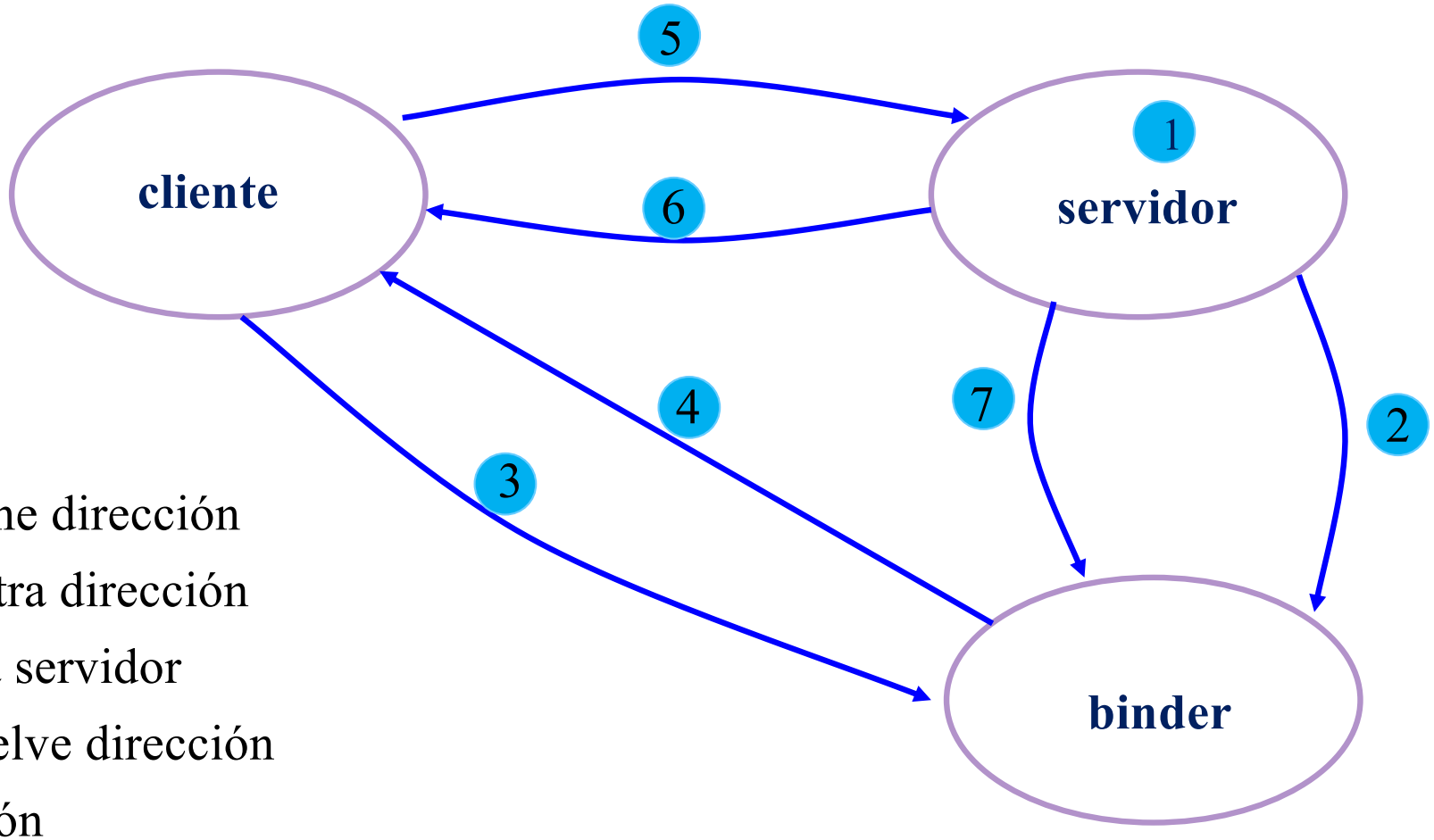
- **Servicio de enlace**: servicio que permite establecer la asociación entre el cliente y el servidor
  - Implica localizar al servidor que ofrece un determinado servicio
  - El servidor debe registrar su dirección en un servicio de nombres (**binder**)

Enlazador dinámico

- **Enlazador dinámico (binder)**: es el servicio que mantiene una tabla de traducciones entre nombres de servicio y direcciones.
- Incluye funciones para:
  - Registrar un nombre de servicio
  - Eliminar un nombre de servicio
  - Buscar la dirección correspondiente a un nombre de servicio

- Cómo **localizar** al enlazador dinámico:
  1. Ejecuta en una dirección fija de un computador fijo
  2. El sistema operativo se encarga de indicar su dirección
  3. Difundiendo un mensaje (*broadcast*) cuando los procesos comienzan su ejecución.

Esquema de registro y enlace



1. Obtiene dirección
2. Registra dirección
3. Busca servidor
4. Devuelve dirección
5. Petición
6. Respuesta
7. Borra dirección (fin del servicio)

Tipos de enlace

- Enlace **no persistente**: el enlace entre el cliente y el servidor se establece en cada RPC
  - Más tolerante a fallos
  - Permite migración de servicios
- Enlace **persistente**: el enlace se mantiene después de la primera RPC
  - Útil en aplicaciones con muchas RPC repetidas
  - Problemas si lo servidores cambian de lugar
- Modelos **híbridos**

Fallos que pueden aparecer con las RPC

- El cliente no es capaz de localizar al servidor
- Pérdidas de mensajes
  - Se pierde el mensaje de petición del cliente al servidor
  - Se pierde el mensaje de respuesta del servidor al cliente
- El servidor falla después de recibir una petición
- El cliente falla después de enviar una petición

Cliente no puede localizar al servidor

- Posibles causas:
  - El servidor puede estar caído
  - El cliente puede estar usando una versión antigua del servidor
    - La versión ayuda a detectar accesos a copias obsoletas

- Cómo indicar el **error** al cliente
  - Devolviendo un código de error (**-1**)
    - No es transparente
      - Ejemplo: sumar(a,b)
  - Elevando una excepción
    - Necesita un lenguaje que tenga excepciones

Pérdida de mensajes del cliente

- Es la más fácil de tratar
- Se activa una alarma (timeout) después de enviar el mensaje
- Si pasado el timeout no se recibe una respuesta se **retransmite el mensaje**

Pérdidas en los mensajes de respuesta

- Más difícil de tratar
- Se pueden emplear alarmas y retransmisiones, pero:
  - ¿Se perdió la petición?
  - ¿Se perdió la respuesta?
  - ¿El servidor va lento?

- Algunas operaciones pueden repetirse y devolverán el mismo resultado (operaciones **idempotentes**)
  - Una transferencia bancaria **no** es idempotente
  - La suma de dos números es idempotente

- La solución con **operaciones no idempotentes** es descartar peticiones ya ejecutadas
  - Un número de secuencia en el cliente
  - Un campo en el mensaje que indique si es una petición original o una retransmisión

Fallos en los servidores

- El servidor no ha llegado a ejecutar la operación
  - Se podría retransmitir
- El servidor ha llegado a ejecutar la operación
- El cliente no puede distinguir los dos
- ¿Qué hacer?
  - No garantizar nada
  - **Semántica al menos una vez**
    - Reintentar y garantizar que la RPC se realiza al menos una vez
    - No vale para operaciones **no idempotentes**
  - **Semántica a lo más una vez**
    - No reintentar, puede que no se realice ni una sola vez
  - **Semántica de exactamente una**
    - Sería lo deseable

Fallos en los clientes

- La computación está activa pero ningún cliente espera los resultados (**computación huérfana**)
  - Gasto de ciclos de CPU
  - Si el cliente rearranca y ejecuta de nuevo la RPC se pueden crear confusiones

Estrategias ante fallos

- Cliente RPC:
  - Cuando no se recibe la respuesta
    - Reenviar la petición
    - Incluir identificador de petición (número de secuencia para evitar ejecuciones repetidas)
- Servidor RPC:
  - Para hacer frente a peticiones duplicadas
    - Filtrar las peticiones (números de secuencia, etc.)
  - Si se necesita reenviar una respuesta de una petición **no idempotente**:
    - Guardar un histórico de las peticiones anteriormente ejecutadas y su respuesta para no ejecutar de nuevo la petición.

Aspectos de implementación

- **Protocolos RPC**
  - Orientados a conexión
    - Fiabilidad se resuelve a bajo nivel
    - Peor rendimiento
  - No orientados a conexión
  - Uso de un protocolo estándar o uno específico
    - Algunos utilizan TCP o UDP como protocolos básicos

Programación con un paquete de RPC

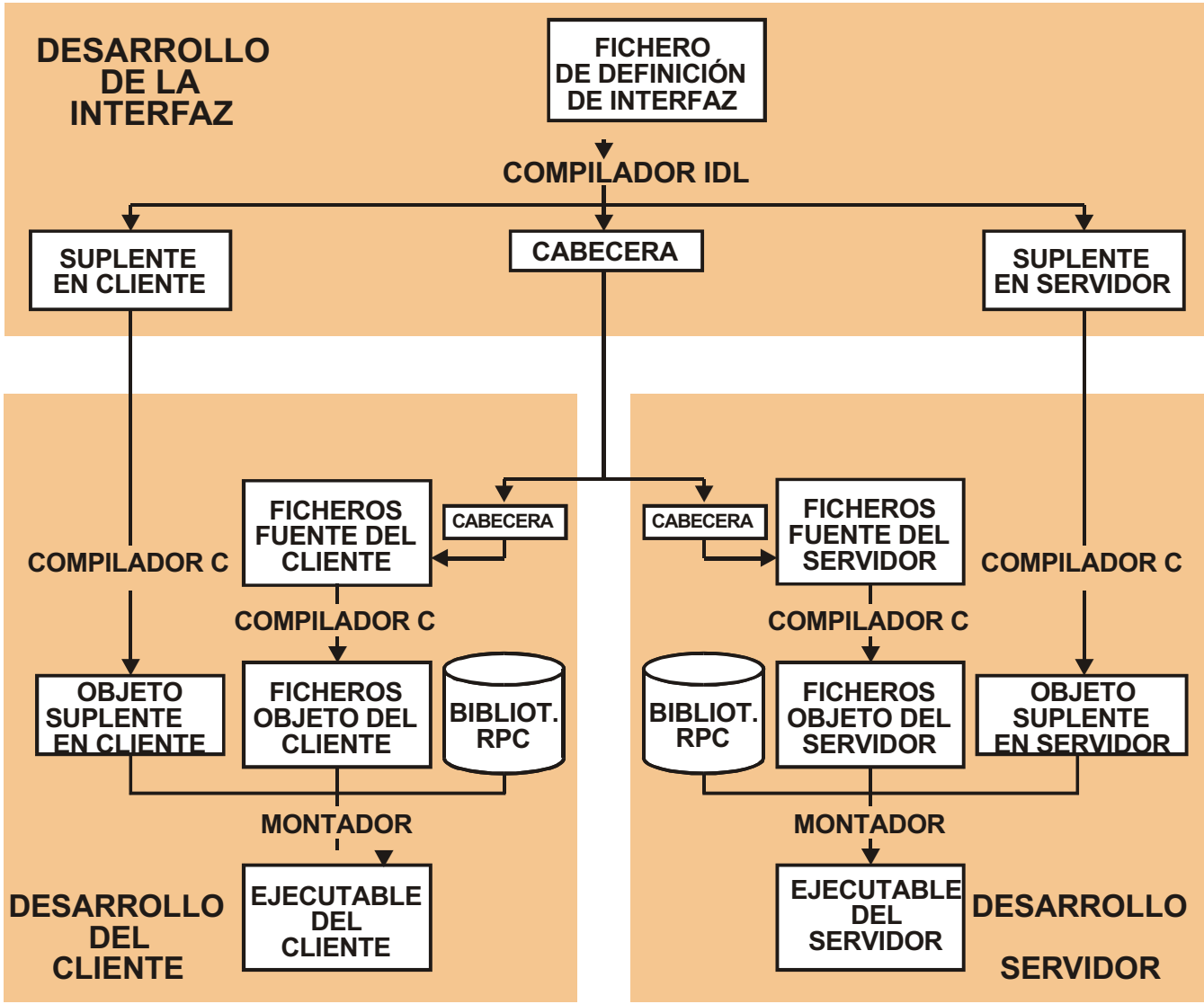
- El programador debe proporcionar:
  - La **definición de la interfaz** (IDL)
    - Nombres de los procedimientos
    - Parámetros que el cliente pasa al servidor
    - Resultados que devuelve el servidor al cliente
  - El **código del cliente**
  - El **código del servidor**

- El compilador de IDL genera :
  - El **suplente** (stub) del cliente
  - El **suplente** (stub) del servidor

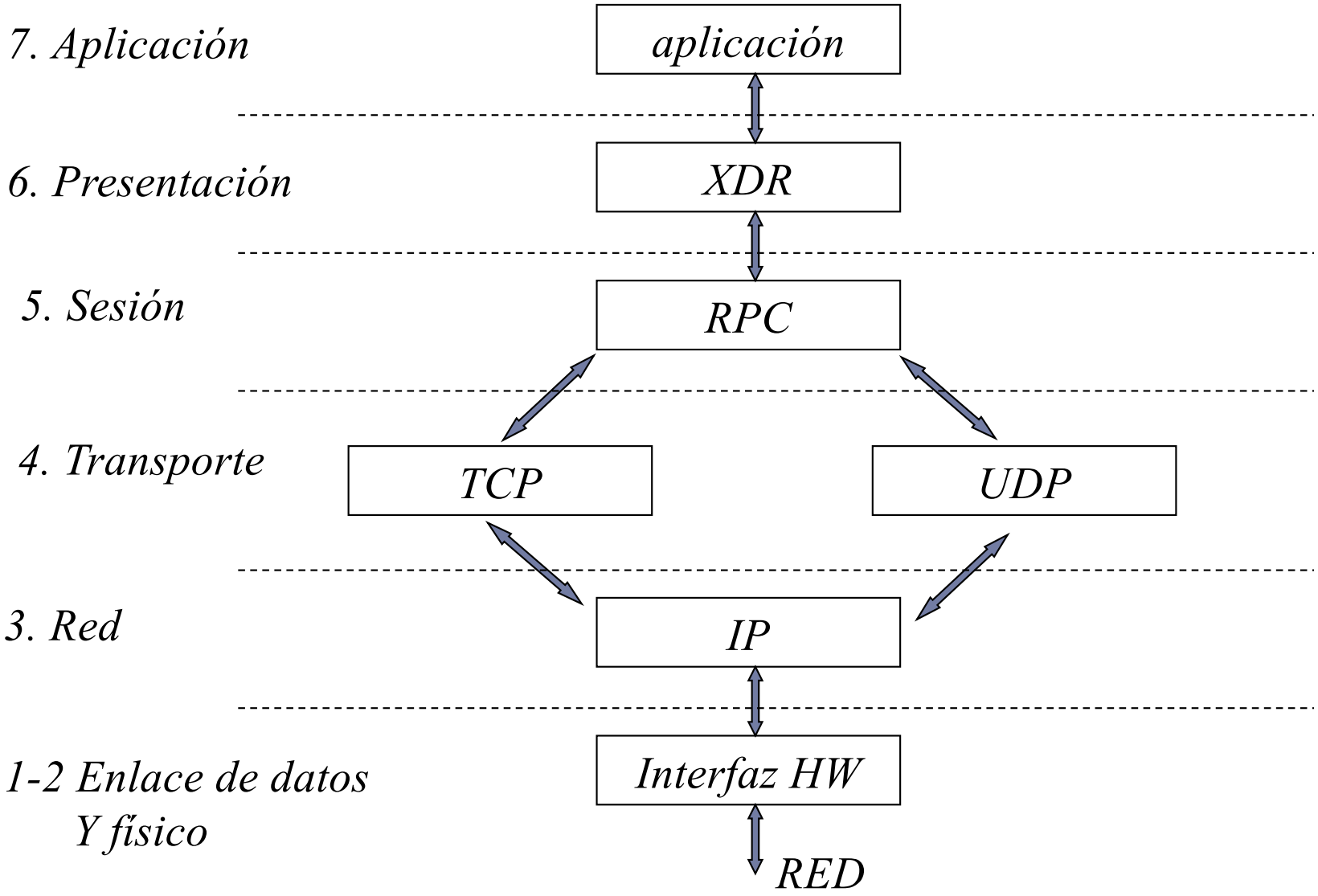
RPC de Sun/ONC

- Diseñado inicialmente para el sistema de ficheros **NFS**
- Descrito en **RFC 1831**
- También se denomina **ONC-RPC** (*Open network computing*)
- Se puede elegir UDP **o** TCP
  - Cliente y servidor deben estar de acuerdo en el protocolo de transporte a utilizar
- Utiliza la semántica **al menos una vez**
- Utiliza un lenguaje de definición de interfaces denominado **XDR**
- Soporte:
  - Para C a través del compilador **rpcgen** en UNIX/Linux
  - Para Java
    - http://sourceforge.net/projects/remotetea/
  - Para plataformas Windows

Desarrollo de la aplicación en C con RPC



RPC de SUN y el modelo OSI

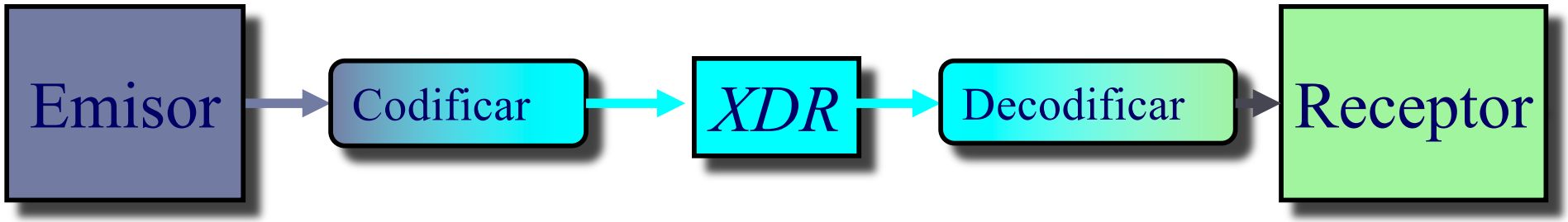


Componentes de ONC-RPC

- Estándar para la representación de datos (XDR)
- Biblioteca de filtros XDR, que convierten los datos de la representación de la máquina local a XDR
- Lenguaje de descripción de RPC (extensión de XDR), que permite describir las interfaces entre el cliente y el servidor (especificación del protocolo)
- Un generador de *stubs*, denominado *rpcgen*. Genera los *stubs* y filtros necesarios
- Biblioteca de funciones de soporte de las RPC
- Un servicio de registro de procedimientos (*portmapper* o *rpcbind*)

Lenguaje XDR

- XDR (eXternal Data Representation) es un estándar que define la representación de tipos de datos (RFC 1832)
- Utilizado inicialmente para representaciones externas de datos
- Se extendió a lenguajes de definición de interfaces

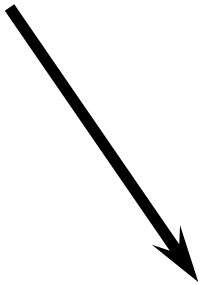


Filtros

- Ejemplo: xdr_init()

Codificación en el emisor

```
int valor;
xdr_int(xstream,&valor);
```



Decodificación en el destino

```
int count;
xdr_int(xstream,&valor);
```

Filtros de datos complejos

- La biblioteca XDR incluye filtros para:
  - Traducir estructuras de datos entre XDR y C
- **`xdr_array()`**
  - Codifica/decodifica arrays de longitud variable
- **`xdr_string()`**
  - filtro de conversión para cadenas de caracteres

Definición de interfaces

- Una **interfaz** contiene:
  - Un número de programa
  - Un número de versión del programa
  - Un conjunto de procedimientos remotos:
  - Un nombre y un número de procedimiento
  - Los procedimientos sólo aceptan un parámetro de entrada (se encapsulan en una estructura)
  - Los parámetros de salida se devuelven mediante un único resultado
  - El lenguaje ofrece una notación para definir:
    - constantes
    - definición de tipos
    - estructuras, uniones
    - programas

Identificación de llamadas a RPC

- Un mensaje de llamada de RPC se identifica unívocamente mediante tres campos enteros sin signo:

  (NUM-PROG, NUM-VERSION, NUM-PROCEDURE)

  NUM-PROG es el número de programa remoto,

  NUM-VERSION es el número de versión de programa,

  NUM-PROCEDURE es el número de procedimiento remoto

  - Detalles de implementación:
    - NUM-PROG se definen en la **RFC 1831**
      - http://www.ietf.org/rfc/rfc1831.txt
    - La primera implementación (versión) de un protocolo deber ser la **1**
    - Los números de procedimientos se especifican por el programador

Ejemplo de definición de interfaz

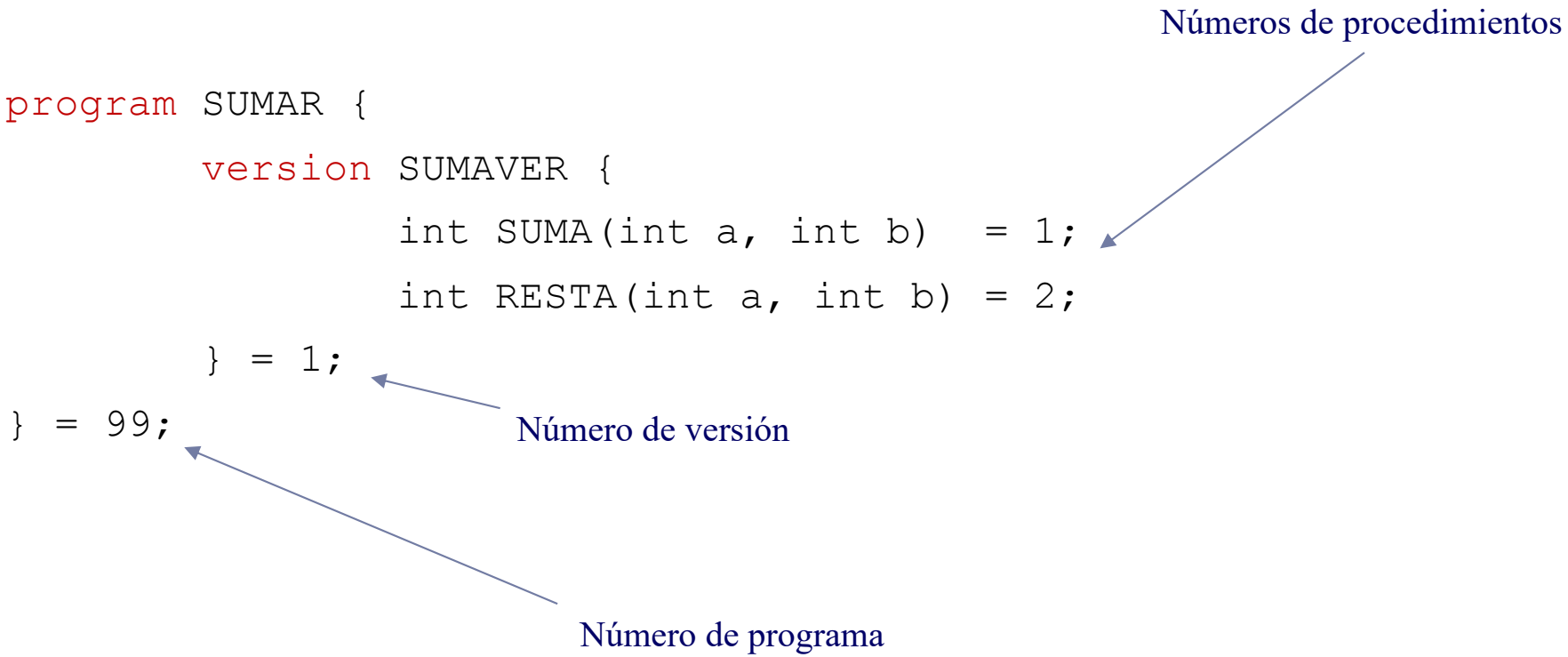
```
struct args{
        int a;
        int b;
};

program SUMAR {
        version SUMAVER {
                int SUMA(struct args a)  = 1;
                int RESTA(struct args a) = 2;
        } = 1;
} = 99;
```

Números de procedimientos

Número de versión

Número de programa

Ejemplo de definición de interfaz con múltiples argumentos



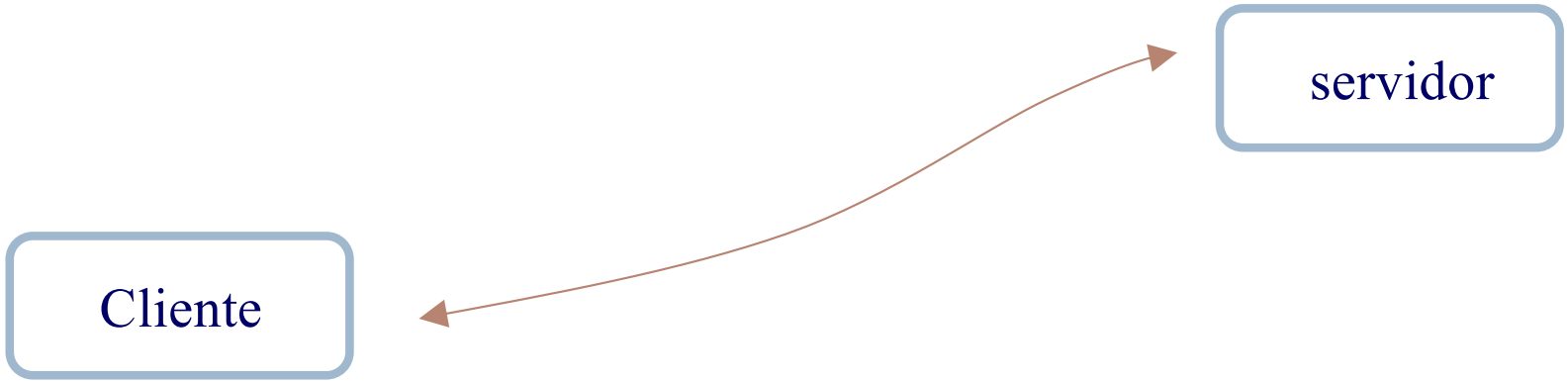
El proceso *rpcbind (portmapper)*

- El enlace en ONC-RPC se realiza mediante un proceso denominado **rpcbind (portmapper)**
- En cada servidor ejecuta un proceso **rpcbind** en un puerto bien conocido (**111**)
- El **rpcbind** almacena por cada servicio local:
  - El número de programa
  - El número de versión
  - El número de puerto
- Enlace dinámico:
  - El número de puertos disponibles es limitado y el número de programas remotos potenciales puede ser muy grande
  - Sólo el rpcbind ejecutará en un puerto determinado (111) y los números de puertos donde escuchan los servidores se averiguan preguntando al rpcbind
- Soporta TCP **y** UDP

El proceso *rpcbind* (*portmapper)*

- Protocolo:
  - Cuando un servidor arranca registra en el *rpcbind local del computador en el que ejecuta el servidor* :
    - El número de programa
    - El número de versión
    - El número de puerto
  - Cuando un cliente necesita invocar un procedimiento remoto envía al **rpcbind** del host remoto (necesita conocer la dirección IP del servidor)
    - El número de programa y el número de versión
    - El proceso *rpcbind* devuelve el puerto del servidor
    - El proceso *rpcbind* ejecuta en el puerto 111

El proceso *rpcbind*



rpcinfo –p guernika.lab.inf.uc3m.es

programa	vers	proto	puerto	
100000	2	tcp	111	portmapper
100000	2	udp	111	portmapper
…				
100024	1	udp	32772	status
100024	1	tcp	59338	status
99	1	udp	46936	
99	1	tcp	40427	

Biblioteca de funciones de RPC

- **rpc.h** es una biblioteca de funciones para desarrollar aplicaciones distribuidas que usan RPC:

```
#include <rpc/rpc.h>
```

- Servicios de RPC para construir aplicaciones
  - En el **cliente**
    - Crear un manejador de cliente
    - Destruir un manejador de cliente
  - En el **servidor**

Servicios de RPC

- Crear un manejador para el cliente

```
CLIENT *
clnt_create (const char *host, const u_long prognum,
   const u_long versnum, const char *nettype)
```

Argumentos:

- host remoto — nombre del host remoto donde se localiza el programa
- prognum — Número de programa del programa remoto
- versnum — Número de versión del programa remoto
- nettype — Protocolo de transporte:
  NETPATH, VISIBLE, CIRCUIT_V, DATAGRAM_V, CIRCUIT_N, DATAGRAM_N, TCP, UDP

Servicios de RPC

- Destruir el manejador del cliente

```
void clnt_destroy (CLIENT *clnt)
```

Argumentos:

- clnt Manejador de RPC del cliente

Servicios de RPC

- Indicar el error en un fallo de RPC:

```
void clnt_perror (CLIENT *clnt, char *s)
void clnt_pcreateerror (CLIENT *clnt, char *s)
```

Argumentos:

- clnt Manejador de RPC del cliente
- S Mensaje de error

Ejemplo: crear/destruir un manejador

```
#include <stdio.h>
#include <rpc/rpc.h>
#define RMTPROGNUM (u_long)0x3fffffffL  /* Define remote program number and version */
#define RMTPROGVER (u_long)0x1
main()
{
   CLIENT *client;                      /* client handle */

   /* Crear el manejador del cliente */
   client = clnt_create("guernika.lab.inf.uc3m.es", RMTPROGNUM, RMTPROGVER, "TCP");
   if (client == NULL){
          clnt_pcreateerror(client, "Could not create client\n");
          exit(1);
   }
   // Invocar procedimiento remoto
   /* Destruir el manejador del cliente */
   clnt_destroy(client);
   exit(0);
}
```

Nombres de procedimiento en el cliente (múltiples argumentos)

- Prototipo de un procedimiento remoto en el cliente:

```
bool_t procedimiento_v   (tipo_arg1 arg1,
                          tipo_arg2 arg2,
                                …
                          tipo_argn argn,
                          CLIENT *clnt);
```

Argumentos de entrada y salida

donde:

- procedimiento_v — Nombre del procedimiento a invocar
- arg1,arg2,…,argn — Argumentos del procedimiento
- clnt — Manejador de un cliente de RPC

NOTA: *v* se sustituye por el número de versión que se invoca

Nombres de procedimientos en el servidor (múltiples argumentos)

- Prototipo del procedimiento remoto a implementar en el servidor:

```
bool_t   procedimiento_v_svc (tipo_arg1 arg1,
                              tipo_arg2 arg2,
                                    …
                              tipo_argn argn,
                              struct svc_req *rqstp)
```

Argumentos de entrada y salida

donde:

- procedimiento_v_svc — Nombre del procedimiento a implementar
- arg1,arg2,…,argn — Argumentos del procedimiento
- rqstp — Estructura que contiene información de la petición

NOTA: *v* se sustituye por el número de versión que se implementa

Compilador de interfaces (rpcgen)

- **rpcgen** es el compilador de interfaces que genera código **C** para:
  - Stub del cliente
  - Stub del servidor y procedimiento principal del servidor
  - Procedimientos para el empaquetado y desempaquetado (filtros XDR)
  - Fichero de cabecera (.h) con los tipos y declaración de prototipos de procedimientos

Sintaxis de rpcgen

- Compilar usando rpcgen

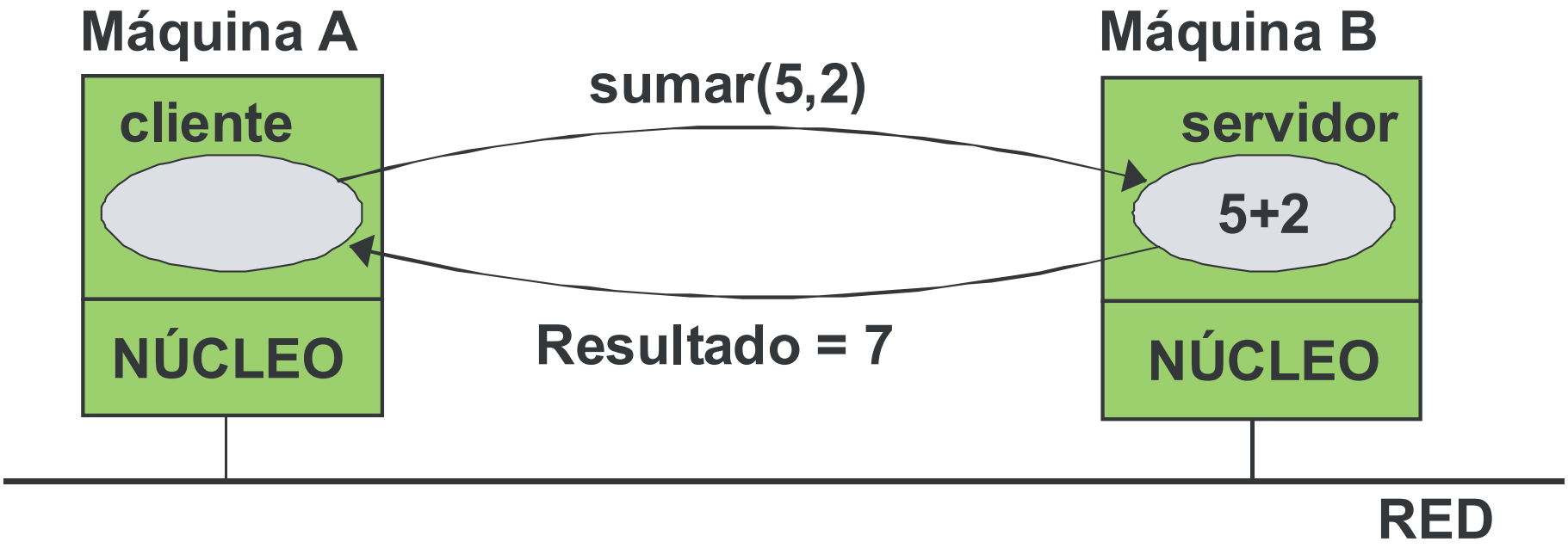
```
rpcgen infile
```

```
rpcgen [-abkCLNTM][-Dname[=value]] [-i size] [-I [-K seconds]] [-Y path] infile
rpcgen [-c | -h | -l | -m | -t | -Sc | -Ss | -Sm] [-o outfile] [infile]
rpcgen [-s nettype]* [-o outfile] [infile]
rpcgen [-n netid]* [-o outfile] [infile]
```

  - **infile** fichero en lenguaje de definición de interfaz de RPC (XDR en ONC-RPC)

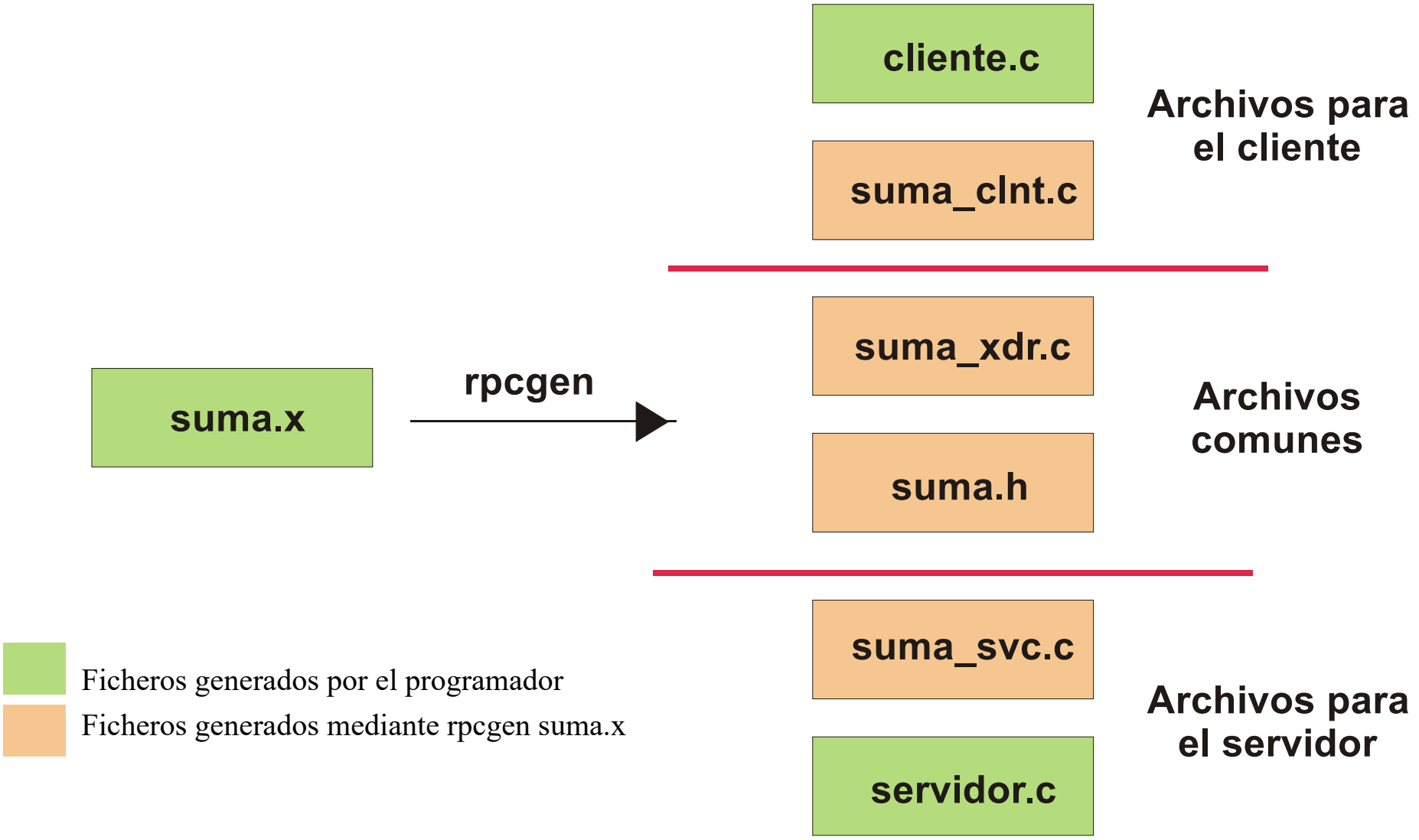
  - Algunas opciones:
    - -N Permite a los procedimientos tener múltiples argumentos
    - -a Genera todos los ficheros incluyendo código de ejemplo para cliente y servidor y Makefile para compilación
    - -M Genera stubs multi-thread para paso de argumentos

Ejemplo: aplicación con RPC



@Fuente: *Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez.* **Mc Graw Hill**

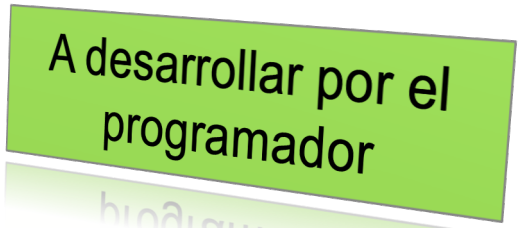
Esquema de la aplicación



Ejemplo: suma.x

suma.x

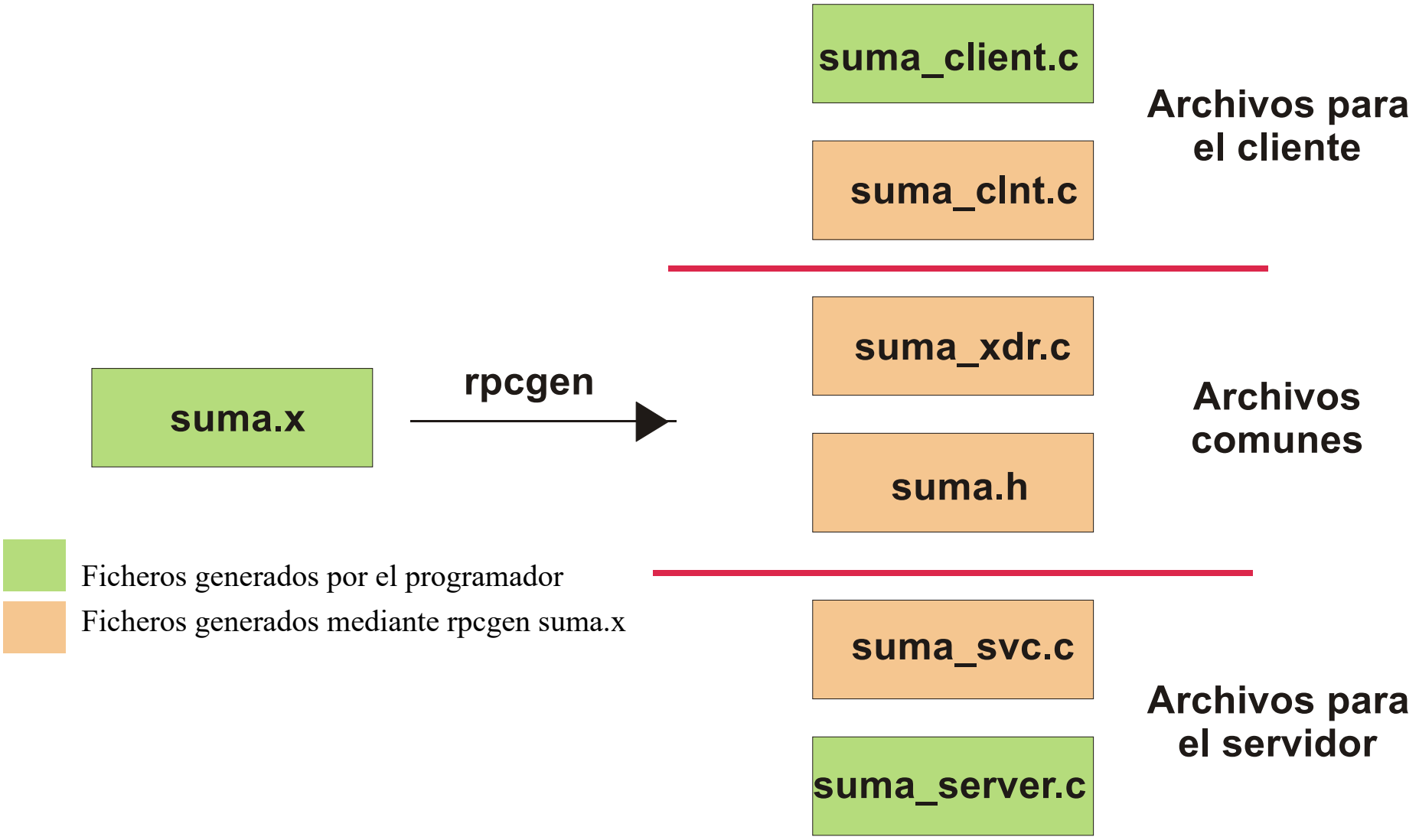
```
program SUMAR {
  version SUMAVER {
       int SUMA(int a, int b)  = 1;
       int RESTA(int a, int b) = 2;
  } = 1;
} = 99;
```



Compilación

- **rpcgen –a –N –M** suma.x
- Opciones:
  - -a: genera todos los ficheros incluyendo ejemplos y Makefile_suma
    - Se compila: make –f Makefile_suma
  - -N: permite varios argumentos en cada llamada (sino, hay que empaquetar los argumentos en un único de tipo struct)
  - -M: genera código *multithread* (código que puede ser utilizado en aplicaciones con varios *threads*)

Esquema de la aplicación



Ejemplo: suma.h

```
#ifndef _SUMA_H_RPCGEN
#define _SUMA_H_RPCGEN

#define SUMAR 99
#define SUMAVER 1

struct suma_1_argument {
        int a;
        int b;
};
typedef struct suma_1_argument suma_1_argument;

struct resta_1_argument {
        int a;
        int b;
};
typedef struct resta_1_argument resta_1_argument;
```

Ejemplo: suma.h (2)

No hace falta tocar

```
#define suma 1
extern  enum clnt_stat suma_1(int , int , int *, CLIENT *);
extern  bool_t suma_1_svc(int , int , int *, struct svc_req *);

#define resta 2
extern  enum clnt_stat resta_1(int , int , int *, CLIENT *);
extern  bool_t resta_1_svc(int , int , int *, struct svc_req *);

extern int sumar_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);...

#endif /* !_SUMA_H_RPCGEN */
```

Ejemplo: suma_server.c

A desarrollar por el programador

Suma_server.c

```
#include "suma.h"
bool_t  suma_1_svc(int a, int b, int *result,  struct svc_req *rqstp){
        bool_t retval;
        /*   insert server code here  */
        *result = a + b;
        retval = TRUE;
        return retval;
}
bool_t resta_1_svc(int a, int b, int *result,  struct svc_req *rqstp){
        bool_t retval;

        /* insert server code here   */
        *result = a - b;
        retval = TRUE;

        return retval;
}
```

Implementación de las funciones

Ejemplo: suma_client.c

A desarrollar por el programador

suma_client.c

```
#include "suma.h"

main( int argc, char* argv[] )
{
   CLIENT *clnt;
   enum clnt_stat retval;
   int res;
   char *host;

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

Ejemplo: suma_cliente.c (II)

A desarrollar por el programador

suma_client.c

```
   /* Paso 1: localizar al servidor */
    clnt = clnt_create(host, SUMAR, SUMAVER, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }


   /* Paso 2: Invocar el procedimiento remoto */
    retval = suma_1(8, 3, &res, clnt);
    if (retval !=  RPC_SUCCESS) {
        clnt_perror(clnt, "call failed:");
    }
   printf("La suma es %d\n", res);

   /* Destruir el manejador */
   clnt_destroy( clnt );
}
```

Binding:
Se contacta con el rpcbind de *host*
Se indica el host:Prog:versión:protocolo
El rpcbin dindica la localización (puerto)

Resultado en res

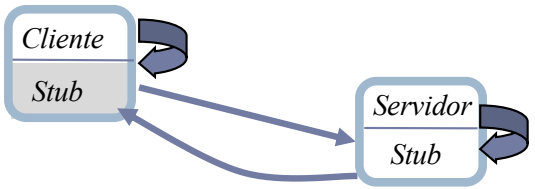
Argumentos de entrada (8, 3)

suma_xdr.c

No hace falta tocar

```
#include "suma.h"

bool_t
xdr_suma_1_argument (XDR *xdrs,
                     suma_1_argument *objp)
{
        if (!xdr_int (xdrs, &objp->a))
                return FALSE;
        if (!xdr_int (xdrs, &objp->b))
                return FALSE;
        return TRUE;
}
.....
```

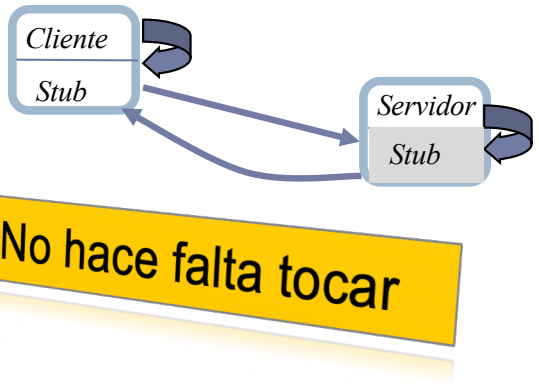
suma_clnt.c (stub del cliente)



No hace falta tocar

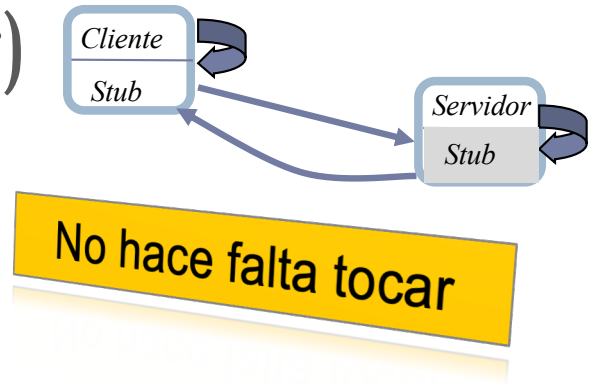
```
#include <memory.h> /* for memset */
#include "suma.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

enum clnt_stat  suma_1(int a, int b, int *clnt_res,  CLIENT *clnt) {
        suma_1_argument arg;
        arg.a = a;
        arg.b = b;

        return (clnt_call (clnt, suma, (xdrproc_t) xdr_suma_1_argument,
                  (caddr_t) &arg, (xdrproc_t) xdr_int, (caddr_t) clnt_res,
                    TIMEOUT));
}
enum clnt_stat  resta_1(int a, int b, int *clnt_res,  CLIENT *clnt){
. . . .
```

suma_svc.c (stub del servidor)



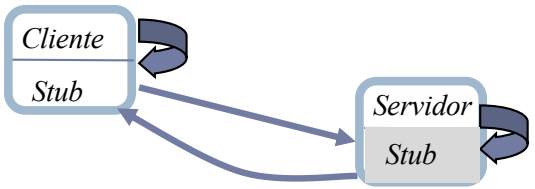

```
int main (int argc, char **argv)
{
        register SVCXPRT *transp;
        pmap_unset (SUMAR, SUMAVER);

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, SUMAR, SUMAVER, sumar_1, IPPROTO_UDP)) {
             fprintf (stderr, "%s", "unable to register (SUMAR, SUMAVER, udp).");
             exit(1);
        }
```

suma_svc.c (stub del servidor)




```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
}
if (!svc_register(transp, SUMAR, SUMAVER, sumar_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register (SUMAR, SUMAVER, tcp).");
        exit(1);
}

svc_run ();
fprintf (stderr, "%s", "svc_run returned");
exit (1);
```

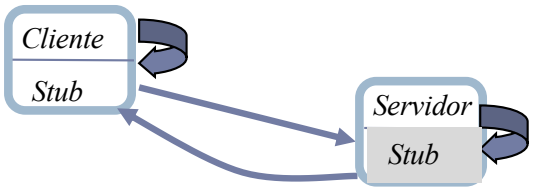
suma_svc.c (stub del servidor)



No hace falta tocar

```
int
_suma_1 (suma_1_argument *argp, void *result, struct svc_req
*rqstp)
{
        return (suma_1_svc(argp->a, argp->b, result, rqstp));
}

int
_resta_1 (resta_1_argument *argp, void *result, struct svc_req
*rqstp)
{
        return (resta_1_svc(argp->a, argp->b, result, rqstp));
}
```

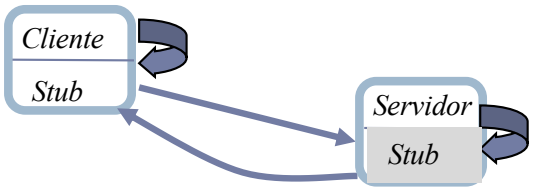
suma_svc.c (stub del servidor)



No hace falta tocar

```
static void sumar_1(struct svc_req *rqstp, register SVCXPRT *transp)
        union {
                suma_1_argument suma_1_arg;
                resta_1_argument resta_1_arg;
        } argument;
        union {
                int suma_1_res;
                int resta_1_res;
        } result;
        bool_t retval;
        xdrproc_t _xdr_argument, _xdr_result;
        bool_t (*local)(char *, void *, struct svc_req *);

        switch (rqstp->rq_proc) {
        case NULLPROC:
                (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
                return;
```

suma_svc.c (stub del servidor)



No hace falta tocar

```
        case suma:
                _xdr_argument = (xdrproc_t) xdr_suma_1_argument;
                _xdr_result = (xdrproc_t) xdr_int;
                local = (bool_t (*) (char *, void *,  struct svc_req *))_suma_1;
                break;

        case resta:
                _xdr_argument = (xdrproc_t) xdr_resta_1_argument;
                _xdr_result = (xdrproc_t) xdr_int;
                local = (bool_t (*) (char *, void *,  struct svc_req *))_resta_1;
                break;

        default:
                svcerr_noproc (transp);
                return;
        }
....
```

Compilación

- Make –f Makefile_suma:
  - gcc –c suma_xdr.c
  - gcc –c suma_svc.c
  - gcc –c suma_clnt.c
  - gcc –c suma_client.c
  - gcc –c suma_server.c

- gcc suma_xdr.o suma_clnt.o suma_client.o –o **cliente**
- gcc suma_xdr.o suma_svc.o suma_server.c –o **servidor**

Estándar XDR

- Utiliza representación *big-endian*
  - No utiliza protocolo de negociación
- Todos los elementos en XDR son múltiplos de 4 bytes
- La transmisión de mensajes utiliza tipado implícito. El tipo de datos no viaje con el valor
- Utiliza conversión de datos simétrica
- Los datos se codifican en un flujo de bytes

Definición de constantes simbólicas

- En XDR

```
const MAX_SIZE  = 8192;
```

- Traducción a C:

```
#define MAX_SIZE 8192
```

Tipos de datos en XDR

- Tipos de datos básicos:

  Enteros con signo (4 bytes):

  Declaración: `int a;`

  Equivalente en C: `int a;`

  Enteros sin signo (4 bytes):

  Declaración: `unsigned a;`

  Equivalente en C: `unsigned int a;`

  Números en coma flotante:

  Declaración: `float a;  double c;`

  Equivalente en C: `float c;  double d;`

Tipos de datos en XDR

Cadenas de caracteres de longitud fija:

Declaración:
```
string a<37>;
```
Equivalente en C:
```
char *a;
```

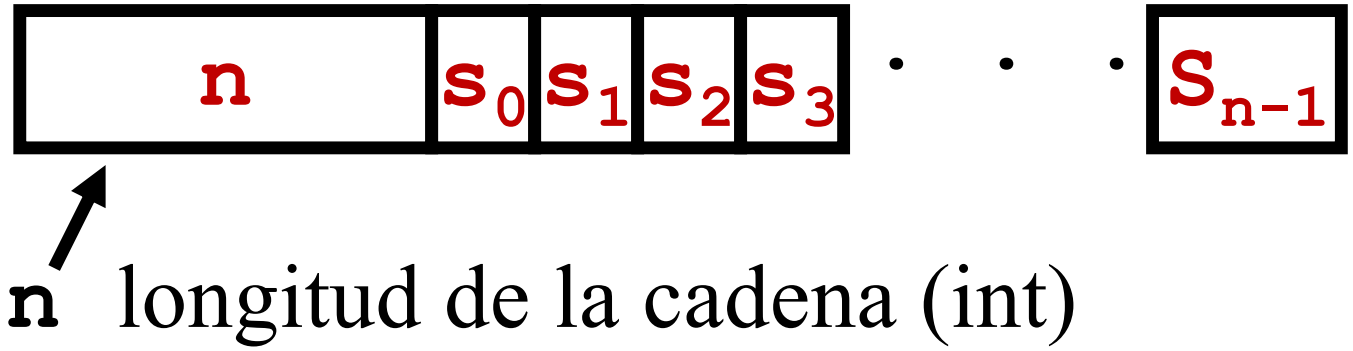
Cadenas de caracteres de longitud variable:

Declaración:
```
string b<>;
```
Equivalente en C:
```
struct {
    u_int b_len;
    srtring *b_val;
} b;
```

Transmisión por la red

```
string s<100>
```

- Se envía: longitud seguido por una secuencia de caracteres ASCII:



Tipos de datos en XDR

Vectores de tamaño fijo:

Declaración: `int a[12];`

Equivalente en C: `int a[12];`

Vectores de tamaño variable:

Declaración: `int d<>;  int d<MAX>`

Equivalente en C:

```
struct {
   u_int d_len;
   int *d_val;
 } d;
```

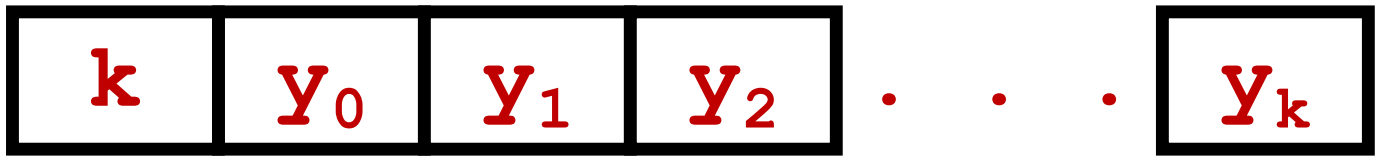
Transmisión por la red

```
int x[n]
```


```
int y<m>
```

k es el tamaño real

$k \leq m$



Tipos de datos en XDR

Cadenas de bytes de longitud fija (secuencias de bytes):

Declaración:
```
opaque a[20];
```
Equivalente en C:
```
char a[20];
```

Cadenas de bytes de longitud variable (secuencias de bytes):

Declaración:
```
opaque b<>;
```
Equivalente en C:
```
struct {
        int a_len;
        char *a_val;
} a;
```

Tipos enumerados

- En XDR:

```
enum color{
  ROJO = 0,
  VERDE = 1,
  AZUL = 2
};
```

- Traducción a C:

```
enum color{
  ROJO = 0,
  VERDE = 1,
  AZUL = 2
};
typedef enum color color;
bool_t xdr_color (XDR *, colortype*);
```

Estructuras

- En XDR:

```
struct punto{
  int x;
  int y;
};
```

- Traducción a C:

```
struct punto{
  int x;
  int y;
};
typdef struct punto punto;
bool_t xdr_punto (XDR *, punto*);
```

Uniones

- En XDR:

```
union resultado switch (int error){
   case 0:
     int n;
   default:
     void;
};
```

- Traducción a C:

```
struct resultado {
   int error;
   union {
      int n;
   } resultado_u;
};
typdef struct resultado resultado;
bool_t xdr_resultado(XDR *, resultado*);
```

Definición de tipos

- En XDR:

```
typedef punto puntos[2];
```

- Traducción a C:

```
typedef punto puntos[2];
```

Listas enlazadas

```
struct lista{
   int x;
   lista *next;
}
```

rpcgen lo reconoce como una lista enlazada

Ejemplo: string.x

string.x

```
program STRING {
        version STRINGVER {
                int vocales(string) = 1;
                char first(string) = 2;
                string convertir(int n) = 3;
        } = 1;
} = 99;
~
```

string_server.c

string_server.c

```
bool_t
vocales_1_svc(char *arg1, int *result,  struct svc_req *rqstp)
{
        bool_t retval;
        int n = 0;
        /* insert server code here */
       int n = 0;
        while (*arg1 != '\0'){
                if (*arg1 == 'a' || *arg1 == 'e' ||
                        *arg1 == 'i' || *arg1 == 'o' || *arg1 == 'u'
)
                          n++;
                arg1++;
        }
        *result = n;
        return retval;
}
```

string_server.c

string_server.c

```
bool_t
first_1_svc(char *arg1, char *result,  struct svc_req *rqstp)
{
        bool_t retval;

        /*
         * insert server code here
         */

        *result = arg1[0];

        return retval;
}
```

string_server.c

string_server.c

```
bool_t
convertir_1_svc(int n, char **result,  struct svc_req *rqstp)
{
        bool_t retval;
        /*
         * insert server code here
         */

        *result = (char *) malloc(256);

        sprintf(*result, "En texto= %d", n);

        return retval;
}
```

string_server.c

string_server.c

```
int
string_1_freeresult (SVCXPRT *transp, xdrproc_t xdr_result,
caddr_t result)
{

        xdr_free (xdr_result, result);

         /*
          * Insert additional freeing code here, if needed
          */

        return 1;
}
```

string_client.c

string_client.c

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

        if (argc < 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        string_1 (host);
        exit (0);
}
```

string_client.c

```
program STRING {
        version STRINGVER {
                int vocales(string) = 1;
                char first(string) = 2;
                string convertir(int n) = 3;
        } = 1;
} = 99;
```

```
void
string_1(char *host)
{
    CLIENT *clnt;

    enum clnt_stat retval_1;
    int result_1;
    char vocales_1_arg1[256];

    enum clnt_stat retval_2;
    char result_2;
    char first_1_arg1[256];

    enum clnt_stat retval_3;
    char *result_3;
    int convertir_1_n;
```

```
enum clnt_stat vocales_1(char *, int *, CLIENT *);

enum clnt_stat first_1(char *, char *, CLIENT *);

enum clnt_stat convertir_1(int , char **, CLIENT *);
```

string_client.c

string_client.c

```
clnt = clnt_create (host, STRING, STRINGVER, "tcp");
if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
}

strcpy(vocales_1_arg1, "Prueba");
retval_1 = vocales_1(vocales_1_arg1, &result_1, clnt);
if (retval_1 != RPC_SUCCESS) {
        clnt_perror (clnt, "call failed");
}
else
        printf("%s  tiene %d vocales \n", vocales_1_arg1,
                                                result_1);
```

string_client.c

string_client.c

```
strcpy(first_1_arg1, "Prueba");
retval_2 = first_1(first_1_arg1, &result_2, clnt);
if (retval_2 != RPC_SUCCESS) {
        clnt_perror (clnt, "call failed");
}
else
        printf("La inicial es %c \n", result_2);
```

string_client.c

string_client.c

```
        convertir_1_n = 124;
        result_3 = (char *) malloc(256);
        retval_3 = convertir_1(convertir_1_n, &result_3, clnt);
        if (retval_3 != RPC_SUCCESS) {
                clnt_perror (clnt, "call failed");
        }
        else
                printf("%d es %s\n",convertir_1_n, result_3);

       free(result_3);


        clnt_destroy (clnt);
}
```

Arrays de longitud variable (vector.x)

C

```
typedef struct {
        u_int t_vector_len;
        int *t_vector_val;
} t_vector;
```

vector.x

```
typedef int t_vector<>;

program VECTOR {
        version VECTORVER {
                int sumar(t_vector v) = 1;
        } = 1;
} = 99;
```

vector_server.c

vector_server.c

```
bool_t
sumar_1_svc(t_vector v, int *result,  struct svc_req *rqstp)
{
        bool_t retval;
        /*
         * insert server code here
         */
        int i;

        *result=0;
        for (i=0; i<v.t_vector_len; i++)
          *result = *result + v.t_vector_val[i];

        return retval;
}
```

vector_client.c

vector_client.c

```
void
vector_1(char *host)
{
        CLIENT *clnt;
        enum clnt_stat retval_1;
        int result_1;
        t_vector  sumar_1_v;
        int i;

        clnt = clnt_create (host, VECTOR, VECTORVER, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
```

vector_client.c

vector_client.c

```
        sumar_1_v.t_vector_len= 5;
        sumar_1_v.t_vector_val= (int *) malloc(5 * sizeof(int));
        for (i =0; i < 5; i ++)
                sumar_1_v.t_vector_val[i] = i;


        retval_1 = sumar_1(sumar_1_v, &result_1, clnt);
        if (retval_1 != RPC_SUCCESS) {
                clnt_perror (clnt, "call failed");
        }
        printf("La suma es %d\n", result_1);

        free (sumar_1_v.t_vector_val);
        clnt_destroy (clnt);
}
```

Lista enlazada (lista.x)

lista.x

```
struct lista {
        int x;
        lista *next;
};

typedef lista *t_lista;

program LISTA {
        version LISTAVER {
                int sumar(t_lista l) = 1;
        } = 1;
} = 99;
```

C

```
struct lista {
        int x;
        struct lista *next;
};
typedef struct lista lista;

typedef lista *t_lista;
```

lista_server.c

lista_server.c

```
bool_t
sumar_1_svc(t_lista l, int *result,  struct svc_req *rqstp)
{
        bool_t retval;
        /*
         * insert server code here
         */

        *result = 0;
        while (l != NULL) {
                *result = *result + l->x;
                l = l->next;
        }
        return retval;
}
```

lista_client.c

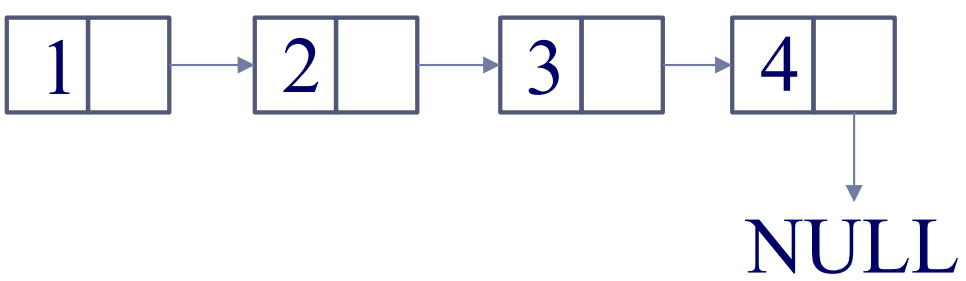
lista_client.c

```
void
lista_1(char *host)
{
        CLIENT *clnt;
        enum clnt_stat retval_1;
        int result_1;
        t_lista  sumar_1_l;

        struct lista n1, n2, n3, n4;

        clnt = clnt_create (host, LISTA, LISTAVER, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }
```

lista_client.c

lista_client.c

```
        sumar_1_l = &n1;
        n1.x = 1;   n1.next = &n2;
        n2.x = 2;   n2.next = &n3;
        n3.x = 3;   n3.next = &n4;
        n4.x = 4;   n4.next = NULL;

        retval_1 = sumar_1(sumar_1_l, &result_1, clnt);
        if (retval_1 != RPC_SUCCESS) {
                clnt_perror (clnt, "call failed");
        }

        printf("La suma es %d\n", result_1);
        clnt_destroy (clnt);
}
```



Gestión de memoria

- Las RPC no reservan memoria dinámica. Se tiene que reservar memoria:
  - El servidor tiene que reservar memoria
  - El cliente tiene que reservar memoria antes de invocar a la RPC

Gestión de memoria

- En general, para cada función RPC descrita en el fichero .x, el stub del servidor generado por rpcgen ya cuenta con una función `xxx_freeresult()', cuya implementación predeterminada hace lo necesario para liberar la memoria asignada para el valor de retorno, típicamente una llamada a xdr_free().
- La implementación de Sun-RPC se encarga de invocar a `xxx_freeresult()' tras dar servicio a la llamada.
- Esto significa que no es necesario liberar memoria en el lado del servidor. Sin embargo, en el cliente, la responsabilidad de liberar la memoria asignada es del usuario a través de una llamada a xdr_free()

Autenticación

- Los mensajes de petición y respuesta disponen de campos para pasar información de autenticación
- El servidor es el encargado de controlar el acceso
- Hay distintos tipos de protocolos de autenticación:
  - Ninguno
  - Al estilo UNIX, basado en *uid* y *gid*
  - Autenticación Kerberos
  - Mediante una clave compartida que se utiliza para firmar los mensajes RPC

Autenticación UNIX

```
CLIENT *cl;
cl = client_create("host", SOMEPROG, SOMEVERS, "udp");

if (cl != NULL) { /* To set UNIX style authentication */
       cl->cl_auth = authunix_create_default(); }
```

Mensajes de petición-respuesta

- Mensaje petición-respuesta