2.3 Llamada a procedimientos remotos (RPC)

(a) Concepto y objetivos

Idea: Ocultar/abstraer los detalles relativos a la comunicación que son comunes a diversas aplicaciones

- Gestión de los diálogos petición-respuesta
- Aplanamiento y formateo de datos (enteros, reales, cadenas, estructuras,...) [*marshaling*, serializacion]
  - Aplanar: organizar datos complejos en un mensaje
  - Desaplanar: extrar datos complejos de un mensaje aplanado
  - Gestionar: representación info. (orden de bytes, tipos complejos, alineado en memoria), diferencias de hardware y S.O.
- Gestión de la interfaz de comunicación (crear y configurar sockets, conectar, escribir, leer, ...)

Aproximación: llamada a procedimientos remotos (RPC:*Remote Procedure Call*)

- Generar automáticamente el código usado en esas tareas comunes
- Ofrecer el entorno y los componentes de apoyo necesarios para dar soporte a esa infraestructura
- Procedimiento llamante y procedimiento llamado se ejecutan en máquinas distintas
- Ofrecer la ilusión de que la llamada remota parezca idéntica a una llamada local (transparencia)

Objetivo: Proporcionar un middelware que simplifique el desarrollo de aplicaciones distribuidas

- Evitar que programador tenga que interactuar directamente con el interfaz de Sockets
  - Abstraer (ocultar) los detalles relativos a la red

- Servidor ofrece procedimientos que el cliente llama como si fueran procedimientos locales
  - Se busca ofrecer un entorno de programación lo más similar posible a un entorno no distribuido
  - El sistema RPC oculta los detalles de implementación de esas llamadas remotas
    - implementa la llamada remota mediante un diálogo petición-respuesta
      - ◇ mensaje de *petición*: { identifica procedimiento llamado; contiene parámetros de la llamada
      - ◇ mensaje de *respuesta*: contiene valor/es devuelto/s
    - se encarga de enviar/recibir mensajes para comunicar ambas partes
    - se encarga de gestionar los contenidos de esos mensajes (empaquetado y formateado de datos)

(b) Funcionamiento general

- **Proceso llamador (cliente)**:
  - Proceso realiza la llamada a una función.
  - Llamada empaqueta id. de función y argumentos en mensaje
  - Envía mensaje a otro proceso.
  - Queda a la espera del resultado.
  - Al recibirlo, lo desempaqueta y retorna el valor
- **Proceso llamado (servidor)**:
  - Recibe mensaje con id. de función y argumentos.
  - Se invoca función en el servidor.
  - Resultado de la función se empaqueta en mensaje
  - Se transmite mensaje de respuesta al cliente.

(c) Ejemplos de entornos RPC

- *Sun-RPC (ONC-RPC: Open Network Computing-RPC)*: RPC muy extendido en entornos Unix, infraestructrua sobre la que se ejecuta NFS (servicio de sistema de ficheros en red), NIS (servicio de directorio)
- *DCE/RPC (Distributed Computing Environmen RPC)*: RPC definido por la Open Software Foundation
- *Java-RMI*: invocación de métodos remotos en Java
- *CORBA (Common Object Requesting Broker Architecture)*: soporta la invocación de métodos remotos bajo un paradigma orientado a objetos en diversos lenguajes
- *SOAP (Simple Object Access Protocol)*: protocolo RPC basado en el intercambio de datos (parámetos+resultados) en formato XML
- *DCOM (Distributed Component Object Model)*: Modelo de Objetos de Componentes Distribuidos de Microsoft, con elementos de DCE/RPC
- *.NET Remoting*: Infraestructura de invocación remota de .NET

(d) Diferencias con llamadas locales (LPC)

- **Punto clave:** manejo de errores
  - Con RPC pueden existir fallos en servidor remoto o en la red
  - Deben detectarse y notificarse al llamador (cliente)
- Acceso a variables globales y efectos laterales en el cliente no son posible
  - Procedim. remoto (servidor) no tiene acceso al espacio de direcciones del cliente $\longrightarrow$ imposibilidad de usar punteros
  - RPC impone un *mayor nivel de encapsulamiento*
- Los parámetros para la llamada remota no pueden pasarse por referencia (sólo por valor)
  - Generalmente se usan mecanismos de copia y restauración para *"simular"* el paso por valor
- Rendimiento de llamadas RPC mucho menor que en llamadas locales
  - Mayor sobrecarga en llamadas RPC (transferencia por red, aplanamiento de datos, etc)
- En alguno entornos se limita el intercambio de estructuras complejas, en otros se usan métodos de aplanado/desaplanado

(d) Aspectos a considerar en entornos RPC

Objetivo: llamada remota lo más similar posible a llamada local

1. Emular la semántica de las llamadas locales:
   - ¿hasta qué punto el **comportamiento** de una llamada remota es **equivalente** al de una llamada local ?
     - Está determinado por el entorno RPC
   - Punto relevante: modo de gestionar los fallos en las llamadas remotas
2. Emular sintaxis de las llamadas locales:
   - ¿hasta qué punto la **forma** en que se realizan llamadas remotas desde el código de los programas clientes es **idéntica** a la de las llamadas locales?
   - Punto relevante: garantizar transparencia

2.3.1 Funcionamiento y principios básicos

(a) Comportamiento ante fallos

Aspecto clave que determina la equivalencia semántica entre llamadas remotas y llamadas locales.

Llamadas locales ofrecen una semántica *"exactamente una vez"* (ejecución fiable)

- El entorno de ejecución de las llamadas locales garantiza que el procedimiento llamado se ejecuta exactamente una vez
- Llamada termina devolviendo un valor de retorno si tuvo éxito o una indicación del error (excepción, código de error) en caso de fallo
- Llamador se queda en espera indefinidamente hasta que finalice llamada
  - En RPC no es posible espera indefinida (posibilidad de fallos)

En llamadas remotas las posibles fuentes de fallos son múltiples.

1. Fallos en los procedimientos llamados
   - La ejecución del proceso llamado se detiene por errores del hardware o del sistema operativo que lo ejecuta (ej.: caida del sistema)
   - También por errores internos del propio procedimiento (divisiones por 0, índices de arrays fuera de rango, etc)
2. Fallos en la comunicación
   - Pérdida de la conexión: la red deja de enviar paquetes (caida de la red, pérdida de un enlace, ...)
   - Corrupción del contenido de alguno de los mensajes enviados
   - Pérdida de paquetes: algún mensaje/s no llega a su destino
   - Recepción fuera de orden: paquetes retrasados recibidos de forma desordenada
3. Otros fallos: bugs en el proceso llamado, ataques, etc

En general el proceso cliente no tiene capacidad para distinguir los diferentes tipos de errores

- Cliente sólo percibe que una o más de sus peticiones no reciben respuesta, pero no llega a saber por qué:
  - la petición(llamada) no llegó al proceso remoto
  - el servidor está caido o no ha llegado a procesar la petición
  - la petición llegó y el servidor la procesó, pero la respuesta no llegó al cliente
  - otros,...
- Dependiendo de los mecanismos definidos y de la semántica se puede volver a pedir la ejecución del procedim. remoto o no

La forma de gestionar los fallos por parte del entorno RPC determina la semántica efectiva de las llamadas remotas.

- Herramientas para controlar los fallos
  - Uso de mensajes de asentimiento (ACKs) con o sin temporizadores (time-out):
    - permiten confirmar la recepción de cada mensaje y detectar la pérdida de mensajes
  - Uso de núm. de secuencia y control de duplicados:
    - cliente y servidor asocian un identificador a sus mensajes
    - el otro extremo mantiene lista con los IDs de los mesajes recibidos
    - se mantiene una copia de los mensajes enviados más recientemente junto con sus IDs

(b) Semántica de las llamadas RPC

Dependiendo del modelo de gestión de fallos por parte del entorno RPC se pueden soportar distintas aproximaciones a la semántica *"exactamente una vez"* de las llamadas locales (LPC)

- No es posible garantizar la semántica *"exactamente una vez"* debido a la posibilidad de fallos de comunicación
- 3 tipos de semántica en llamadas RPC (de menor a mayor complejidad)

semántica *tal-vez* semántica *al-menos-una-vez* semántica *como-máximo-una-vez*

Semántica *tal-vez*:

Procedimiento remoto puede ejecutarse una vez o ninguna

- Cliente puede recibir una respuesta o ninguna
- Funcionamiento:
  - Cliente envía petición y queda a la espera un tiempo
  - Si no llega respuesta dentro del tiempo de espera, continúa su ejecución
  - Cliente no tiene realimentación en caso de fallo (no sabe que pasó)
- Sólo admisible en aplicaciones donde se tolere la pérdida de peticiones y la recepción de respuestas con retaso (fuera de orden)

Semántica *al-menos-una-vez*

Procedimiento remoto se ejecuta una o más veces

- Cliente puede recibir una o más respuestas
- Funcionamiento:
  - Cliente envía petición y queda a la espera un tiempo
  - Si no llega respuesta o ACK dentro del tiempo de espera, repite la petición
  - Servidor no filtra peticiones duplicadas $\Rightarrow$ procedim. remoto puede ejecutarse repetidas veces
  - Cliente puede recibir varias respuestas
- Sólo es aplicable cuando se usan exclusivamente **operaciones idempotentes** (repetibles)
  - Una operación es **idempotente** si se puede ejecutar varias veces resultando el mismo efecto que si se hubiera ejecutado sólo una
    - **Nota**: en ocasiones una operación no idempotente puede implementarse como una secuencia de operaciones idempotentes
- Admisible en aplicaciones donde se tolere que se puedan repetir invocaciones sin afectar a su funcionamiento

Semántica *como-máximo-una-vez*

El procedimiento remoto se ejecuta exactamente una vez o no llega a ejecutarse ninguna

- Cliente recibe una respuesta o una indicación de que no se ha ejecutado el procedim. remoto
- Funcionamiento:
  - Cliente envía petición y queda a la espera un tiempo
  - Si no llega respuesta o ACK dentro del tiempo de espera, repite la petición
  - Servidor filtra las peticiones duplicadas y guarda historial con las respuestas enviadas (servidor con memoria) $\Rightarrow$ procedim. remoto sólo se ejecuta una vez
  - Cliente sólo recibe una respuesta si la petición llegó y se ejecutó el procedim., si no recibe informe del error
- Semántica más próxima a la de llamadas a procedim. locales (semántica sólo una vez) en presencia de fallos

Resumen:

Semántica	Funcionamiento
tal-vez	▪ cliente no retransmite sus peticiones (no usa ACK) ▪ servidor no filtra peticiones duplicadas
al-menos-una	▪ cliente retransmite sus peticiones (usa ACK + temporizador) ▪ servidor no filtra peticiones duplicadas ▪ ante peticiones repetidas, servidor repite ejecución
como-máximo-una	▪ cliente reintenta retransmitir peticiones (usa ACK + temporizador) ▪ servidor filtra peticiones duplicadas ▪ ante peticiones repetidas, servidor retransmite las respuestas pasadas

(c) Componentes e implementación de entornos RPC

Para ofrecer un mecanismo de llamada a procedimientos remotos sintácticamente equivalente al de llamada local el entorno RPC debe proporcionar y dar soporte a una infraestructura que ofrezca **transparencia en la invocación remota**.

- **Objetivo**: es deseable que el programador del sistema distribuido no perciba la diferencia entre llamada local y llamada remota (transparencia)
- Para las tareas adicionales (empaquetado de datos, comunicación) un entorno RPC debe ofrecer
  - Código adicional para soportar la llamada remota a un procedimiento concreto
    - normalmente generado de forma automática
  - Librerías y herramientas complementarias (*runtime*) para soportar la ejecución del entorno RPC
- **Idea clave**: uso de *"representantes"*, tanto del cliente como del servidor
  - Representante del servidor en la máquina cliente (***stub***)
    - realiza el papel de servidor en la máquina cliente
  - Representante del cliente en la máquina servidor (***skeleton***)
    - realiza el papel de cliente en la máquina servidor
  - Proporcionan transparencia en la llamada remota
  - Generados automáticamente en base a la interfaz definida para el procedim. remoto
    - Programador sólo debe programar el código del procedim. remoto (servidor) y el código que hace la llamada remota. (cliente)

Componentes típicos de los entornos RPC

Stubs (representante del servidor → recibe la llamada del cliente)

- Proporciona transparencia en el lado del cliente
- Posee un interfaz idéntico al del procedim. remoto (misma declaración)
  - cada procedimiento remoto que desee llamar el cliente debe tener su propio *stub*
- El cliente realiza una llamada local al procedim. del *stub* como si fuera el servidor real
- Tareas realizadas por el *stub*
  - Localiza al servidor que implemente el procedim. remoto usando un servicio de *binding*
  - Empaqueta los parámetros de entrada (aplanado, *marshalling*) en un formato común para cliente y servidor
  - Envía el mensaje resultante al servidor
  - Espera la recepción del mensaje de respuesta
  - Extrae resultados (desaplanado, *unmarshalling*) y los devuelve al cliente que hizo la llamada

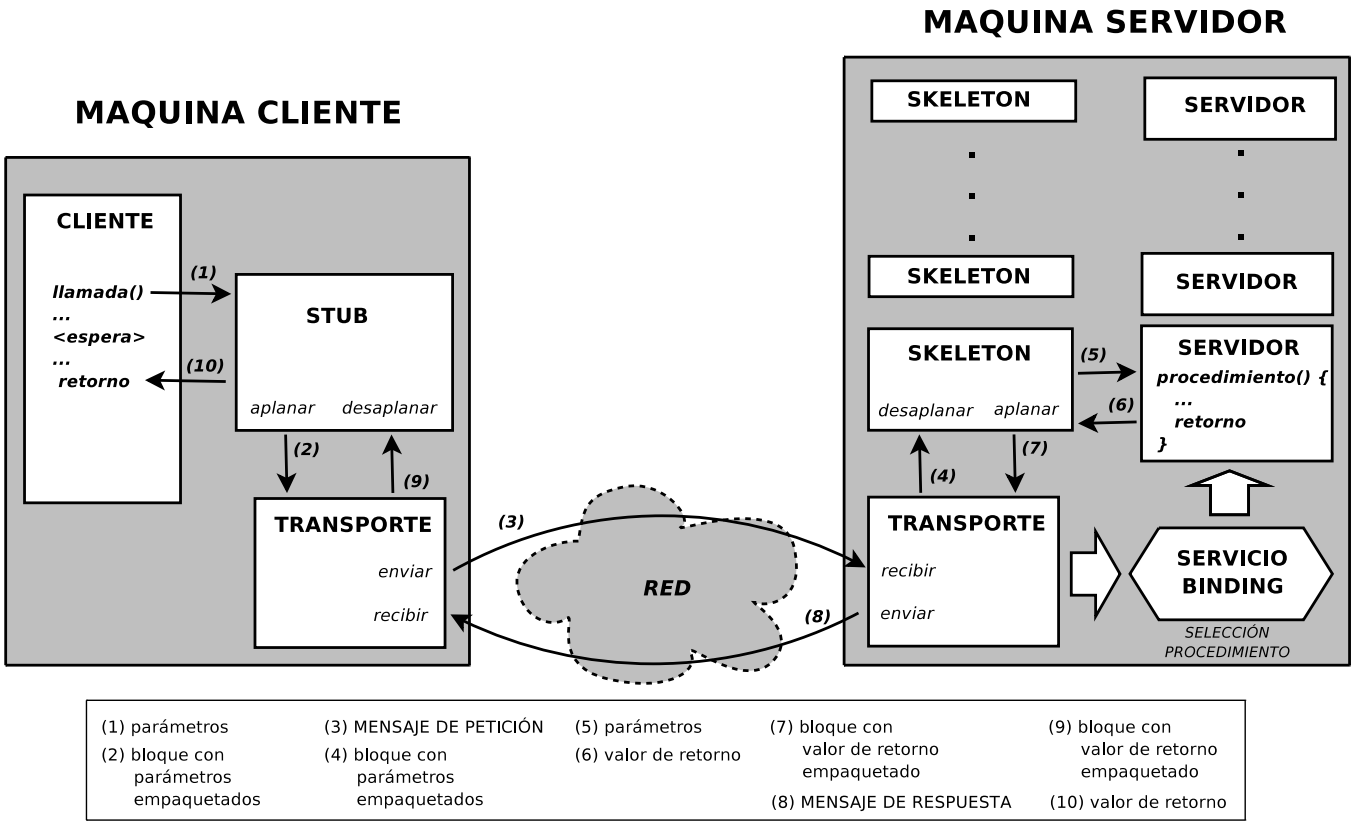
Skeleton (representante del cliente → realiza la llamada al servidor)

- Proporciona transparencia en el lado del servidor
- Conoce el interfaz ofrecido por el procedim. remoto
  - cada procedimiento remoto que ofrece el servidor debe tener su propio *skeleton*
- Realiza llamadas locales al servidor como si fuera el cliente real
  - responsable de la invocación "real" al procedim. remoto
- Tareas realizadas por el *skeleton*
  - Registra el procedimiento en el servicio de *binding*
  - Ejecuta bucle de espera de mensajes
  - Recibe petición
  - Desempaqueta el mensaje (desaplanado, *unmarshalling*)
  - Determina qué método concreto invocar
  - Invoca el procedim. con los argumentos recibidos y recupera el valor devuelto
  - Empaqueta el valor devuelto (aplanado, *marshalling*)
  - Envía mensaje al *stub* del cliente

Otros elementos

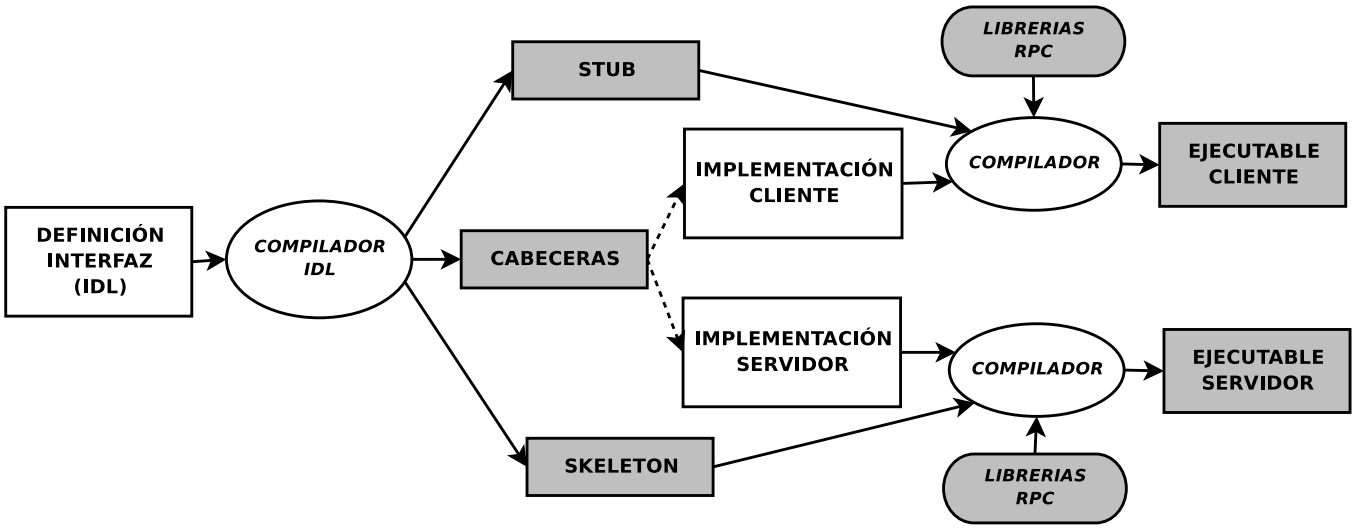
- **Servicio de *binding***
  - Responsable de la transparencia de localización
    - Servicio auxiliar que complementa a *stub* y *skeleton*
  - Gestiona la **asociación** entre el **nombre del procedim. remoto** (y su versión) con su **localización** en la máquina servidor (dirección, puertos, *skeleton*, etc)
  - Realiza la búsqueda del *skeleton* de la implementación concreta del procedim. remoto llamado por un cliente
  - Selecciona *skeleton*+servidor que atenderá la llamada remota
  - Ejemplos: `portmapper` en Sun-RPC, protocolo UDDI en servicios web, `rmiregistry` en Java-RMI
- **Compilador de interfaces**
  - A partir de la descripción del interfaz del procedim. remoto genera de forma automática el código del *stub* y del *skeleton*
    - *stub*+*skeleton* **sólo dependen** de la interfaz del procedim. remoto
  - Dependiendo del entorno RPC puede generar otro código adicional que sea necesario
  - Interfaz del procedim. remoto (datos necesarios para generar *stub*+*skeleton*)
    - especifica el interfaz ofrecido por el procedim (args. de entrada + valor devuelto)
    - especifica cómo será el aplanado/desaplanado
    - opcionalmente aporta info. que se usará para localizar el procedim. remoto (núm. versión)
  - **Definición de interfaces de procedim. remotos:**
    1. Usando el mismo lenguaje de programación con el que será implementado
       - Ejemplo: Java-RMI (usa directamente interfaces Java)
    2. Usando un lenguaje de definicion de interfaces independiente del lenguaje de programación **[IDL: *interface definition language*]**
       - Compilador IDL hace la traducción al lenguaje de implementeción
       - Ejemplos: XDR usado en Sun-RPC, Corba-IDL, WSDL en servicios web

Componentes típicos de los entornos RPC



(1) parámetros	(3) MENSAJE DE PETICIÓN	(5) parámetros	(7) bloque con valor de retorno empaquetado	(9) bloque con valor de retorno empaquetado
(2) bloque con parámetros empaquetados	(4) bloque con parámetros empaquetados	(6) valor de retorno	(8) MENSAJE DE RESPUESTA	(10) valor de retorno

Funcionamiento del compilador de interfaces

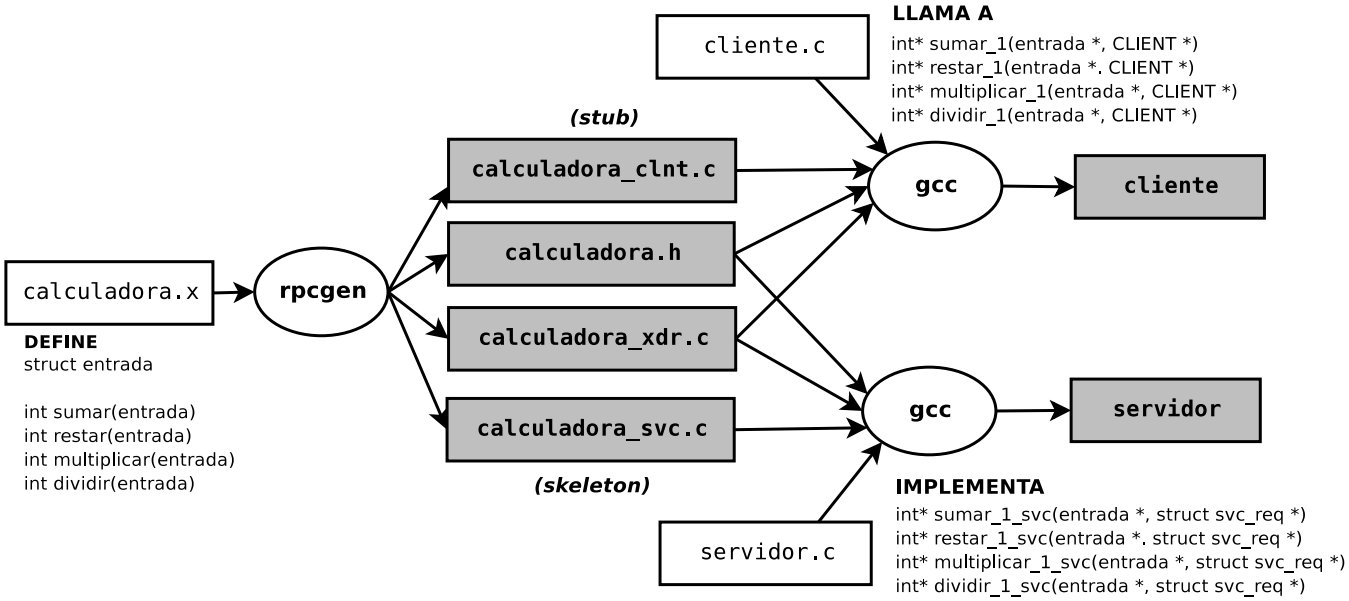


2.3.2 Sun-RPC

- Definido originalmente por Sun Microsystems dentro del desarrollo del sistema de ficheros distribuidos NFS (Network File System)
  - Disponible en multitud de plataformas
  - También denominado ONC-RPC (*Open Network Computing RPC*)
  - Definido en los RFCs 1831 y 1832
    - RFC 1831: especificación mecanismo RPC (`www.ietf.org/rfc/rfc1831.txt`)
    - RFC 1832: especificación lenguaje XDR (`www.ietf.org/rfc/rfc1832.txt`)
- Sigue el esquema general de llamada remota con pequeñas modificaciones/limitaciones
- Sun RPC emplea XDR (*eXternal Data Representation*)
  - XDR especifica cómo aplanar/desaplanar los datos intercambiados entre *stub* y *skeleton* en un formato independiente de la arquitectura
  - Usado para definir el tipo y la estructura de los argumentos y valores de retorno de los procedimientos remotos
- Emplea como lenguaje IDL (*interface definition language*) una ampliación de XDR que permite la definición de procedimientos
  - Permite identificar procedimientos y versiones con un núm.
  - Especifica argumentos de entrada + valor de retorno (no la implementación)
- Incluye un compilador para XDR (**rpcgen**)
  - Genera código para *stub* y *skeleton* a partir de la especificación de los procedimientos remotos
  - Genera código para aplanar y desaplanar datos en formato XDR

- **portmapper**: proceso responsable de la localización de procedimientos remotos
  - Responsable de las tareas de registro y *binding*
  - Los servidores registran con el *portmapper* los procedim. remotos que exportan
  - El *portmapper* queda a la escucha (puerto 111) y redirecciona las peticiones de accesos a procedim. hacia sus respectivos puertos locales de escucha

Esquema de generación de stub y skeleton a partir de la definición XDR del interfaz remoto



Funcionamiento Sun-RPC

(a) Definición de interfaces

Especificación del interfaz de los procedim. remotos en notación XDR

- Por convención tienen la extensión .x (`calculadora.x`)

Contenido:

1. Definición XDR de los tipos de datos usados
   - Sun RPC puede usar como argumentos y resultados estructuras arbitrarias de datos
     - XDR especifica cómo se aplanan/desaplanan para transmitirlos por la red
   - Sintaxis similar a C, pero con diferencias
   - XDR ofrece notación para definir:
     - constantes (`const`)
     - definición de tipos (`typedef`)
     - vectores de tamaño constante o variable { `int datos[TAMANO]` / `int datos<>` }
     - estructuras y uniones (`struct` y `union`)
2. Definición de la interfaz de los procedimientos
   - Los procedim. remotos se agrupan en "*programas*" y "*versiones*"
     - Cada *programa* tiene un nombre y un identificador (nº entero)

```
0x00000000 - 0x1FFFFFFF: definidos por Sun (portmapper, nfs, nis, ...)
0x20000000 - 0x3FFFFFFF: definidos por el usuario
0x40000000 - 0x5FFFFFFF: reservado
0x60000000 - 0xFFFFFFFF: reservado
```

     - Cada *programa* tiene al menos una *versión* con su propio nombre e identificador
     - Para cada procedim. remoto se define su prototipo y se asigna un identificador numérico
   - **Definición de procedimientos remotos**
     - Internamente cada procedim. remoto es identificado de forma única por el triple *(nº de programa, nº de versión, nº de procedim)*

- **Importante:** Sun-RPC sólo permite procedimientos con un único parámetro.
  - Para pasar 2 o más parámetros es necesario usar estructuras para agruparlos

    Nota: En versiones posteriores de Sun-RPC se ha incluido soporte para 2 o más parámetros, aunque por compatibilidad aunque no suele usarse esa posibilidad
- Los parámetros de salida se devuelven mediante un único resultado
- Los parámetros y valores que no sean de tipos básicos deben haber sido definidos usando la sintaxis XDR

Ejemplo: `calculadora.x`

```
struct entrada {
   int arg1;
   int arg2;
};

program CALCULADORA {
     version CALCULADORA_VER {
             int sumar(entrada) = 1;
             int restar(entrada) = 2;
             int multiplicar(entrada) = 3;
             int dividir(entrada) = 4;
     } = 1;
} = 0x30000001;
```

(b) Compilación de interfaces

rpcgen es el compilador de interfaces (compilador IDL)

- Transforma la especificación XDR en un conjunto de ficheros C

Ficheros generados (`rpcgen fichero.x`)

- Código del *stub* del cliente (`fichero_clnt.c`)
  - Funciones C a llamar desde las aplicaciones clientes para realizar la llamada remota
- Código del *skeleton* de la parte servidor (`fichero_svc.c`)
  - Incluye un procedimiento `main()` desde el que se realiza el registro del procedimiento remoto en el **port mapper**
  - Realiza las llamadas *"reales"* a las implementaciones de los métodos remotos
- Código para el aplanamiento/desaplanamiento (*marshalling*) de los tipos XDR definidos (*fichero_xdr.c*)
  - Sólo se genera si se han definido tipos de datos complejos
- Archivo de cabecera con los tipos y declaración de prototipos (`fichero.h`)
  - Deberá ser incluido en los ficheros que implementen el código de los clientes y del servidor.
  - Contiene
    - definiciones C de los tipos XDR declarados en el interfaz `fichero.x`
    - prototipo de las funciones que deberán ser llamadas desde los clientes
    - prototipo de las funciones que deberán ser implementadas por el servidor

Opciones interesantes

- `rpcgen -N`: usa la nueva versión de Sun RPC que permite procedimientos con dos o más parámetros
- `rpcgen -Ss`: genera una implementación vacía de los procedimientos remotos (útil como punto de partida para escribir el servidor)
- `rpcgen -Sc`: genera código para un cliente por defecto (útil como punto de partida para escribir los clientes)

Ejemplo

```
$ rpcgen calculadora.x

calculadora_clnt.c : código del stub del cliente
calculadora_svc.c  : código el skeleton del servidor
calculadora_xdr.c  : código para aplanar/desaplanar datos XDR (se usan tipos complejos)
calculadora.h      : fichero de cabecera
```

(c) Implementación de los clientes

El programa del cliente se implementa desde cero o a partir del modelo ofrecido por `rpcgen -Sc`

- Debe incluir el fichero de cabecera `.h` generado por `rpcgen`

Para importar interfaz remota cliente debe conocer: { nombre/dir. del servidor; nº de programa; nº de versión }

Cliente no establece conexión directa con [*skeleton* + procedim. remoto]
$\longrightarrow$ se hace mediante el **port mapper**

- *port mapper* se encarga de ofrecer *transparencia de localización*
- Asocia un puerto local al conjunto de procedimientos remotos que forman parte de un programa+versión concreto

Funcionamiento:

1. Establecimiento de conexión

   CLIENT * clnt_create(char * host, long prog, long vers, char * proto)

   - Recibe { nombre de la máquina remota; nº de programa + nº de versión; protocolo (TCP, UDP o ambos) }
   - Contacta con `port mapper` remoto y devuelve handle del *stub* del cliente

2. Llamada al procedimiento remoto

   La llamada a los procedimientos remotos está soportada por los procedimientos *stub* y de aplanamiento.

   - Procedimientos remotos son llamados de forma indirecta, a través de los *stubs*
     - nombre función *stub*: [NOMBRE PROCEDIM]_[VERSION] (...)
     - recibe 2 argumentos: { puntero al parámetro de entrada; handle del cliente (CLIENT *) }
     - devuelve 1 resultado: puntero al valor devuelto

Nota: Si se usa la nueva modalidad de Sun-RPC (`rpcgen -N`) el núm de parámetros del *stub* es variable:

- El nombre de la función $stub$ sigue el [nombre]_[version]) y recibe como último parametro el handler
- Los parámetros de entrada podrán ser más de uno y se pasan directamente, no como punteros
- El valor devuelto también se recibe directamente, no como un puntero

3. Liberación del handle: `clnt_destroy(CLIENT * clnt)`

Ejemplo: esquema general del Cliente

- Incluir declaraciones de constantes y tipos

```
#include "calculadora.h"
```

- Crear referencia al *stub*

```
CLIENT *clnt;
clnt = clnt_create (host, CALCULADORA, CALCULADORA_VER, "tcp");
if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
}
```

- Establecer parámetros y llamar a "`procedimiento_version`"

```
entrada param;
param.arg1 = 10;
param.arg2 = 10;

resultado = sumar_1(&param, clnt);
if (resultado == (int *) NULL) {
       clnt_perror (clnt, "fallo en llamada a sumar_1");
}
```

- Liberar referencia al *stub*

```
clnt_destroy(clnt);
```

(d) Implementación de los procedimientos remotos

La implementación de los procedimientos remotos se puede crear desde cero o a partir del modelo ofrecido por `rpcgen -Ss`

- Debe incluir el fichero de cabecera `.h` generado por **rpcgen**

La función a implementar será llamada por el *skeleton* generado por `rpcgen` cuando se invoque el procedimiento remoto

- Prototipo de la función a implementar
  - Nombre: [NOMBRE PROCEDIM]_[VERSION]_svc (...)
  - Recibe 2 argumentos de entrada
    - Puntero al *argumento de entrada*, que será del tipo especificado en la definición XDR
    - Puntero a una estructura `struct svc_req` con información sobre la petición recibida
  - La función tiene 1 *valor de retorno*: puntero al tipo de dato a devolver
    - La variable sobre la que se escribe el valor a devolver debe declararse como `static` para que su dirección de memoria no sea sobrescrita al salir de la función

Ejemplo:

```
#include "calculadora.h"
...
int * sumar_1_svc(entrada *argp, struct svc_req *rqstp){
        static int  result;
        result = entrada->arg1 + entrada->arg2;
        return &result;
}
```

(e) Funcionamiento de los *stubs* y *skeletons*

Contenidos del *stub* (`calculadora_clnt.c`)

- Generado automáticamente por `rpcgen`
- Ofrecen a los clientes una función "[NOMBRE PROCEDIM]_[VERSION]" como interfaz de acceso para cada uno de los procedimientos remotos
- Cada función *stub* realiza la llamada remota mediante la llamada a la función **clnt_call()**
  - Se encarga de llamar a las rutinas de aplanado/desaplanado que correspondan y de gestionar el envío/recepción de mensajes
  - Función **clnt_call(clnt, procnum, inproc, in, outproc, out, tout)** Argumentos:
    - `CLIENT *clnt`: handle del cliente
    - `long procnum`: identificador del procedim., definido en { fichero XDR / cabecera `.h` }
    - `xdrproc_t inproc`: puntero a función XDR para *aplanar* param. entrada
    - `char * in`: puntero al parámetro de entrada
    - `xdrproc_t outproc`: puntero a función XDR para *desaplanar* valor retorno
    - `char * out`: puntero al resultado
    - `struct timeval tout`: tiempo total en segundos que puede durar la llamada remota
- Implementa semántica AL-MENOS-UNA-VEZ
  - Temporización entre reintentos { establecida al crear el handle / modificado con `clnt_control()` }
  - nº reintentos = tiempo total llamada / temporización reintentos
  - Si hay respuesta, `resultado = 0`
  - Si, tras n reintentos, no hay respuesta, `resultado = error`
- Con UDP, la longitud de los mensajes está limitada a 8 Kbyte
- **Ejemplo**: *stub* para el procedimiento `int sumar(entrada) = 1;`

```
int * sumar_1(entrada *argp, CLIENT *clnt) {
   static int clnt_res;
   memset((char *)&clnt_res, 0, sizeof(clnt_res));
   if (clnt_call(clnt, sumar, (xdrproc_t) xdr_entrada, (caddr_t) argp,
                 (xdrproc_t) xdr_int, (caddr_t) &clnt_res, TIMEOUT) != RPC_SUCCESS)
      return (NULL);
   }
   return (&clnt_res);
}
```

Contenidos del *skeleton* (`calculadora_svc.c`)

- Generado automáticamente por `rpcgen`
- Tiene dos componentes
  - Programa principal (`main()`)
    1. Crea socket de escucha en un puerto al azar $\left\{ \begin{array}{l} \textbf{svctcp\_create()} \\ \textbf{svcudp\_create()} \end{array} \right.$
    2. Registra el interfaz remoto en el `port mapper` local (**svc_register()**)
       - Asocia el par (*nº de programa, nº de versión*) con *nº de puerto escucha*

       ```
       svc_register(transp,CALCULADORA,CALCULADORA_VER,calculadora_1,IPPROTO_TCP)
       ```

    3. Queda a la espera de peticiones (**svc_run ()**)
  - Procedimiento *"repartidor"*

    [nombre programa]_[version](struct svc_req *rqstp, register SVCXPRT *transp)
    1. Recupera el mensaje de petición recibido de los *stubs*
    2. Extrae el *nº de procedimiento* invocado
       Se usa para seleccionar las funciones de aplanado/desaplanado y el procedimiento concreto a invocar
    3. Desaplana los argumentos recibidos
    4. Llama a la implementación del procedimiento ([NOMBRE]_[VERSION]_[svc] ())
       $\longrightarrow$ [proporcionado por usuario]
    5. Recibe el valor de retorno y lo aplana
    6. Envia el mensaje de respuesta al *stub* del cliente