



# **Desarrollo de Sistemas Distribuidos**

## **Práctica 2**

### **Programación RPC**

Dpto. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

**Curso 2017-18**

# Índice

- Presentación de objetivos
- Comunicaciones entre procesos en sistemas distribuidos
  - Introducción
  - Sockets y RPC
- RPC
  - Objetivos
  - Modelo de llamadas remotas a procedimientos
  - Creación de Aplicaciones RPC
  - Compilación de Programas RPC
  - Ejecución de Programas RPC
- Ejercicio Propuesto

# Objetivos

- Objetivo **general**:
  - En esta práctica se pretende que el alumno conozca y adquiera experiencia en el manejo de los mecanismos para la implementación de programas distribuidos con llamadas a procedimientos remotos RPC (**R**emote **P**rocedure **C**all)
- Objetivos **específicos**:
  - Conocer los mecanismos para definir interfaces a procedimientos remotos
  - Aprender a implementar interfaces remotas y hacerlas accesibles
  - Implementar programas cliente y servidor basados en esta tecnología
  - Compilar, desplegar y ejecutar los programas desarrollados
  - Resolver problemas de comunicación entre aplicaciones de usuario

# Objetivos

Enlaces a información complementaria

- [RFC 1831: RPC Specification](https://www.ietf.org/rfc/rfc1831.txt)
  - <https://www.ietf.org/rfc/rfc1831.txt?number=1831>

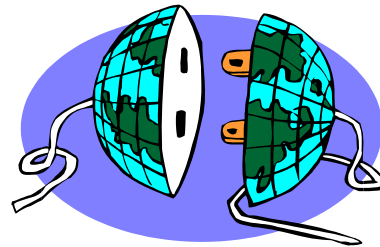
# Introducción

- En sistemas distribuidos, programas que se ejecutan en espacios de direcciones **diferentes**, y posiblemente, equipos **diferentes**, pueden necesitar **comunicarse** unos con otros



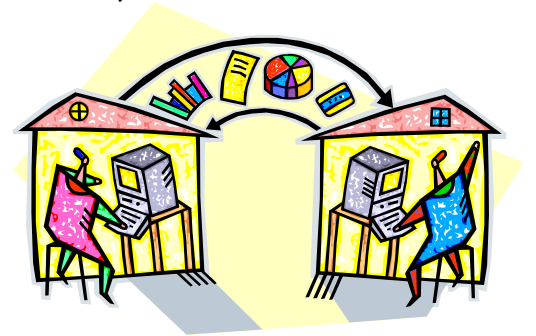
# Sockets y RPC

- Un mecanismo básico de comunicación es el basado en **sockets** (enchufes). Esta tecnología requiere el tratamiento de protocolos de **bajo nivel** para la codificación y decodificación de los mensajes intercambiados, lo que dificulta la programación y es más proclive a errores.



# Sockets y RPC

- Una alternativa a los sockets es RPC (*Remote Procedure Call*), que permite abstraer e implementar las comunicaciones, como llamadas a procedimientos convencionales (locales), pero posiblemente ejecutados en máquinas remotas. La codificación, empaquetado y transmisión de argumentos (proceso conocido como *serialización* o *marshalling*) se realiza mediante algún protocolo estándar de formateo de datos, como XDR (*eXternal Data Representation*).



- RPC resulta adecuado para programas basados en llamadas a procedimientos.

# ¿Por qué surgió RPC?

- Protocolos orientados a funciones

- Telnet, FTP

- No pueden ejecutar funciones con parámetros  
“ejecuta la función Y con los parámetros X1, X2 en la máquina Z”

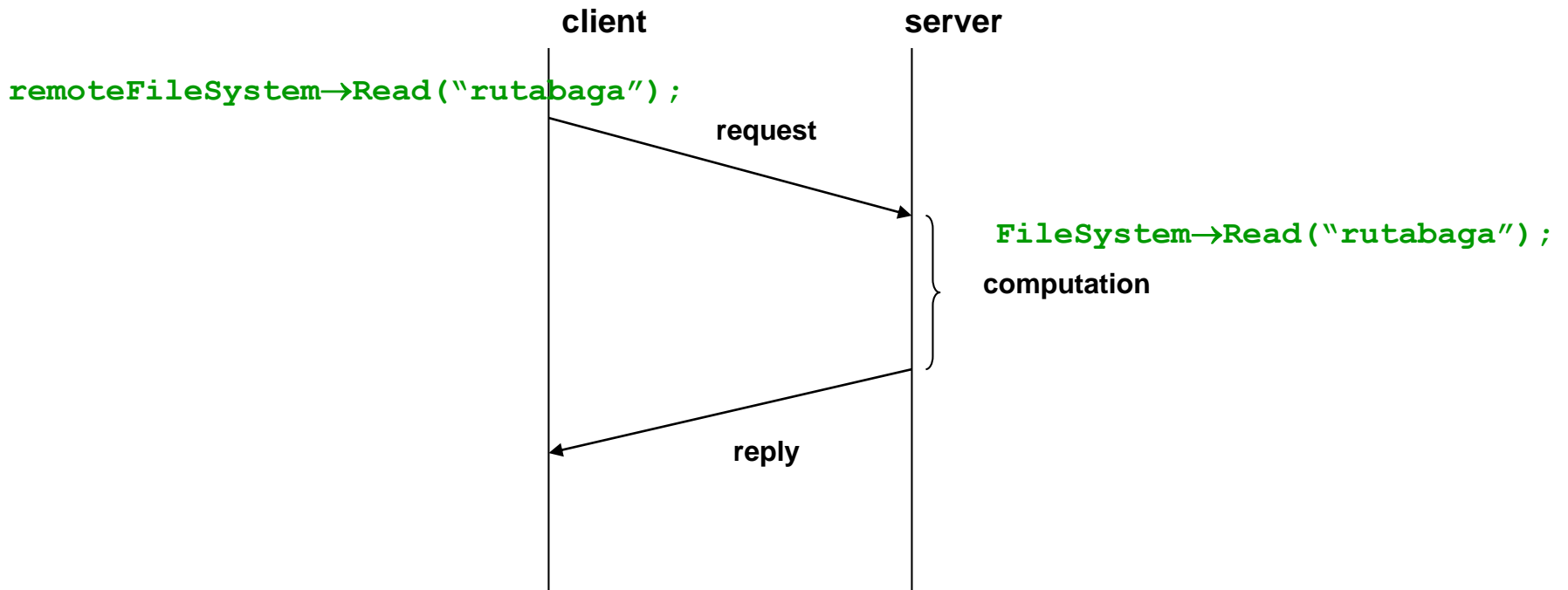


**RPC hace esto  
de forma  
transparente  
al programador**

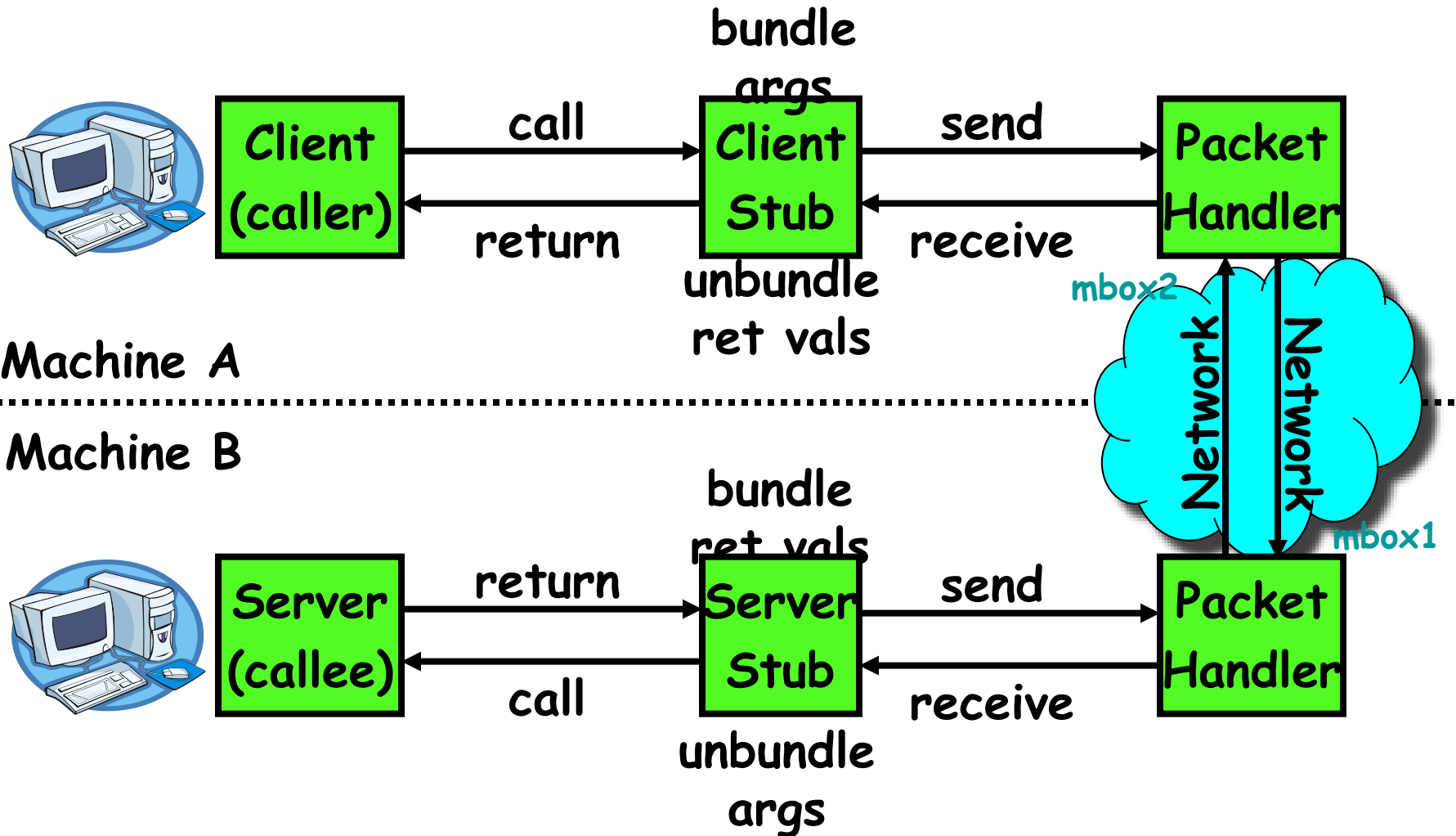
- » Necesitamos construir la interfaz del programa
      - » Construir el entorno en tiempo real – formato de los comandos de salida, interfaz con red de comunicación, serializar la respuesta entrante



# RPC – funcionamiento



# RPC Funcionamiento



# RPC Stubs

- *Stub cliente*

- Tiene la misma interfaz que una función local
- Agrupa los argumentos en un mensaje y lo envia al stub del servidor
- Espera respuesta, desagrupa resultados
- Devuelve resultados al cliente

- *Stub servidor*

- Simula una llamada local
- Escucha en un socket al stub del cliente
- Desagrupa los argumentos en variables locales
- Hace la llamada local
- Agrupa el resultado en un mensaje para el cliente

# RPC Sun

## 1. Identificación unívoca de procedimiento

```
/* Archivo msg.x: Protocolo de impresion de un mensaje remoto */  
program MESSAGEPROG {  
    version PRINTMESSAGEEVERS {  
        int PRINTMESSAGE (string) = 1;  
    } = 1;  
} = 0x20000001;
```

3 Número de procedimiento

2 Número de versión

1 Número de programa

**/etc/rpc** guarda el número de programa con su nombre

# RPC Sun

- Números de programas

0 - 1ffffff	Definidos por Sun
20000000 - 3ffffff	Definidos por los usuarios para programas particulares
40000000 - 5ffffff	Reservados para programas que generan números de programas dinámicamente
60000000 - fffffff	Reservados para uso futuro

# RPC Sun

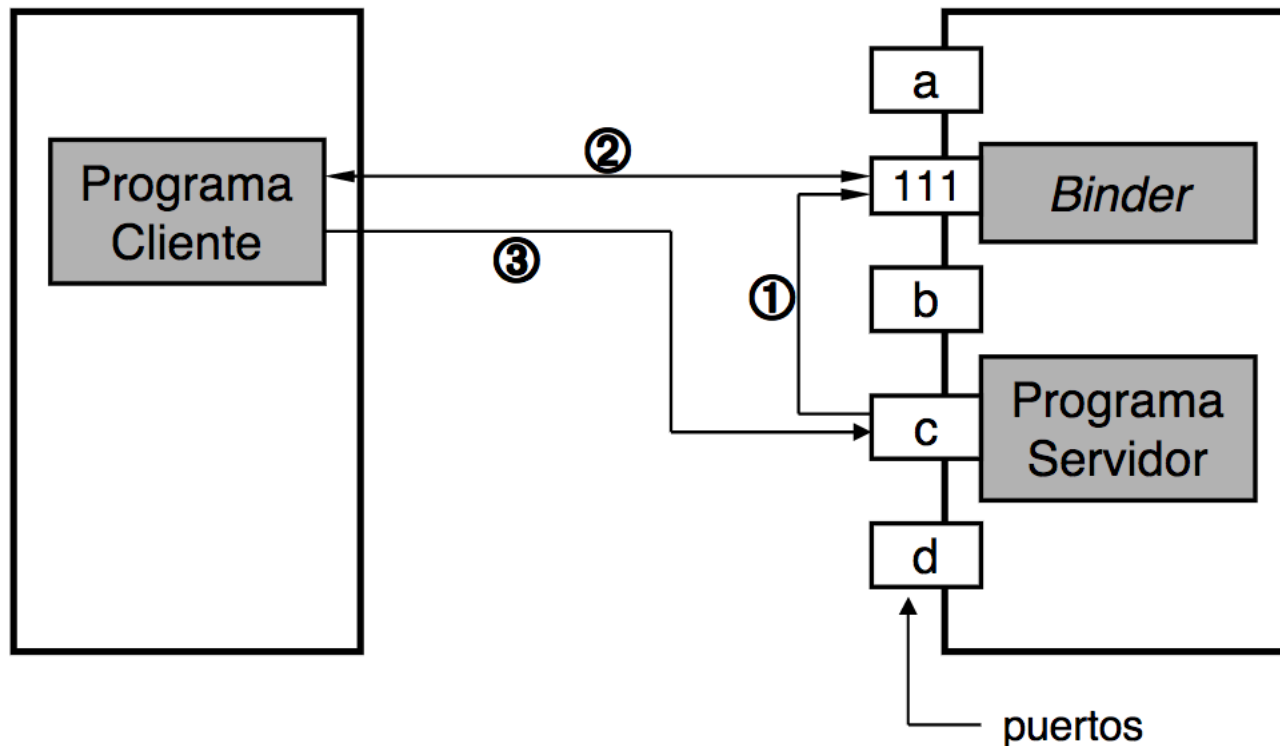
## 2. Selección de red

- **netpath.** Variable que especifica el orden para intentar los transportes (eg tcp:udp)
- **visible.** Utiliza fichero /etc/netconfig, los que tienen el flag v activado en orden de aparición.
- **tcp**
- **udp**

```
clnt = clnt_create(server, DIRPROG, DIRVER, "netpath");
```

# RPC Sun

## 3. Directorio de servicios *rpcbin*. Enlace dinámico (*binding*)



# RPC Sun

## 3. Directorio de servicios *rpcbin*. Enlace dinámico (*binding*)

- *Asocia servicios con direcciones.*
  - *Rpcbin está en el puerto 111*
  - *Un servicio se registra*
- *Borrar registros*
- *Obtener dirección de un programa concreto*
- *Listar direcciones de programas (*rpcinfo -p*)*
- *Realizar una llamada remota a un cliente*
- *Devolver la hora*



# RPC Sun

## 4. Varios niveles

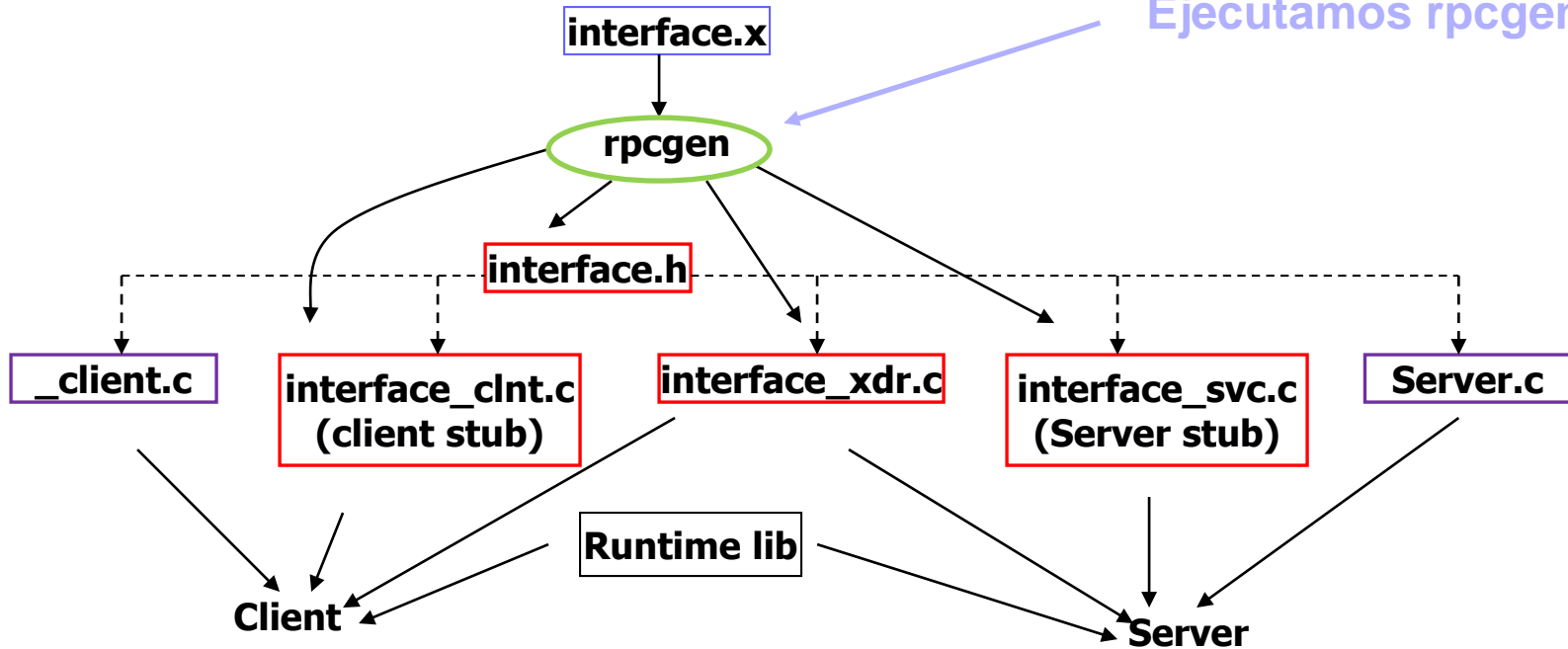
- Permite interactuar a varios niveles (ej. Nivel bajo – encriptación). Nosotros no lo vamos a usar, ya que usamos solo alto nivel con *rpcgen*

## 5. *XDR, representación externa de datos, que permite interoperar con diferentes arquitecturas*

# rpcgen

Definimos una interfaz, datos y rutinas accesibles remotamente

Ejecutamos rpcgen



**Source: R. Stevens, *Unix Network Programming (IPC) Vol 2, 1998***

# rpcgen

Rpcgen genera automaticamente:

- Cabecera con definiciones comunes al cliente y servidor \*.h
- Rutinas xdr para los tipos de datos definidos \*\_xdr.c. Convierten datos al formato XDR y viceversa.
- Los stub del cliente y el servidor \*\_svc.c y \*\_clnt.c
- Estructuras de los programas cliente y servidor \*\_client.c y \*\_server.c

# Ejemplo

## 1. Crear tipos datos y definir funciones remotas:

```
/* dir.x : Protocolo de listado de directorio remoto */
const MAX= 255; /* longitud maxima de la entrada directorio */
typedef string nametype<MAX>; /* entrada directorio */
typedef struct namenode *namelist; /* enlace en el listado */

struct namenode{
    nametype name; /* nombre de la entrada de directorio */
    namelist next ; /* siguiente entrada */
};

/* La siguiente union se utiliza para discriminar entre llamadas
 * con exito y llamadas con errores */
union readdir_res switch (int errno) {
    case 0:
        namelist list; /* sin error: listado del directorio */
    default:
        void; /* con error: nada */
};

program DIRPROG {
    version DIRVER {
        readdir_res READDIR(nametype) = 1;
    } =1;
} = 0x20000155;
```

**String = char \* en C**

**Mayúsculas**

# Ejemplo

## 2. Generar los stub, plantillas, etc.

### ■ Rpcgen -Nca dir.x

- (-N). Código estilo C = genera el código con los argumentos pasados por valor (sin struct si hay varios)

- Archivos plantilla

Opción	Función
-a	Genera todos los archivos plantilla
-Sc	Genera la plantilla para el cliente
-Ss	Genera la plantilla para el servidor
-Sm	Genera la plantilla del archivo para la utilidad make

- (-C). Código ANSI-C o C++. Se usa junto con -N.
- (-M). Multithread. Código Multithread seguro. Para entornos multihebras.

servidor

# Ejemplo

`/* msg_proc.c: implementacion del procedimiento remoto */`

`#include <stdio.h>`

`#include "msg.h"`

`/* el archivo lo genera rpcgen */`

`int *`

`printmessage_1(msg, req)`

`char **msg;`

`struct svc_req *req; /* detalles de la llamada */`

`{`  
`static int result; /* es obligatorio que sea estatica */`

`FILE *f;`

`f = fopen("/dev/console", "w");`

`if (f == (FILE *)NULL) {`

`result = 0;`

`return (&result);`

`}`

`fprintf(f,"%s\n", *msg);`

`fclose(f);`

`result = 1;`

`return (&result);`

`}`

Añade (\_1) número versión a la función

Puntero a un array de caracteres  
(argumentos)

Rpcgen (SIN la opción -N).

Información  
contexto;  
programa, versión,  
puntero a  
información de  
transporte

Estática. Para poder  
devolver un puntero y  
que tenga sentido en la  
máquina remota

Puntero a un entero

```
dirprog_1(char *server, nametype dir)
```

```
{
```

```
CLIENT *clnt;
```

```
readdir_res *result;
```

```
namelist nl;
```

```
#ifndef DEBUG
```

```
clnt = clnt_create(server, DIRPROG, DIRVER, "netpath");
```

```
if (clnt == (CLIENT *) NULL) {
```

```
    clnt_pcreateerror(server);
```

```
    exit(1);
```

```
}
```

```
#endif /* DEBUG */
```

```
result = readdir_1(dir, clnt);
```

```
if (result == (readdir_res *) NULL) {
```

```
    clnt_perror(clnt, server);
```

```
    exit(1);
```

```
}
```

```
if (result->errno != 0) {
```

```
    errno = result->errno;
```

```
    perror(dir);
```

```
    exit(1);
```

```
}
```

```
for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->next)
```

```
    printf("%s\n", nl->name);
```

```
xdr_free (xdr_readdir_res, result);
```

cliente

# Ejemplo

↑  
Cualquier transporte (Tcp,  
udp) con el flag v en  
/etc/netconfig

Crea una estructura de  
datos (handle) del  
cliente, que es la que  
pasa a la rutina del  
stub para llamar al  
procedimiento remoto  
*Clnt\_destroy liberará  
memoria cuando ya no  
se realicen más  
llamadas*

Libera memoria  
asignada a la llamada  
RPC, similar a *free()*

# Ejemplo

- `_xdr.c` Hace la serialización con XDR.
  - En este ejemplo muestra una función `xdr_pointer()` que sigue una cadena de punteros y codifica el resultado en una cadena de bytes. Solucionando el problema de pasar punteros del espacio de direcciones de una máquina a otra (en el cual no serían válidos).



# Ejemplo

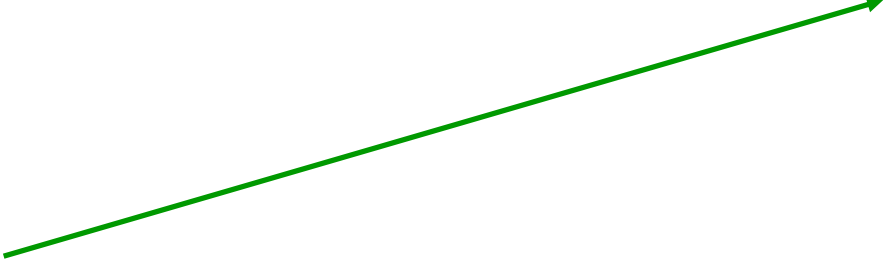
## 3. Compilación y enlazado

- `Make -f makefile.dir`  `rpcgen` genera un makefile

- `gcc dir_client.c dir_clnt.c dir_xdr.c -o cliente -lnsl`

- `gcc dir_server.c dir_svc.c dir_xdr.c -o servidor -lnsl`

`Siempre se ejecuta en  
segundo plano, sin  
necesidad de invocarlo  
con &`



`libnsl contiene las funciones de red`



# Ejemplo

## 4. Ejecutar servidor en una máquina

- `./servidor`

## 5. Ejecutar cliente en otra (o en otro shell si es en local)

- `./cliente <nombre_servidor> <parametros>`

# Lenguaje RPC

No es C  
Aunque se  
parece mucho

- Es una extensión del lenguaje XDR
- Definiciones
  - No asignan espacio
  - Las variables necesitan ser declaradas posteriormente

No son como las  
declaraciones



```
<lista-definiciones> ::= <definicion>; | <lista-definiciones>;  
<definicion> ::= <definicion-enum> |  
                  <definicion-const> |  
                  <definicion-typedef> |  
                  <definicion-struct> |  
                  <definicion-union> |  
                  <definicion-program>
```

# Lenguaje RPC

- Enumeraciones. Misma sintaxis que C.

RPC

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

--->

C

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

# Lenguaje RPC

- Constantes

RPC

```
const DOCENA = 12;
```

--->

C

```
#define DOCENA 12
```

- Definiciones de tipo

```
typedef string tipo_nombref<255>;
```

--->

```
typedef char *tipo_nombref;
```

# Lenguaje RPC

RPC no soporta  
declaraciones  
de variables

- Declaraciones de tipos
  - Deben formar parte de un *struct* o *typedef*

RPC

C

```
tipocolor color;
```

--->

```
tipocolor color;
```

simple

```
tipocolor paleta[8];
```

--->

```
tipocolor paleta[8];
```

Array fijo

```
int altura<12>;
```

--->

```
struct {  
    u_int altura_len;  
    int *altura;  
} altura;
```

Array variable

```
listaelementos *siguiente;
```

--->

```
listaelementos *siguiente;
```

punteros

# Lenguaje RPC

- Estructuras

RPC

```
struct coord {  
    int x;      --->
```

C

```
struct coord  
    int x;
```

- Uniones

```
union resultado_leido switch (int errno) {  
    case 0:  
        tipodato dato;  
    default:  
        void;  
};
```

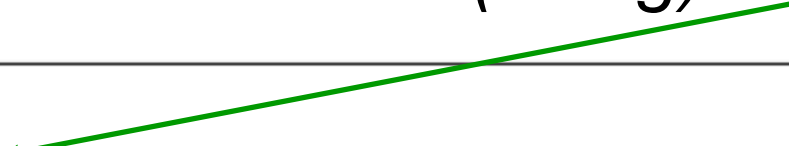
```
struct resultado_leido {  
    int errno;  
    union {  
        tipodato dato;  
    } resultado_leido_u;  
};  
typedef struct resultado_leido resultado_leido;
```

# Lenguaje RPC

- Casos especiales

- Booleano *bool\_t*

- Cadenas de caracteres (*string*) Longitud máxima



```
string nombre<32>;      --->      char *nombre;  
string nombrecompleto<>;  --->      char *nombrecompleto;
```

- Datos opacos (*sin tipo fijo*)

```
opaque bloquedisco[512];  --->      char bloquedisco[512];  
opaque datosf<1024>;      --->      struct {  
                                u_int datosf_len;  
                                char *datosf_val;  
                                } datosf;
```

- Void. Solo en definiciones de union y programas



# Calculadora

- Entrada:

```
<programa> <maquina> <entero> <operador> <entero>
```

- <maquina> es la IP o nombre del servidor
- <operador> puede ser + - \* /
- El cliente filtra la entrada, y llama a la operación correspondiente del servidor
- El servidor realiza la operación y devuelve el resultado.

# DNS

red1

**equipo10** (IP 1.0)  
(Servidor DNS red1)

Nombre	IP
equipo11	1.1
equipo12	1.2
...	...
DNS red2	2.0
DNS red3	3.0

Función DNS2

Función DNS3

**equipo11** (IP 1.1) **equipo12** (IP 1.2) **equipo13** (IP 1.3)

DNS 1.0

DNS 1.0

DNS 1.0

red2

**equipo20** (IP 2.0)  
(Servidor DNS red2)

Nombre	IP
equipo21	2.1
equipo22	2.2
...	...
DNS red1	1.0

**equipo21** (IP 2.1) **equipo22** (IP 2.2) **equipo23** (IP 2.3)

DNS 2.0

DNS 2.0

DNS 2.0

red3

**equipo30** (IP 3.0)  
(Servidor DNS red3)

Nombre	IP
equipo31	3.1
equipo32	3.2
...	...
DNS red1	1.0

**equipo31** (IP 3.1) **equipo32** (IP 3.2) **equipo33** (IP 3.3)

DNS 3.0

DNS 3.0

DNS 3.0

Función DNS1

# DNS

- Necesitan ser cliente y servidor a la vez.
- Crear .x con la función DNS1 solo, para no tener que modificar todos los códigos por separado de todos los DNS
- En el .h añadir manualmente DNS2 y DNS3 con sus número defunción
- En el servidor, cliente y stub se modifica DNS1 por lo que necesitemos (DNS2, DNS3)
- Compilar el servidor con el stub del cliente (o con .o).

# DNS

- El nombre de la máquina lo podeis hacer
  - Cadena caracteres: maquina2.red3
  - Dos valores: maquina2, red3