

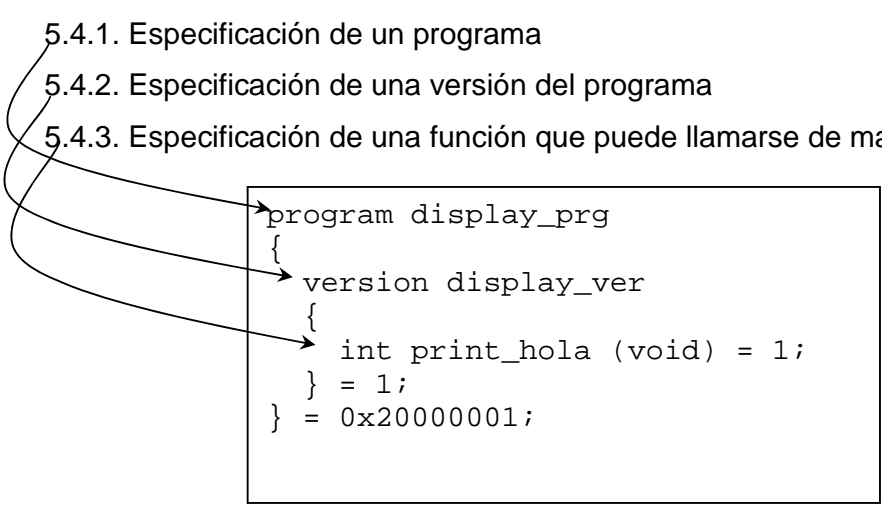
# Sistemas Distribuidos 2009

## Clase 2

### Ejemplo de RPC

1. Abrir una terminal, a partir de aquí se considera que se está en \$HOME, el signo ">" identifica la línea de comandos de la terminal
2. Los archivos a usar están en hola.tar.gz, asumiendo que está en \$HOME:
  - > mkdir rpc
  - > mv hola.tar.gz rpc
  - > cd rpc
  - > gunzip hola.tar.gz
  - > tar -xvf hola.tar.gz
3. Un programa con una única función
  - 3.1. El programa: prg\_comp.c
  - 3.2. La única función: int print\_hola(void)
  - 3.3. Compilar y ejecutar:
    - > gcc -o prg\_comp prg\_comp.c
    - > ./prg\_comp
  - 3.4. Hasta acá no hay nada nuevo
4. A partir de ahora, el objetivo será tener
  - 4.1. Por un lado el programa principal que hace la llamada a la función que no será local, sino que estará en otro proceso. Al programa que hace la llamada se le llama "cliente" o "cliente RPC" en la literatura
  - 4.2. Por otro lado un programa que implementa, contiene y ejecuta la función que es llamada desde otro proceso. Al programa que ejecuta la función se le suele llamar "servidor" o "servidor RPC" en la literatura
  - 4.3. Se hará una especificación .x de lo que se puede llamar de manera remota
  - 4.4. Se generarán "automáticamente" tanto el cliente como el servidor a partir de la especificación
  - 4.5. Se implementarán cliente y servidor de acuerdo a lo que se quiera/necesite

- 5. Para hacer que la función pueda ser llamada desde un proceso cualquiera
  - 5.1. Language de RPC o de especificación RPC: Interface Definition Language
  - 5.2. El IDL es independiente de C, Java, etc., es otro lenguaje
  - 5.3. El IDL no es de ejecución, es declarativo, dice lo que existe, no cómo existe
  - 5.4. Archivo de especificación: hola.x
    - 5.4.1. Especificación de un programa
    - 5.4.2. Especificación de una versión del programa
    - 5.4.3. Especificación de una función que puede llamarse de manera remota



```
program display_prg
{
  version display_ver
  {
    int print_hola (void) = 1;
  } = 1;
} = 0x20000001;
```

- 5.5. El programa, la versión y la/s funciones tienen identificador numérico
  - 5.5.1. Desde 0x20000001 en adelante para programa (es *usual* uno solo)
  - 5.5.2. Desde 1 en adelante para versión (usualmente hay una sola)
  - 5.5.3. Desde 1 en adelante para funciones (pueden haber varias)
- 6. Generación de código

> `rpcgen -a hola.x`

- 6.1. Se generan múltiples archivos:
  - 6.1.1. Los más importantes para los programadores (cliente y servidor):  
hola\_client.c      hola\_server.c
  - 6.1.2. Los que podemos dejar de lado o usar sin conocer por ahora:  
hola\_clnt.c      hola.h      hola\_svc.c      Makefile.hola

- 7. En la implementación del cliente y del servidor, se verá primero el servidor

## 8. Implementación del servidor: hola\_server.c

### 8.1. Se genera solamente la “cáscara” o el “modelo” de funciones del servidor:

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "hola.h"

int *
print_hola_1_svc(void *argp, struct svc_req *rqstp)
{
    static int result;

    /*
     * insert server code here
     */

    return &result;
}
```

### 8.2. Tendrá tantas funciones como se hayan declarado en el .x

### 8.3. El nombre de las funciones se genera con

#### 8.3.1. El nombre declarado

#### 8.3.2. El identificador numérico

#### 8.3.3. El sufijo “\_svc” (asociado a “service”)

#### 8.3.4. De lo anterior queda explicado el por qué de print\_hola\_1\_svc

### 8.4. Valor de retorno de cada función: un puntero al declarado en el .x

#### 8.4.1. Por eso la función retorna int \*

### 8.5. Todas las funciones tendrán dos parámetros

#### 8.5.1. El primero: un puntero al parámetro declarado en el .x, en este caso:

```
void *argp
```

#### 8.5.2. El segundo: que no usaremos y dejaremos sin conocer

### 8.6. Todas las funciones tendrán una variable local “static” del mismo tipo al que apunta el valor de retorno. En este caso se retorna un puntero a entero y por eso la variable local es de static int:

```
static int result;
```

### 8.7. Todas las funciones retornarán el puntero a la variable local “static”:

```
return &result;
```

8.8. El objetivo es agregar código de manera tal que

8.8.1. Se usen los parámetros

8.8.2. Se genere un valor útil en la función para ser retornado

8.8.3. En base al ejemplo anterior, el código correspondiente sería

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "hola.h"

int *
print_hola_1_svc(void *argp, struct svc_req *rqstp)
{
    static int result;

    /*
     * insert server code here
     */

    result = printf("Hola, mundo\n");

    return &result;
}
```

Que es el contenido del archivo hola\_server.c.modif y que se corresponde con el código de la función cuyo objetivo era que se llame de manera remota: print\_hola( )

## 9. Implementación del cliente: hola\_client.c

### 9.1. Tendrá una función local y una función main( )

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "hola.h"

void
display_prg_1(char *host)
{
    CLIENT *clnt;
    int *result_1;
    char *print_hola_1_arg;

#ifdef DEBUG
    clnt = clnt_create (host, display_prg, display_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
    if (result_1 == (int *) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
}

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

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    display_prg_1 (host);
    exit (0);
}
```

### 9.2. El nombre de la función local se genera con

9.2.1. El nombre declarado del programa en el .x

9.2.2. El identificador numérico de la versión del programa declarada en el .x

9.2.3. De lo anterior queda explicado el por qué de display\_prg\_1

9.3. En todos los casos, el contenido de la función main( ) es

9.3.1. El control de que haya un parámetro de línea de comandos

9.3.2. La llamada a la función local con el primer parámetro de línea

(el parámetro de línea de comandos debería ser el nombre-DNS del servidor)

9.4. El contenido de la función local puede dividirse en tres partes:

9.4.1. Administrativo previo a llamadas remotas

9.4.2. Una llamada a cada función/procedimiento remoto

9.4.3. Administrativo posterior a las llamadas remotas

9.5. Administrativo previo a las llamadas remotas

```
#ifndef DEBUG
    clnt = clnt_create (host, display_prg, display_ver, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */
```

Que es siempre igual (independiente de lo que haya en el .x), y podemos considerar que “construye” la conexión con el servidor caracterizada por:

host: nombre de la máquina en la que se ejecuta el proceso servidor

display\_prg: nombre del programa que contiene las funciones remotas

display\_ver: número de versión del programa anterior

"udp": protocolo de transporte de los datos

estimar que

es el contenido del archivo hola\_server.c.modif y que se corresponde con el código de la función cuyo objetivo era que se llame de manera remota: print\_hola( )

9.6. Llamada a cada función/procedimiento remoto

```
result_1 = print_hola_1((void*)&print_hola_1_arg, clnt);
if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
}
```

Que también se hace siempre de la misma manera:

a) Se asignan los parámetros. En este caso no se hace porque no tiene, está declarada con parámetro void

b) se hace la llamada asignando el valor de retorno en la variable result\_ correspondiente

c) Se controla si algo falló comprobando si el valor de retorno es NULL

## 9.7. Finalmente, administrativo posterior a las llamadas remotas

```
#ifndef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
```

Que también es siempre igual, independientemente de lo que haya en el .x

## 9.8. El objetivo es agregar código de manera tal que

9.8.1. Se usen los parámetros

9.8.2. Se use el valor de retorno de la función llamada

9.8.3. En base al ejemplo anterior, correspondería agregar el código

```
if (*result_1 > 0)
    printf("Mision cumplida\n");
else
    printf("Incapaz de mostrar mensaje\n");
```

Inmediatamente después del control de result\_, es decir si la llamada remota no falló. Esto es exactamente lo que se hizo en el archivo hola\_client.c.modif y que se corresponde con el código del programa original prg\_comp.c

## 10. Compilación y ejecución

10.1. Renombrar archivos modificados, para que sean compilados

```
> mv hola_client.c.modif hola_client.c
> mv hola_server.c.modif hola_server.c
```

10.2. Compilar para generar ejecutables

```
> make -f Makefile.hola
```

que genera los ejecutables hola\_client y hola\_server

10.3. Ejecutar en una terminal el servidor

```
> ./hola_server
```

10.4. Ejecutar en otra terminal el cliente

```
> ./hola_client localhost
```

11. Convendría separar lo administrativo de lo importante para la aplicación, que es la propia llamada remota: pre\_rpc( ), llamada\_remota( ), post\_rpc( ), que es lo que está en el archivo hola\_client.c.modif2