

# Programación Distribuida y Tiempo Real

## TP1: sockets y comunicaciones

1. Todos deberían haber recibido respuesta de su entrega
2. Como con toda API, no interactuamos con lo que “representa”
3. Sockets
  - 3.1. C: interactuamos con un file descriptor
  - 3.2. Java: interactuamos con DataInputStream o DataOutputStream
4. Específicamente read: en ningún caso “exactamente”
  - 4.1. “up to”
  - 4.2. “some”
5. Problema ==> buscar o preguntar por solución...
6. Que read no nos devuelva todo no significa que no haya llegado... SO
  - 6.1. Descartar reenvío
  - 6.2. Descartar “paquetizar” (...¿qué tamaño?)
  - 6.3. Sí hay que volver a leer
7. Volver a leer:
  - 7.1. A partir de lo que se leyó
  - 7.2. Lo que resta por leer
  - 7.3. No

```
while (bytesTotal < limit)
{
    /* Recv data from client */
    sizeData=fromclient.read(bufferReceived);
    bytesTotal=bytesTotal +sizeData;
}
```

## 8. No son útiles

### 8.1. Crear un String a partir de un buffer con datos “desconocidos”

```
byte[] buffer = new byte[10000];  
...  
fromclient.read(buffer);  
String str = new String(buffer);  
...  
System.out.println("Longitud: "+str.length());
```

### 8.2. “Combinaciones” como

```
byte[] buffer;  
buffer = new byte[1000000];  
  
/* Recv data from client */  
fromclient.read(buffer);  
  
//Chequeo de llegada correcta de datos  
ArrayList<Byte> datosDeLlegada = new ArrayList<Byte>();  
for(Byte b:buffer)  
{  
    datosDeLlegada.add(b);  
}  
if(datosDeLlegada.size() == 1000000)
```

## 9. Otras alternativas

9.1. “Codificar” el final ==> codificar el contenido

9.2. “Protocolo”: cantidad, msg

9.3. En Java: readFully(byte[] b, int off, int len)

9.4. ...

## 10. Al margen...

```
String inputline = "";  
for(int i = 0; i < len; i++)  
{  
    inputline = inputline + 'A';  
}  
  
/* Get the bytes... */  
buffer = inputline.getBytes();  
  
/* Send read data to server */  
toserver.write(buffer, 0, buffer.length);
```

# Programación Distribuida y Tiempo Real

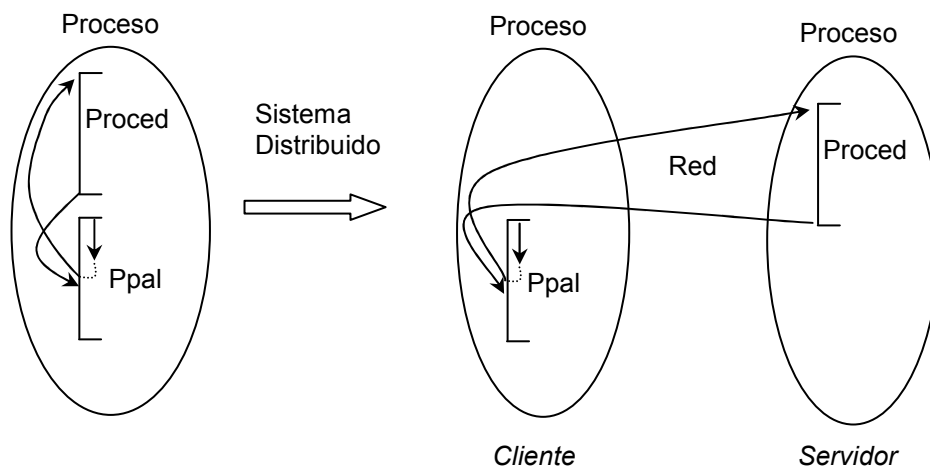
## TP2: RPC

### 1. Mecanismos iniciales/sencillos para programar en sistemas distribuidos

- 1.1. Sockets ¿programar o comunicar?
- 1.2. Procedural: RPC
- 1.3. Orientado a Objetos: Java RMI
- 1.4. Ambos se *presentan* como cliente/servidor
- 1.5. Qué tienen en común
- 1.6. Qué tienen de diferente
- 1.7. Historia del desarrollo

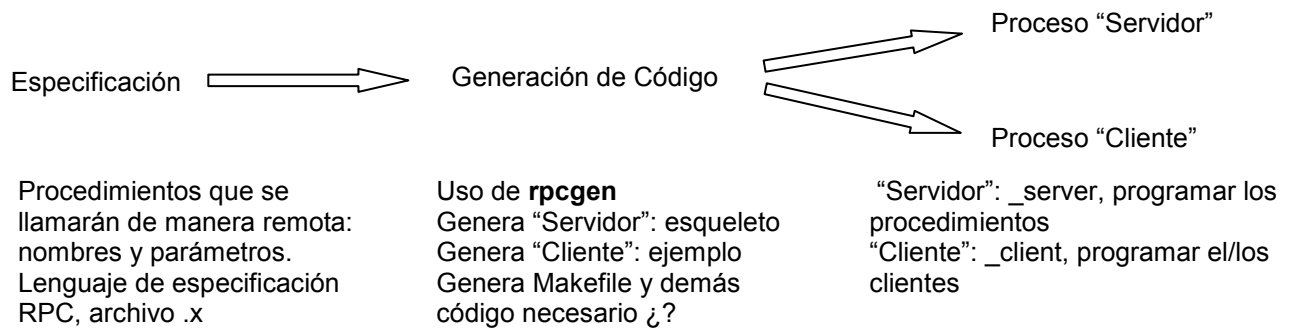
### 2. RPC: Remote Procedure Call (ONC RPC)

- 1.1. RPC: programación procedural para sistemas distribuidos, *extensión* del modelo de programación procedural para sistemas distribuidos



- 1.2. XDR: antecesor de RPC. Fin de los '80 inicio de los '90: dado que se programa sobre TCP/IP y no hay representación de datos ==> Sun define una representación de datos independiente de todo lo demás. RFC 1832 - XDR: External Data Representation Standard,  
<http://tools.ietf.org/html/rfc4506>
- 1.3. Agregado de RPC sobre XDR: no mucho más que la especificación de los propios procedimientos que se pueden llamar de manera remota. RFC 1831 - RPC: Remote Procedure Call Protocol Specification Version 2,  
<http://tools.ietf.org/html/rfc5531>
- 1.4. RPC hace posible que se invoque un procedimiento que está en otro proceso en otra máquina.

## 2. Esquema general del proceso de *desarrollo*:



## 3. Ejemplo: `hola_rpc.tar`

### 3.1. Archivos de especificación: .x

### 3.2. Uso de `rpcgen` en Linux

### 3.3. Ejemplo *paso a paso* de `prg_comp.c` hacia `hola.x` (y sus asociados)

#### 3.3.1. `prg_comp` tiene un procedimiento, no está distribuido

#### 3.3.2. A partir de `hola.x` (la especific.) se llegará a "lo mismo" pero con RPC

### 3.4. Cáscaras-esqueletos de programas

### 3.5. Ideas de *stubs-representantes-talones* de los procedimientos

### 3.6. Modelo cliente/servidor de base (hasta en los nombres que genera `rpcgen`): `_client` `_server`: `_client` es el que hace la llamada y `_server` es el que *tiene* y ejecuta el procedimiento llamado

## 4. Abrir una terminal, a partir de aquí se considera que se está en `$HOME`, el signo ">" identifica la línea de comandos de la terminal

## 5. Los archivos a usar están en `hola_rpc.tar`, asumiendo que está en `$HOME`:

```
> cd
```

```
> tar -xvf hola_rpc.tar
```

Se tendrá un directorio `rpc` con dos directorios: `sindistr` y `rpcdistr`

## 6. Directorio `sindistr`: Un programa con una única función

### 6.1. El programa: `prg_comp.c`

### 6.2. La única función: `int print_hola(void)`

### 6.3. Compilar y ejecutar:

```
> cd rpc/sindistr
```

```
> gcc -o prg_comp prg_comp.c
```

```
> ./prg_comp
```

### 6.4. Hasta acá no hay nada nuevo

## 7. A partir de ahora, el objetivo será tener

- 7.1. 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
- 7.2. 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
- 7.3. En el ejemplo, será

`int print_hola(void)`

el “procedimiento remoto”

- 7.4. Se hará una especificación .x de lo que se puede llamar de manera remota
- 7.5. Se generarán “automáticamente” tanto el cliente como el servidor a partir de la especificación
- 7.6. En general, se implementarán cliente y servidor de acuerdo a lo que se quiera/necesite

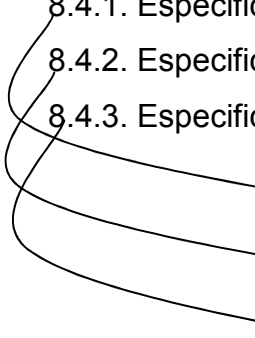
## 8. Para hacer que la función pueda ser llamada desde un proceso cualquiera

- 8.1. Language de RPC o de especificación RPC: Interface Definition Language
- 8.2. El IDL es independiente de C, Java, etc., es otro lenguaje
- 8.3. El IDL no es de ejecución, es declarativo, dice lo que existe, no cómo existe
- 8.4. Archivo de especificación: hola.x (en el directorio rpc/rpcdistr)

### 8.4.1. Especificación de un programa

### 8.4.2. Especificación de una versión del programa

### 8.4.3. Especificación de una función que puede llamarse de manera remota



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

## 8.5. El programa, la versión y la/s funciones tienen identificador numérico

- 8.5.1. Desde 0x20000001 en adelante para programa (es *usual* uno solo)
- 8.5.2. Desde 1 en adelante para versión (usualmente hay una sola)
- 8.5.3. Desde 1 en adelante para funciones (pueden haber varias)

## 9. Generación de código

> rpcgen -a hola.x

### 9.1. Se generan múltiples archivos:

#### 9.1.1. Los más importantes para los programadores (cliente y servidor):

hola\_client.c          hola\_server.c

#### 9.1.2. Los que podemos dejar de lado o usar sin conocer por ahora:

hola\_clnt.c          hola.h          hola\_svc.c          Makefile.hola

## 10. En la implementación del cliente y del servidor, se verá primero el servidor

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

### 11.1. 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;
}
```

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

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

#### 11.3.1. El nombre declarado

#### 11.3.2. El identificador numérico

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

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

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

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

### 11.5. Todas las funciones tendrán dos parámetros

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

```
void *argp
```

11.5.2. El segundo: que no usaremos y dejaremos sin conocer

11.6. 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;
```

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

```
return &result;
```

11.8. El objetivo es agregar código de manera tal que

11.8.1. Se usen los parámetros

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

11.8.3. 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( )

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

### 12.1. 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);
}
```

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

12.2.1. El nombre declarado del programa en el .x

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

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



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

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

12.3.2. 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)

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

12.4.1. Administrativo previo a llamadas remotas

12.4.2. Una llamada a cada función/procedimiento remoto

12.4.3. Administrativo posterior a las llamadas remotas

12.5. 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

12.6. 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

12.7. 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

12.8. El objetivo es agregar código de manera tal que

12.8.1. Se usen los parámetros

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

12.8.3. 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

### 13. Compilación y ejecución

13.1. Renombrar archivos modificados, para que sean compilados

> mv ../hola\_client.c.modif hola\_client.c

> mv ../hola\_server.c.modif hola\_server.c

13.2. Compilar para generar ejecutables

> make -f Makefile.hola

que genera los ejecutables hola\_client y hola\_server

13.3. Ejecutar en una terminal el servidor

> ./hola\_server

Posibles fallas en Ubuntu:

a) Falta rpcbind:

> sudo apt-get install aptitude

> sudo aptitude install portmap

a) Falta reiniciar "inseguro":

sudo -i service rpcbind stop; sudo -i rpcbind -i -w; sudo -i service rpcbind start

o

service rpcbind stop; rpcbind -i -w; service rpcbind start

13.4. Ejecutar en otra terminal el cliente

> ./hola\_client localhost

14. 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`

<http://download.oracle.com/docs/cd/E19683-01/816-1435/rpcgenpguide-41939/index.html>

(opción -N)

13. En caso de ser necesario: imagen de disco para instalar máquina virtual

14. RPC Concurrente... ¿o paralelo?

14.2. ¿Varios clientes a la vez?

14.3. ¿Un servicio más rápido?

14.4. ¿Ambos?