

# VIRTUALIZACIÓN Y SISTEMAS OPERATIVOS AVANZADOS

## GUÍA 2. SOCKETS Y RPC

### OBJETIVO

- Implementar una aplicación cliente/servidor para intercambiar información entre dos nodos.
- Se implementarán dos soluciones, una con Sockets y otra con Remote Procedure Calls (RPC).

### PREPARACIÓN CONTENEDORES

Para este trabajo práctico, se continuará utilizando la máquina virtual, con el sistema Ubuntu 22.04 (y que hacía las veces de sistema host), utilizada en el trabajo anterior.

En este caso, se utilizarán dos contenedores LXC distintos dentro de la maquina virtual, uno simular un equipo cliente y otro simular un servidor.

Para crear estos contenedores, se deberá ejecutar el siguiente script desde una terminal de la máquina virtual host:

```
sudo lxc launch ubuntu:jammy ubuntuCClient
```

```
sudo lxc launch ubuntu:jammy ubuntuCServer
```

Luego, cuando se quiera acceder a una terminal en cada contenedor (como usuario root), ejecutar lo siguiente (cada comando en una terminal separada):

```
lxc exec ubuntuCClient bash
```

```
lxc exec ubuntuCServer bash
```

Una vez dentro de los contenedores, ejecutar lo siguiente (en cada uno) para configurar el usuario ubuntuadm dentro de los contenedores:

```
usermod -l ubuntuadm ubuntu
```

```
sudo groupmod -n ubuntuadm ubuntu
```

```
sudo usermod -d /home/ubuntuadm -m ubuntuadm
```

Para establecer una pass nueva para el usuario ubuntuadm en los contenedores usar lo siguiente:

```
sudo passwd ubuntuadm
```

Se sugiere usar como password ubuntuadm.

Adicionalmente ejecutar lo siguiente para configurar el acceso con password a través de ssh:

```
sed -i "s/PasswordAuthentication no/PasswordAuthentication yes/" /etc/ssh/sshd_config
```

```
systemctl restart sshd
```

Con esto, será posible acceder por ssh a los contenedores si es necesario. Para ver las direcciones IP asignadas, ejecutar el comando:

```
lxc list
```

Una vez configurados los contenedores, se debe instalar rpcbind en cada uno. Para eso ejecutar el siguiente comando en cada contenedor:

```
sudo apt-get install rpcbind
```

Finalmente se debe ejecutar lo siguiente en la máquina virtual host (para instalar la librería requerida para usar RPC):

```
sudo apt install libntirpc-dev
```

## INICIAR Y CONECTAR A CONTENEDORES

Para iniciar los contenedores LXC, usar el siguiente comando (si es que no están iniciados):

```
lxc start ubuntuCClient
```

```
lxc start ubuntuCServer
```

Si no es posible (o no se quiere) por algún motivo conectar por SSH a los contenedores, se puede ejecutar lo siguiente dependiendo el contenedor al cual se quiere conectar:

```
lxc exec ubuntuCClient -- su --login ubuntuadm
```



```
lxc exec ubuntuCServer -- su --login ubuntuadm
```

## USO SOCKETS

Para escribir programas que utilicen sockets, se usará la librería de C sys/socket.h.

Junto con la guía y dentro de la carpeta SOCKET, comprimida en el archivo codigos.zip, se puede encontrar un archivo client.c y server.c (tomados de ejemplo de <https://www.geeksforgeeks.org/socket-programming-cc/>).

En estos códigos se hace uso de distintas funciones para la comunicación con sockets. Las mismas serán descritas a continuación (se puede encontrar más información en <https://linux.die.net/man/2/>).

## FUNCIONES SERVIDOR

### 1. Creación Socket



*int socket(int domain, int type, int protocol)*

domain: Dominio de comunicación: AF\_INET (IPv4), AF\_INET6 (IPv6).

type: Tipo de comunicación: SOCK\_STREAM (TCP), SOCK\_DGRAM (UDP).

protocol: Protocolo de internet. 0 es IP.

Retorna un descriptor del socket.

### 2. Setear las opciones del socket

*int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen)*

sockfd: Descriptor del socket devuelto por la función socket.

level: Nivel para el cual se setea la opción. SOL\_SOCKET (opciones a nivel socket), IPPROTO\_IP (opciones a nivel IP), IPPROTO\_IPV6.

optname: Nombre de la opción. SO\_REUSEADDR, SO\_REUSEPORT: Se setean en 1 para permitir reutilizar IP y puerto en diferentes conexiones.

optval: Puntero a los datos de la opción.

optlen: Longitud de optval.

### 3. Bind

*int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)*

sockfd: Descriptor del socket.

addr: Estructura que contiene la dirección IP y el numero de puerto, entre otras cosas.

addrlen: Longitud de addr.

### 4. Listen

*int listen(int sockfd, int backlog)*

sockfd: Descriptor del socket.

backlog: Define el valor máximo de conexiones pendientes para el socket.

### 5. Accept

*int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen)*

En caso de éxito, esta llamada al sistema devuelve un descriptor de archivo para el socket aceptado para la conexión específica. Este descriptor o handle se usará para leer los datos que son recibidos desde un cliente específico, y para enviar datos a un cliente específico. Luego de utilizar el canal de comunicación, se debe cerrar el mismo usando el descriptor.

## FUNCIONES CLIENTE

### 1. Creación del socket

Igual que para el caso del servidor.

### 2. Conversión de IPs de texto a forma binaria

*int inet\_pton(int af, const char \*restrict src, void \*restrict dst)*

af: Tipo de IP (IPv4 o IPv6).

src: Origen de la IP (texto con la dirección IP).

dst: Destino a donde escribir la IP en forma binaria.

### 3. Conectarse al servidor

*int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)*

## COMUNICACIÓN ENTRE CLIENTE Y SERVIDOR

### 1. Enviar el mensaje al otro extremo (cliente o servidor).

*ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags)*

sockfd: File descriptor del socket usado.

buf: Espacio de memoria (variable) donde se encuentra el contenido a enviar.

len: Longitud del contenido a enviar.

flags: 0.

Retorna la cantidad de bytes enviados, -1 si hubo error.

## **2. Recibir el mensaje.**

*ssize\_t read(int fd, void \*buf, size\_t count)*

fd: File descriptor del socket usado.

buf: Espacio de memoria (variable) donde se escribirá el contenido leído.

count: Cantidad de caracteres a ser leídos.

Retorna la cantidad de bytes leídos, -1 si hubo error.

## **COMPILACIÓN DE ARCHIVOS**

Para compilar los archivos fuente para el cliente y servidor, primero se deberá transferir cada archivo al contenedor correspondiente. Para hacerlo, ejecutar el comando que se utilizó en el TP1 que tenía la siguiente estructura:

```
scp <archivo> ubuntuadm@<ip>:<dir-destino>
```

Para obtener las IP's destino de los contenedores ejecutar el comando `lxc list`. Elegir además un directorio destino válido/existente (se recomienda `/home/ubuntuadm/`).

Si hay problemas para transferir los archivos de esta manera, se puede escribir el código utilizando el editor nano en cada contenedor. La forma de usarlo es:

```
nano <nombre-archivo-a-crear>
```

Una vez que se disponga de los códigos fuente en cada contenedor, se puede compilar los mismos de la siguiente manera:

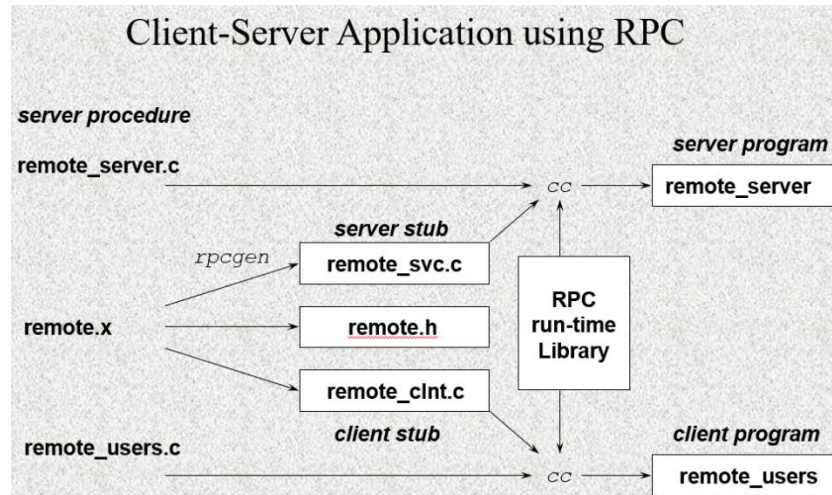
```
gcc -o client client.c
gcc -o server server.c
```

## **ACTIVIDADES**

Los archivos provistos `client.c` y `server.c` realizan un intercambio de mensajes entre el contenedor cliente y el contenedor servidor. Con estos archivos, realizar lo siguiente:

1. Modificar el código cliente para conectar al equipo y programa servidor. Se debe poder especificar la IP del servidor como parámetro en el programa cliente.
2. Modificar el código del servidor para que, una vez que responde a una solicitud del cliente, se quede a la espera de la siguiente petición.

## **GUIA RPC**



## INTRODUCCIÓN

Se utilizará la herramienta `rpcgen` (<https://linux.die.net/man/1/rpcgen>). Esta herramienta genera el código C necesario para implementar RPC en forma de un archivo cabecera (.h), y un stub tanto para el cliente como para el servidor. Estos archivos se incluyen luego en los archivos fuente de los programas para cliente y servidor.

La función de los archivos stub es tomar el lugar del procedimiento a ejecutar, que en realidad está ubicado en un equipo remoto, para atender la llamada al procedimiento desde el programa local.

En la carpeta RPC, comprimida en `codigos.zip`, se puede encontrar el archivo `add.x` que se usará para las siguientes actividades.



## GENERACIÓN DE CODIGO FUENTE

**rpcgen** toma como entrada un archivo con extensión .x. Este archivo describe una lista de procedimientos, que pueden ser ejecutados remotamente, con sus parámetros (sin sus implementaciones), y está en formato RPCL (Remote Procedure Call Language).

Para usar el archivo con `rpcgen`, ejecutar:

```
rpcgen -C add.x
```

La ejecución de este comando generará los siguientes archivos:

```
add_clnt.c (stub cliente)
add_svc.c (stub servidor)
add_xdr.c (XDR)
add.h (Archivo header - compartido por cliente y servidor)
```

En este caso, no se genera un archivo XDR porque la aplicación usa solo los tipos de datos básicos que se encuentran en la librería `libnsl` (que se incluye luego al compilar). La representación de datos externos (XDR) es un formato estándar de serialización de datos, para usos como protocolos de redes informáticas. Permite transferir datos entre diferentes tipos de sistemas informáticos. La conversión de la representación local a XDR se denomina codificación. La conversión de XDR a la representación local se

denomina decodificación. XDR se implementa como una biblioteca de software de funciones que se puede usar entre diferentes sistemas operativos y también es independiente de la capa de transporte.

### Generar códigos de ejemplo para el cliente y el servidor:

En la carpeta RPC se podrá encontrar dos archivos de ejemplo (llamados `add_client.c` y `add_server.c`), que pueden ser compilados como programas que se ejecuten en el lado cliente y el lado servidor (como se mostrará en la siguiente sección). Sin embargo, con `rpcgen` se pueden generar códigos de ejemplo para ambos casos. Esto se puede hacer como se muestra a continuación:

```
rpcgen -C -Sc add.x > addclient.c
rpcgen -C -Ss add.x > addservices.c
```


En el caso del cliente, se genera un código de ejemplo para mostrar el uso del procedimiento remoto y cómo vincular el servidor antes de llamar a los stubs del lado del cliente generados por `rpcgen`.

Para el servidor, se genera un “esqueleto” de código para el procedimiento remoto en el lado del servidor. Se debe completar el código real para los procedimientos remotos.


### COMPILACION DE PROGRAMAS CLIENTE/SERVIDOR


Para compilar los programas en cuestión que se ejecutarán en el cliente y en el servidor, es necesario incluir todos aquellos archivos fuente que son necesarios para proveer y consumir el procedimiento remoto.


#### Archivo de entrada para *rpcgen*


 `add.x`

#### Archivos generados por *rpcgen*


 `add_svc.c`


 `add_xdr.c`

 `add_clnt.c`

 `add.h`

#### Archivos a ser compilados para ejecutar en cliente y servidor

 `add_server.c`

 `add_client.c`

A continuación, se ejemplifica como sería el comando para cada caso utilizando los códigos ejemplo generados con `rpcgen`:

Servidor:

```
gcc -g -o add_server add_svc.c add_server.c add_xdr.c -I/usr/include/tirpc -lnsl -ltirpc
```

Cliente:

```
gcc -g -o add_client add_clnt.c add_client.c add_xdr.c -I/usr/include/tirpc -lnsl -ltirpc
```

Tener en cuenta que, como existe un archivo XDR (add\_xdr.c), debe ser incluido también.

Como se menciono antes, en la carpeta RPC se podrá encontrar dos archivos de ejemplo (llamados add\_client.c y add\_server.c), que pueden ser compilados como programas que se ejecuten en el lado cliente y el lado servidor (como se mostrará en la siguiente sección).

## ACTIVIDADES

Utilizando el archivo add.x como base, realizar:

1. Compilar los códigos para cliente y servidor.
2. Modificar el cliente y el servidor para que realicen un intercambio de mensajes similar al realizado utilizando sockets.
3. Agregar en el servidor una función que permita sumar dos números y devolver el resultado, y su correspondiente llamada desde el cliente.

## MAS INFORMACIÓN SOBRE RPCGEN

<https://linux.die.net/man/1/rpcgen>

<https://freetechorb.wordpress.com/2019/04/28/adding-two-integers-using-rpcgen/>

<https://freetechorb.wordpress.com/2019/03/24/centralized-to-distributed-what-is-rpc-and-how-it-works/>

<https://docs.oracle.com/cd/E19683-01/816-1435/6m7rrfn7k/index.html>