Programación Distribuida y Tiempo Real

Ulises Jeremias Cornejo Fandos[1]

[1] *Licenciatura en Informática, Facultad de Informática, UNLP*

compiled: October 2, 2019

1. Para los ejemplos de RPC proporcionados (*.tar, analizar en el orden dado a los nombres de los archivos):

1.A. Mostrar cómo serían los mismos procedimientos si fueran locales, es decir haciendo el proceso inverso del realizado en la clase de explicación de RPC.

Si los procesos fueran todos locales, la programación de los mismos resultaria en una compilación de uno o más ".c" con una única función *main* y un único binario ejecutable. En el ejemplo provisto se puede observar como todo el código del programa se encuentra en un único ".c", como se describe anteriormente.

En este caso no habría latencia en la comunicación dado que no existiría tal comunicacón. Al no definirse un modelo cliente/servidor, ya no existiría una comunicación entre hosts y todos los llamados a funciones se realizarían en el mismo espacio de direcciones.

1.B. Ejecutar los procesos y mostrar la salida obtenida (del "cliente" y del "servidor") en cada uno de los casos.

Las capturas se muestran a continuación:

1.B.1. Simple

```
root@0584e584b861:/pdytr/practica-2/resources/src/1-simple# ./client localhost 3 2
3 + 2 = 5
3 - 2 = 1
root@0584e584b861:/pdytr/practica-2/resources/src/1-simple#
```

Fig. 1. Salida de la ejecución del cliente dados los parámetros *localhost*, *3* y *2*

.

```
root@0584e584b861:/pdytr/practica-2/resources/src/1-simple# ./server
Got request: adding 3, 2
Got request: subtracting 3, 2
```

Fig. 2. Salida de la ejecución del servidor.

1.B.2. U1

La ejecución del servidor no tiene salida al momento de resolver una consulta del cliente.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/2-u1# ./client localhost 3
UID 3, Name is sys
root@0584e584b861:/pdytr/practica-2/resources/examples/2-u1#
```

Fig. 3. Salida de la ejecución del cliente dados los parámetros *localhost* y *3*

.

1.B.3. Array

```
root@0584e584b861:/pdytr/practica-2/resources/examples/3-array# ./vadd_client localhost
8 3 5 9 2
8 + 3 + 5 + 9 + 2 = 27
root@0584e584b861:/pdytr/practica-2/resources/examples/3-array#
```

Fig. 4. Salida de la ejecución del cliente dados los parámetros *localhost*, 8, 3, 5, 9 y 2

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/3-array# ./vadd_service
Got request: adding 5 numbers
```

Fig. 5. Salida de la ejecución del servidor.

1.B.4. List

La ejecución del servidor no tiene salida al momento de resolver una consulta del cliente.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/4-list# ./client localhost 8 3 5
9 2
8 3 5 9 2
Sum is 27
root@0584e584b861:/pdytr/practica-2/resources/examples/4-list#
```

Fig. 6. Salida de la ejecución del cliente dados los parámetros *localhost*, 8, 3, 5, 9 y 2

.

1.C. Mostrar experimentos donde se produzcan errores de conectividad del lado del cliente y del lado del servidor. Si es necesario realice cambios mínimos para, por ejemplo, incluir sleep() o exit(), de forma tal que no se reciban comunicaciones o no haya receptor para las comunicaciones. Verifique con UDP y con TCP.

En ambos casos se modifica la implementación de la función `int *add_1_svc(operands *argp, struct svc_req *rqstp);` en el archivo *simpservice.c*. Es decir, la implementación del proceso servidor, agregando un sleep o exit, según corresponda, respectivamente.

1.C.1. UDP

Agregando un llamado a la función **exit**, el proceso servidor termina en medio de la operación, (*ver fig. 7*) por lo que el cliente termina la comunicación cuando finaliza el timeout definido en su proceso, (*ver fig. 8*).

Lo mismo pasa si el servidor tarda mucho en procesar lo pedido y responder, cosa que se simula agregando un llamadao a la función **sleep** dentro del proceso servidor. Cuando el sleep tiene una duración mayor a la del timeout definido en el cliente, el servidor no da su respuesta antes del timeout y el cliente cierra la conexión, (*ver fig. 10*). En este caso, el servidor continuará su ejecución esperando el proximo request, (*ver fig. 9*).

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./server
Got request: adding 3, 2
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 7. Salida de la ejecución del service utilizando UDP y la función exit.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./client localhost 3 2
Trouble calling remote procedure
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 8. Salida de la ejecución del cliente utilizando UDP.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./server
Got request: adding 3, 2
Got request: adding 3, 2
Got request: adding 3, 2
Got request: adding 3, 2
Got request: adding 3, 2
```

Fig. 9. Salida de la ejecución del service utilizando UDP y la función sleep.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./client localhost 3 2
Trouble calling remote procedure
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 10. Salida de la ejecución del cliente utilizando UDP.

.

1.D. TCP

El resultado es exactamente el mismo que el descripto anteriormente en el caso de UDP, cuando el servidor termina cuando se agrega el **exit**, (*ver figs. 11 y 12*) y cuando tarda mas de lo debido con la función **sleep**, (*ver figs. 13 y 14*).

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./server
Got request: adding 3, 2
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 11. Salida de la ejecución del service utilizando TCP y la función exit.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./client localhost 3 2
Trouble calling remote procedure
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 12. Salida de la ejecución del cliente utilizando TCP.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./server
Got request: adding 3, 2
```

Fig. 13. Salida de la ejecución del service utilizando TCP y la función sleep.

.

```
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple# ./client localhost 3 2
Trouble calling remote procedure
root@0584e584b861:/pdytr/practica-2/resources/examples/1-simple#
```

Fig. 14. Salida de la ejecución del cliente utilizando TCP.

.

2. Describir/analizar las opciones a) - N b) -M y - A, verificando si se pueden utilizar estas opciones y comentar que puede ser necesario para tener procesamiento concurrente del "lado del cliente" y del "lado del servidor" con la versión utilizada de rpcgen. Una lista completa de opciones se describe en http://download.oracle.com/docs/cd/E19683-01/816-1435/rpcgenpguide-1939/index.html

2.A.

El flag -N intenta de generar código en un estándar más nuevo que el ANSI (flag -C). Al intentar compilar el código con ese flag, utilizando el código original, falla. Esto se debe a que rcpgen, genera un .h las funciones con el tipo SIN puntero, a diferencia del flag -C, que los genera como punteros a operand.

2.B.

El flag -M sirve para generar código seguro para la concurrencia, utilizando un paŕametro extra al servicio, de tipo int *.
El flag -A, es la configuracion por default, que dependiendo el sistema en el que se compila, va a ser (o no), seguro para la concurrencia multihilo.

3. Analizar la transparencia de RPC en cuanto al manejo de parámetros de los procedimientos remotos. Considerar lo que sucede en el caso de los valores de retorno. Puede aprovechar los ejemplos provistos.

rpcgen utiliza la estructura definida en el archivo ".x" para generar estructuras C en ambos puntos (cliente y servidor), con las cuales va a trabajar casteando.

El servicio recibe punteros a estas estructuras, las cuales va a trabajar y retornar nuevamente casteando a (caddr_t), el cual es equivalente a un `void *`.

Esto nos permite trabajar con cualquier tipo de C, siempre volviendo a castear a la estructura definida a partir del ".x".