Programación Distribuida y Tiempo Real

Ulises Jeremias Cornejo Fandos[1]

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

compiled: November 3, 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".

4. Con la finalidad de contar con una versión muy restringida de un sistema de archivos remoto, en el cual se puedan llevar a cabo las operaciones enunciadas informalmente como:

- **leer**: dado un nombre de archivo, una posición y una cantidad de bytes a leer, retorna 1) los bytes efectivamente leídos desde la posición pedida y la cantidad pedida en caso de ser posible, y 2) la cantidad de bytes que efectivamente se retornan leídos.
- **escribir**: dado un nombre de archivo, una cantidad de bytes determinada, y un buffer a partir del cual están los datos, se escriben los datos en el archivo dado. Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos. En todos los casos se retorna la cantidad de bytes escritos.

4.A. Defina e implemente con RPC un servidor. Documente todas las decisiones tomadas.

4.B. Implemente un cliente RPC del servidor anterior que copie un archivo del sistema de archivos del servidor en el sistema de archivos local y genere una copia del mismo archivo en el sistema de archivos del servidor. En todos los casos se deben usar las operaciones de lectura y escritura del servidor definidas en el item anterior, sin cambios específicos del servidor para este item en particular. Al finalizar la ejecución del cliente deben quedar tres archivos en total: el original en el lado del servidor, una copia del original en el lado del cliente y una copia en el servidor del archivo original. El comando diff no debe identificar ninguna diferencia entre ningún par de estos tres archivos.

Para este punto, se implementaron las funciones básicas, `write`, `read` y `list`

Como primera instancia, se define el archivo `.x` de la siguiente forma:

```
#define VERSION_NUMBER 1

%#define DATA_SIZE UINT_MAX

struct ftp_file {
    string name<PATH_MAX>;
        opaque data<>;
    int continue_reading;
};

struct ftp_wfile {
    string name<PATH_MAX>;
    string mode<>;
        opaque data<>;
        uint64_t checksum;
};

struct ftp_lreq {
    string name<PATH_MAX>;
    int all;
};

struct ftp_req {
    string name<PATH_MAX>;
    uint64_t pos;
    uint64_t bytes;
};

program FTP_PROG {
    version FTP_VERSION {
        ftp_file READ(ftp_req) = 1;
        int WRITE(ftp_wfile) = 2;
        string LIST(ftp_lreq) = 3;
    } = VERSION_NUMBER;
} = 555555555;

#define FTP_PROG 555555555
```

Se busco manejar cualquier tipo de archivos, por eso el uso del tipo opaque.

Luego se definieron el cliente junto a los comandos disponibles.

Con el objetivo de mejorar la legibilidad del codigo, se opto por utilizar un arreglo de estructuras con punteros a funciones para diferenciar rapidamente los comandos.

El parseo de los argumentos al cliente cuenta con 2 etapas, la primera busca algun comando valido, y se realiza a recorriendo manualmente `argv`.
La segunda busca las flags posibles para los comandos, esto se hizo usando la libreria `getopt` para mayor facilidad.

La conexion con el servidor tuvo que ser obligatoriamente del tipo **TCP** para soportar los grandes volumenes de datos.

5. Timeouts en RPC:

5.A. Desarrollar un experimento que muestre el timeout definido para las llamadas RPC y el promedio de tiempo de una llamada RPC.

Para el experimento del timeout usaremos el caso 1-simple. Vemos que si ponemos en el proceso servidor un delay de 25 mediante un sleep(25) en el medio de la comunicación:

Entonces la comunicación se realiza exitosamente y la llamada al proceso remoto termina de forma correcta.

En cambio si ponemos en el proceso servidor un delay de 26 mediante un sleep(26) en el medio de la comunicación.

Entonces la comunicación se corta por parte del cliente y la llamada no se completa. (*Ver fig. 15*).

Mediante estos resultados podemos afirmar que el timeout definido en el proceso cliente es el 25 segundos, por los que si el proceso servidor o la comunicación tardan más de ese tiempo entonces el cliente finalizará la comunicación abruptamente dando por sentado una falla en la misma.

```
root@6c2c0819b695:/pdytr/6-sleep/a# for i in {1..10}; do ./client localhost 10 3; done
Time elapsed at the time of the timeout 25.0251
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0278
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0277
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0277
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0242
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0239
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0229
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0178
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0216
Trouble calling remote procedure
Time elapsed at the time of the timeout 25.0236
Trouble calling remote procedure
root@6c2c0819b695:/pdytr/6-sleep/a#
```

Fig. 15. Tiempos de ejecución calculados al momento de alcanzar el timeout

.

5.B. Reducir el timeout de las llamadas RPC a un 10% menos del promedio encontrado anteriormente. Mostrar y explicar el resultado para 10 llamadas.

Luego de enviar una solicitud al servidor, un proceso cliente espera un período default de 25 segundos para recibir una respuesta. Este valor puede ser modificado utilizando la rutina `clnt_control()`.

Usando una estructura timeval, se puede configurar los segundos que se deseen para el timeout del cliente se la siguiente forma:

```
struct timeval tv;
CLIENT *clnt;

tv.tv_sec = 60;
tv.tv_usec = 0;

clnt_control(clnt, CLSET_TIMEOUT, &tv);
```

Teniendo en cuenta esto, se prueba con un timeout de 10 segundos, realizando el experimento anterior nuevamente 10 veces obteniendo el siguiente resultado. (*Ver fig. 16*).

```
root@6c2c0819b695:/pdytr/6-sleep/b# for i in {1..10}; do ./client localhost 10 3; done
Time elapsed at the time of the timeout 10.0088
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0109
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0111
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0099
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0086
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0093
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0097
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0087
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0078
Trouble calling remote procedure
Time elapsed at the time of the timeout 10.0107
Trouble calling remote procedure
root@6c2c0819b695:/pdytr/6-sleep/b#
```

Fig. 16. Tiempos de ejecución calculados al momento de alcanzar el timeout

.

5.C. Desarrollar un cliente/servidor RPC de forma tal que siempre se supere el tiempo de timeout. Una forma sencilla puede utilizar el tiempo de timeout como parámetro del procedimiento remoto, donde se lo utiliza del lado del servidor en una llamada a sleep(), por ejemplo.

En la estructura cliente/servidor generada, se recibe como argumento un tiempo el cual define el timeout del cliente y se envia al servidor para que este haga un sleep con el cuadrado de ese tiempo para que siempre se genere un timeout.