



UNIVERSIDAD
NACIONAL
DE LA PLATA

Programación distribuida y de tiempo real

Práctica 2 - RPC

Alumnos:

Alan Fabian Castelli (726/9)

Franco Daniel Fico (1006/7)



Introducción

Se pone en práctica el protocolo de comunicación de RPC (Remote Procedure Call), en la cual se podrá observar los conceptos de UDP, TCP, la implementación de un servidor de archivo (utilizando .pdf, .png, .txt y binarios para su transferencia) y análisis de Time out, teniendo en cuenta que la programación implementada fue lenguaje C.

Comandos útiles

Antes de realizar los ejercicios de la práctica, se presentan los comandos necesarios para generar los archivos en lenguaje C e implementar la comunicación RPC entre proceso cliente y servidor.

En un principio, se define una estructura en un archivo de extensión *.x, el cual tendrá los procedimientos y variables que se “compartirán” entre el proceso cliente y el servidor en el momento de la comunicación. Para esto se utiliza el comando:

```
rpcgen -a <nombre_archivo>.x
```

Para el propósito de la práctica, se tendrá dos archivos a modificar, por un lado el *<nombre_archivo>_client.c* y del otro el *<nombre_archivo>_server.c*. Este último contiene los procesos que fueron declarados en la estructura .x y que el cliente tiene a disposición para realizar la comunicación. Cuando se considera que la implementación está terminada, se compila los archivos mediante el comando make:

```
make -f Makefile.<nombre_archivo>
```

Finalmente, para usarlo, se deben abrir dos terminales que ejecuten por un lado *./<nombre_archivo>_client <parametros>* y por otro el servidor, *./<nombre_archivo>_server*. Para evitar problemas al momento de la ejecución, es recomendable tener en superusuario las terminales o acompañar los comandos explicados con la opción de *sudo*.



Respuesta 1

A partir de la comunicación de RPC, se debe ejecutar una serie de ejemplos dados por la cátedra y responder los incisos.

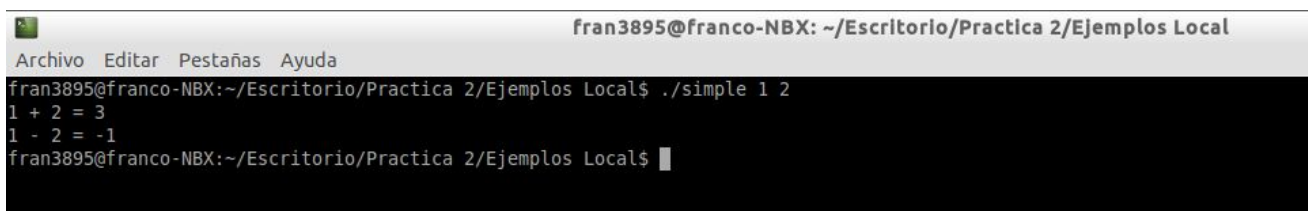
Inciso -a

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

Para este inciso, se realizaron cambios generales en todos los procedimientos. Éstos fueron:

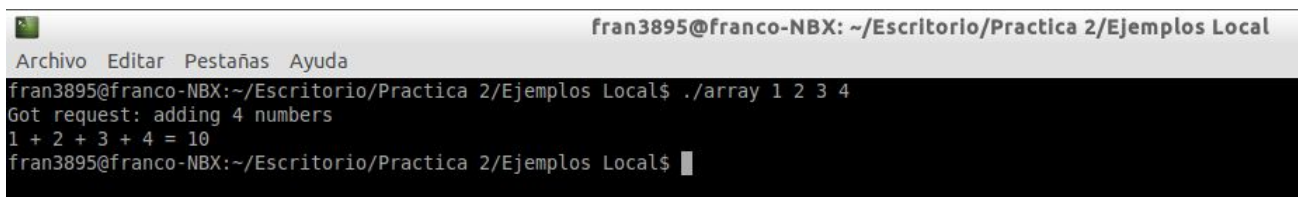
- Ya no hay dos procedimientos separados (cliente y servidor), sino que en el procedimiento que era cliente originalmente ahora existe una función que cumple la función que antes realizaban los procedimientos remotos del servidor.
- Al realizar el llamado al procedimiento, no hace falta pasar como parámetro el host.
- No se creará un tipo cliente, por lo que no hará falta chequear si se generó correctamente, así como ya no será necesario enviarlo como parámetro a las funciones del cliente.
- No es necesario tener una estructura en la que se carguen los parámetros a enviar a los procedimientos remotos.

Las siguientes imágenes ilustran los resultados de la ejecución de cada uno de los procedimientos de forma local. Los resultados podrán compararse con los mostrados en el inciso siguiente:



```
fran3895@franco-NBX: ~/Escritorio/Practica 2/Ejemplos Local
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$ ./simple 1 2
1 + 2 = 3
1 - 2 = -1
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$
```

Figura 1.1: Simple local



```
fran3895@franco-NBX: ~/Escritorio/Practica 2/Ejemplos Local
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$ ./array 1 2 3 4
Got request: adding 4 numbers
1 + 2 + 3 + 4 = 10
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$
```

Figura 1.2: Array local



```
fran3895@franco-NBX: ~/Escritorio/Practica 2/Ejemplos Local
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$ ./u1 1
UID 1, Name is daemon
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$ ./u1 daemon
Name daemon, UID is 1
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$
```

Figura 1.3: U1 local

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/Ejemplos Local
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$ ./list 1 2 3 4
1 2 3 4
Sum is 10
fran3895@franco-NBX:~/Escritorio/Practica 2/Ejemplos Local$
```

Figura 1.4: List local

Inciso -b

Ejecutar los procesos y mostrar la salida obtenida (del “cliente” y del “servidor”) en cada uno de los casos.

Nota: Al ejecutar cada uno de los procesos clientes de los ejemplos provistos por la cátedra, además de los parámetros específicos de cada uno se debe incluir entre ellos el host, que en este caso en particular será *localhost*.

Al ejecutar el proceso “Simple”, se pasan como parámetros del cliente dos números enteros. El servidor calculará la suma y resta entre ellos y las devolverá como salida:

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/1-simple_V2
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/1-simple_V2$ ./simp_client localhost 1 2
1 + 2 = 3
1 - 2 = -1
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/1-simple_V2$
```

Figura 1.5: Simple cliente

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/1-simple_V2
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/1-simple_V2$ ./simp_server
Got request: adding 1, 2
Got request: subtracting 1, 2
```

Figura 1.6: Simple servidor

Al ejecutar el proceso “Array”, se pasan como parámetros del cliente una cantidad *n* de números enteros. El servidor calculará la suma total y la devolverá como salida:



```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/3-array
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/3-array$ ./vadd_client localhost 1 2 3 4
1 + 2 + 3 + 4 = 10
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/3-array$
```

Figura 1.7 Array cliente

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/3-array
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/3-array$ ./vadd_server
Got request: adding 4 numbers
```

Figura 1.8: Array servidor

Al ejecutar el proceso “U1”, se pasan como parámetros del cliente o bien un número entre 0 y 9, que será el UID de un proceso, o bien un nombre de proceso. En el primer caso, el servidor devolverá como salida el nombre del proceso asociado a dicho UID. En el segundo caso, el servidor devolverá como salida el UID del proceso especificado.

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/2-u1
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/2-u1$ ./userlookup_client localhost 1
UID 1, Name is daemon
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/2-u1$ ./userlookup_client localhost daemon
Name daemon, UID is 1
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/2-u1$
```

Figura 1.9: U1 cliente

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/2-u1
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/2-u1$ ./userlookup_server
```

Figura 1.10: U1 servidor

Al ejecutar el proceso “List”, se pasan como parámetros del cliente una cantidad n de números enteros. El cliente agregará cada uno a una lista enlazada y los imprimirá en pantalla. El servidor calculará la suma entre ellos y la devolverá como salida:

```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/4-list
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/4-list$ ./ll_client localhost 1 2 3 4
1 2 3 4
Sum is 10
fran3895@franco-NBX:~/Escritorio/Practica 2/ejemplosRPC/4-list$
```

Figura 1.11: List cliente



```
fran3895@franco-NBX: ~/Escritorio/Practica 2/ejemplosRPC/4-list$ ./ll_server
```

Figura 1.12: List servidor

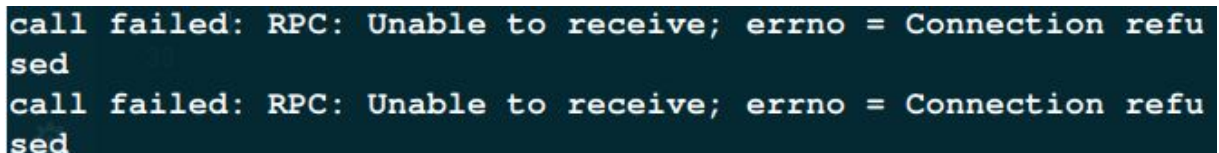
Inciso -c

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.

Los experimentos realizados en la práctica consisten en realizar cambios en el código, colocando el comando `sleep(tiempo_espera)`; y `exit(1)`; Para romper el cliente, lo que se realizó fue (tanto en conexión UDP como TCP) colocar un sleep de 15 segundos, el cual permite romper la conexión con con servidor.

De la misma forma, comentando el sleep del cliente, se añadió un exit evaluado en 1 del lado del servidor, el cual permite terminar la ejecución del mismo, haciendo que se produzca la rotura con el cliente.

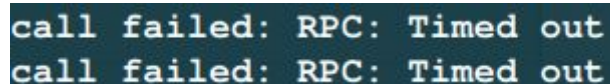
En todos los casos descritos con anterioridad, se tuvo el mismo error, y se puede observar en la **figura 1.13**.



```
call failed: RPC: Unable to receive; errno = Connection refused
call failed: RPC: Unable to receive; errno = Connection refused
```

Figura 1.13: Error en el servidor/cliente

Por último, lo que se trató de observar fue el *time out* que se producía en el caso que el servidor se quede ejecutando un sleep de 90 segundos, permitiendo así dejar al cliente en un modo de espera. Tal es así, que en la terminal del cliente se pudo observar el siguiente mensaje:



```
call failed: RPC: Timed out
call failed: RPC: Timed out
```

Figura 1.14: Error producido por el Servidor al cliente

Respuesta 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.

Opción -N: Utiliza el nuevo estilo de rpcgen. Esto permite que los procedimientos tengan múltiples argumentos como parámetros si se desea. A su vez, implementa un estilo de pasar de parámetros que se asemeja mucho al lenguaje C. Por lo tanto, al pasar un argumento a un procedimiento remoto, no tiene que referenciar a un puntero, sino al propio argumento. Este comportamiento es diferente al estilo antiguo del código generado por rpcgen. Este modo no se utiliza de forma predeterminada debido a los problemas de implementación que se podrían generar.

Opción -M: Generar estados de multithread para pasar argumentos y resultados entre código generado por rpcgen y código escrito por el usuario. Esta opción es útil para los que deseen utilizar subprocesos en su código.

Opción -A: Genera el código de RPC en modo MT Auto. Éste modo permite a los servidores RPC utilizar threads Solaris para procesar peticiones de los clientes de forma concurrente. Esta opción activa la opción -M automáticamente, así que no necesita ser especificada explícitamente. La opción -M es necesaria porque cualquier código generado debe ser seguro para multithreading.

Para poder contar con procesamiento concurrente, se debería disponer de algún mecanismo de control para evitar que más de un cliente a la vez acceda a variables compartidas o a los procedimientos del servidor, dado que en dicho caso se podrían producir errores en el manejo de los datos.

Respuesta 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.

Cuándo se utiliza RPC, no existe el pasaje de parámetros por referencia tradicional, dado que las direcciones de memoria en un proceso no tienen sentido en otro (a no ser que compartan memoria entre ellos). Si se necesita que un procedimiento remoto realice cambios sobre datos de un procedimiento local, se debe enviar una copia de los datos a



ellos. Luego de que se realice el procesamiento de dichos datos, el procedimiento remoto enviará una copia de los datos con los cambios que se hayan realizado, y el procedimiento que realizó la petición originalmente deberá modificar los datos de forma local.

Respuesta 4

Se debe realizar un servidor de archivos (una versión restringida), en el cual la lectura contenga las siguientes características:

- ***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.***

Inciso -a

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

Para implementar el servidor de archivos, se consideraron algunas decisiones que permitieron darle más facilidad para la implementarla y respetar las consignas de la práctica:

- El servidor se realiza de forma local, por lo que en el momento de ejecutar el cliente, se debe definir el parámetro de host como “localhost”.
- La conexión utiliza el protocolo de UDP. En el caso de querer implementarlo con TCP no producirá ningún tipo de cambio.
- Para trabajar con los archivos, se utilizó las propiedades del lenguaje C para poder abrirlos, copiar, pegar y cerrarlos. Además, se debe tener en cuenta que se trabajara con los archivos con el formato de *binario* pudiendo copiar íntegramente los archivos y su contenido
- Se tendrá implementado los procesos de *LECTURA* y *ESCRITURA*. El primero leerá una cadena de bytes desde un archivo y retornará además la cantidad de bytes leídos. El segundo, dado un nombre de archivo, lo creará o abrirá en el servidor y escribirá en él una cadena de bytes que se enviará como parámetro.



Inciso -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 ítem anterior, sin cambios específicos del servidor para este ítem 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.

Carpeta sin título Para realizar dicho inciso, vamos a considerar 3 archivos, el cual tendrá el mismo nombre pero distinta extensión, con el fin de realizar las pruebas de la práctica. Por eso definimos:

- *prueba.txt*
- *prueba.png*
- *prueba.pdf*

Cabe destacar que para realizar la simulación, se considera que los archivos creados y el original se encontraran en el mismo directorio que la implementación del RPC.

Desde el lado del cliente, al llamar al procedimiento se especifica el nombre del archivo a utilizar. En una estructura se almacenarán dicho nombre junto con la posición a partir de la cual se leerá en el archivo y la cantidad de bytes a leer. Luego de realizarse la primera lectura, se creará un archivo local en el que se escribirá la cadena de bytes devuelta por el servidor y se generará una copia “remota” que se almacenará en el sistema de archivos del servidor, utilizando la operación de escritura del servidor a la que se enviará la misma cadena de bytes. Este proceso continuará hasta que no queden bytes por leer en el archivo original.

En la **Figura 2.1** se podrá observar los archivos que se crean al realizar la escritura y lectura de parte del servidor y del cliente, implementado de forma local, considerando el entorno como el mismo servidor de archivos. En la parte inferior tendremos los 3 archivos originales y en la superior, las copias que crean el cliente y el servidor.

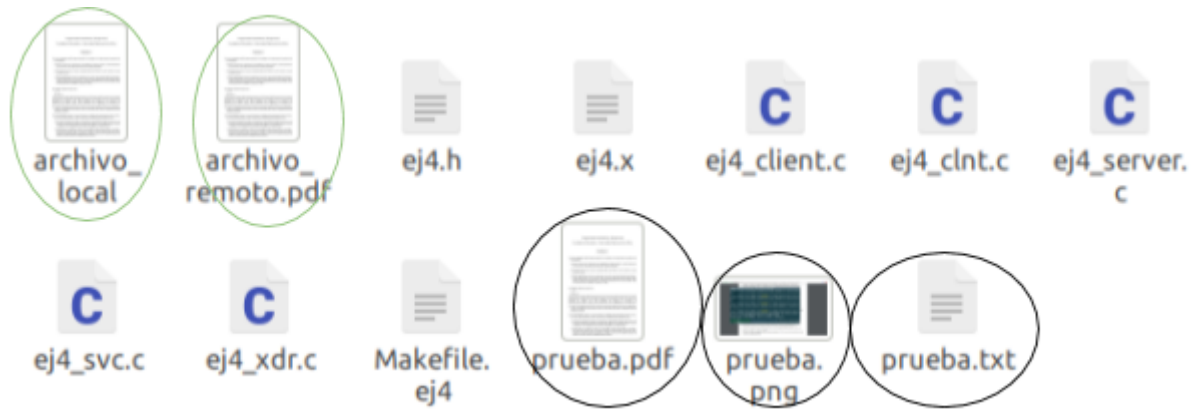


Figura 2.1: archivos para la Lectura/Escritura

Las siguientes imágenes ilustran el llamado desde consola a los procedimientos Cliente y Servidor y su respuesta al realizarse el copiado del archivo “prueba.txt”. Además, se muestra la salida del comando diff y la comparación entre el archivo original, la copia local y la copia remota, indicando que no existen diferencias entre ellos:

```
fran3895@franco-NBX: ~/Escritorio/FilesystemV2
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/FilesystemV2$ sudo ./filesystemv2_client localhost "prueba.txt"
fran3895@franco-NBX:~/Escritorio/FilesystemV2$ diff prueba.txt Archivo_local
fran3895@franco-NBX:~/Escritorio/FilesystemV2$ diff prueba.txt Archivo_remoto
fran3895@franco-NBX:~/Escritorio/FilesystemV2$
```

Figura 2.2: Cliente con el archivo de texto

```
fran3895@franco-NBX: ~/Escritorio/FilesystemV2
Archivo Editar Pestañas Ayuda
fran3895@franco-NBX:~/Escritorio/FilesystemV2$ sudo ./filesystemv2_server
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
Archivo a abrir para escritura: Archivo_remoto
Archivo para escritura abierto
Escribiendo archivo...
```

Figura 2.3: Servidor con el archivo de texto

Asimismo, se puede comprobar que el funcionamiento no cambia para otros tipos de archivo, por ejemplo, con un PDF:



Figura 2.6: Cliente con archivo .png

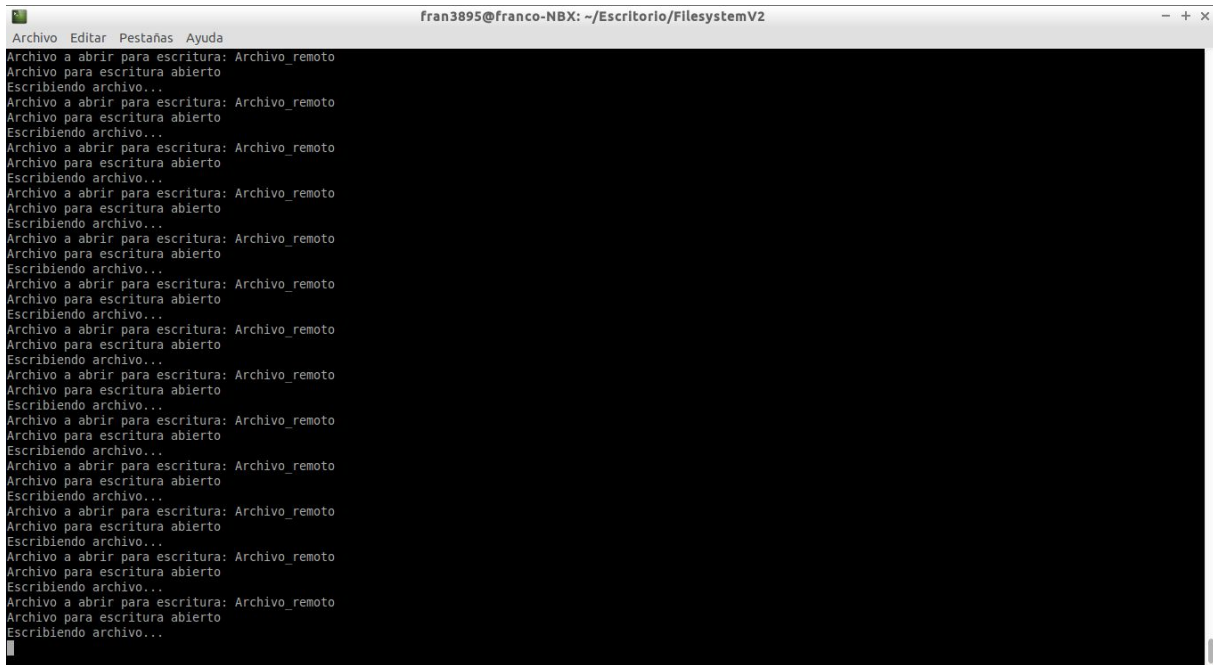


Figura 2.7: Servidor con archivo .png

Respuesta 5

Inciso -a

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

Para desarrollar este experimento, se implementó un conteo de tiempo del lado del servidor y por parte del cliente, dando como resultado lo siguiente:

```
Resultado de inicio cliente: 0.003704
Resultado servidor:1.000000
Resultado final: 1.003704
```

Figura 3.1: Resultados finales en el cliente

Por otro lado, para que lo anterior se visualizara con algo de precisión, se optó por colocar un sleep del lado del servidor, con valor a un segundo, permitiendo así tener el tiempo final observado en la **figura 3.1**.

Inciso -b

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

Realizando las mismas pruebas 10 veces, se obtuvo el siguiente cuadro de tiempo:



Prueba	Tiempo Final en cliente
1	1.003420
2	1.005049
3	1.004880
4	1.004132
5	1.003991
6	1.003314
7	1.004130
8	1.004280
9	1.004871
10	1.004187

Promedio Final = 1,003842 segundos

Para reducir el promedio de tiempo encontrado, al tiempo total del sleep (que contiene valor 1 segundo) se lo redujo en un 10%, haciendo que todos los valores anteriores se reduzcan en dicho porcentaje y obteniendo así el valor promedio deseado. Realizando las pruebas tenemos la siguiente tabla:

Pruebas	Tiempos final en cliente (10% menos)
1	0.904682
2	0.903990
3	0.904266
4	0.904301
5	0.904058
6	0.904738
7	0.904983
8	0.903617
9	0.904761



10	0.903676
----	----------

Promedio Final = 0.9037 segundos

Se pudo observar que el tiempo bajo aproximadamente un 10% en promedio, como indica la consigna.

Inciso -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.

Para realizar este inciso, se plantea que en el servidor se tenga una variable denominada *tiempoExtra* que contenga la suma de los valores dispuestos por el cliente (ingresados como uno de los parámetros del programa). De esta forma, siempre el servidor va a tardar en ejecutarse el tiempo anterior más el actual. La **figura 3.2** describe lo planteado:

```
alan@alan-Lenovo-G475:~/Documentos/PDTR/P2/ej5_(completo)$ s
udo ./superadorTiempo_client localhost 2
Resultado de inicio cliente: 0.005146
Resultado servidor:1.993218
Resultado final: 1.988072
alan@alan-Lenovo-G475:~/Documentos/PDTR/P2/ej5_(completo)$ s
udo ./superadorTiempo_client localhost 2
Resultado de inicio cliente: 0.004997
Resultado servidor:3.992770
Resultado final: 3.987773

alan@alan-Lenovo-G475:~/Documentos/PDTR/P2/ej5_(completo)$ s
udo ./superadorTiempo_server
El tiempo de espera en servidor: 2.000000
t_ini_servidor: 0.006782
El resultado del servidor en segundos: 1.993218
-----
El tiempo de espera en servidor: 4.000000
t_ini_servidor: 0.007230
El resultado del servidor en segundos: 3.992770
-----
```

Figura 3.2: Cliente con ingreso 2 y Servidor respondiendo con sleep();