Programación Distribuida y Tiempo Real

Práctica 2

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

1.
Para volverlo local se movieron los métodos add_1_svc y sub_1_svc del servidor al código del cliente, a estos se le eliminó la parte _svc del nombre para que funcionen. Se eliminó todo lo relacionado a la conexión en el código del cliente. Se retorna el valor de los métodos movidos sin comprobar que se haya podido alcanzar el procedimiento remoto, ya que no es más remoto. Se importaron las librerías necesarias (stdlib.h).
También se cambiaron la cantidad de inputs por consola, se eliminó el input de url en el cliente.
2.
Para volverlo local se movieron los métodos byname_1_svc y bynum_1_svc del servidor al código del cliente, a estos se le eliminó la parte _svc del nombre para que funcionen. Se eliminó todo lo relacionado a la conexión en el código del cliente. Se retorna el valor de los métodos movidos sin comprobar que se haya podido alcanzar el procedimiento remoto, ya que no es más remoto. Se importaron las librerías necesarias (pwd.h, stdlib.h, sys/types.h).
También se cambiaron la cantidad de inputs por consola, se eliminó el input de url en el cliente.
3.
Para volverlo local se movió el método vadd_1_svc del servidor al código del cliente, se le eliminó la parte _svc del nombre para que funcionen. Se eliminó todo lo relacionado a la conexión en el código del cliente. Se retorna el valor de los métodos movidos sin comprobar que se haya podido alcanzar el procedimiento remoto, ya que no es más remoto.
También se cambiaron la cantidad de inputs por consola, se eliminó el input de url en el cliente.
4.
Para volverlo local se movió el método sum_1_svc del servidor al código del cliente, se le eliminó la parte _svc del nombre para que funcionen. Se eliminó todo lo relacionado a la conexión en el código del cliente. Se retorna el valor de los métodos movidos sin comprobar que se haya podido alcanzar el procedimiento remoto, ya que no es más remoto.
También se cambiaron la cantidad de inputs por consola, se eliminó el input de url en el cliente.

En todos los casos se mantuvo el tipo definido para pasar datos por rpc, aunque, al realizar el procedimiento de manera local, se pueden pasar los parámetros de forma normal.

También sería posible combinar los métodos remotos con los métodos locales que construyen estos tipos.

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

1.
Servidor:
./server
Got request: adding 1, 2
Got request: subtracting 1, 2

Cliente:
./client localhost 1 2
1 + 2 = 3
1 - 2 = -1

2.
Servidor:
./server

Cliente:
./client localhost 1000
UID 1000, Name is okason

3.
Servidor:
./vadd_service
Got request: adding 2 numbers
Got request: adding 3 numbers

Cliente:
./vadd_client localhost 1 2
1 + 2 = 3
./vadd_client localhost 1 2 3
1 + 2 + 3 = 6

4.
Servidor:
./server
Cliente:
./client localhost 1 2
1 2
Sum is 3

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 comunicacio nes o no haya receptor para las comunicaciones. Verifique con UDP y con TCP.

1.
UDP
-exit() en servidor, genera “Trouble calling remote procedure” en cliente luego de un delay.
-sleep(30) en servidor, genera “Trouble calling remote procedure” en cliente luego de un delay.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), Trouble calling remote procedure

TCP
-exit() en servidor, genera “Trouble calling remote procedure” instantáneamente luego de interrumpir al servidor.
-sleep(30) en servidor, genera “Trouble calling remote procedure” en cliente luego de un delay.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “localhost: RPC: Remote system error - Connection refused”

2.
UDP
-exit() en servidor, call failed:: RPC: Unable to receive; errno = Connection refused luego de un delay.
-sleep(30) en servidor, “call failed:: RPC: Timed out Segmentation fault (core dumped)” luego de un tiempo.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “call failed:: RPC: Unable to receive; errno = Connection refused
Segmentation fault (core dumped)”

TCP
-exit() en servidor, genera “call failed:: RPC: Unable to receive; errno = Connection reset by peer
Segmentation fault (core dumped)” instantáneamente luego de interrumpir al servidor.

-sleep(30) en servidor, “call failed:: RPC: Timed out Segmentation fault (core dumped)” luego de un tiempo.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “localhost: RPC: Remote system error - Connection refused”

3.
UDP
-exit() en servidor, genera “Trouble calling remote procedure” en cliente luego de un delay.
-sleep(30) en servidor, “Trouble calling remote procedure” luego de un tiempo.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), Trouble calling remote procedure

TCP
-exit() en servidor, genera “Trouble calling remote procedure” instantáneamente luego de interrumpir al servidor.
-sleep(30) en servidor, “Trouble calling remote procedure” luego de un tiempo.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “localhost: RPC: Remote system error - Connection refused”

4.
UDP
-exit() en servidor, call failed:: RPC: Unable to receive; errno = Connection refused luego de un delay.
-sleep(30) en servidor, call failed: RPC: Timed out.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “call failed:: RPC: Unable to receive; errno = Connection refused”

TCP
-exit() en servidor, genera “call failed:: RPC: Unable to receive; errno = Connection reset by peer” instantáneamente luego de interrumpir al servidor.
-sleep(30) en servidor, call failed: RPC: Timed out.
-El servidor no tiene a quien retornar la llamada, no se genera error en este, sigue recibiendo llamadas sin problemas.
-Si el servidor no da respuesta (no se corre), “localhost: RPC: Remote system error - Connection refused”

2)
Describir/analizar las opciones
a)

-N

Usa el newstyle de rpcgen. Lo que permite que los procesos recivan multiples argumentos. Tambien usa un estilo de pasaje de parametros similar al de C. Entonces, cuando se pasa un argumento a un procedimiento remoto, no es necesario pasar un puntero al argumento, sino el mismo argumento. Este comportamiento es diferente del oldstyle del codigo generado por rpcgen. El default es el oldstyle por compatibilidad con versiones anteriores.

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/rpcgepguide-1939/index.html

Por default, el codigo generado por rpcgen no es multithreading safe. Usa variables globales no protegidas y retorna resultados en forma de variables estaticas.
Utilizar -M permite generar codigo MT-safe para su utilizacion en un ambiente de multithread.

-A permite utilizar MT-auto mode el cual deja que los servidores RPC usen automaticamente threads de Solaris para procesar request de los clientes concurrentemente. -A tambien activa -M, por lo que no es necesario especificarlo si esta activado -A.

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.

RPC se encarga de la codificacion y decodificacion de los parámetros y resultados en salida y entrada.
El pasaje de parámetros es por valor, RPC no soporta pasaje por referencia.
Es necesaria la utilización de estructuras para poder enviar múltiples datos a través de parámetros a procedimientos remotos. Esto se ve en todos los ejemplos dados, por ejemplo, en el primer ejemplo se usa la struct operands para llevar los 2 operandos.
Por lo tanto, si solo se intenta pasar un parámetro, la transparencia con respecto a estos es muy buena, ya que se puede llamar de la misma forma, pero agregando un dato tipo CLIENT. En cambio, si se desean pasar múltiples parámetros será necesario definir una estructura en el archivo .x .
Los valores de retorno funcionan de forma similar, en caso de ser solo 1 valor, se puede pasar normalmente, en caso de querer retornar más de un valor, será necesaria la utilización de una estructura.

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.**

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

Se crearon los archivos rpcfs.x y rpcfsservice.c.
En rpcfs.x se crean las estructuras necesarias para pasar los parametros al read y write, y para devolver los bytes leidos y la cantidad de bytes leídos en el read. Aquí también se definen los nombres y parámetros de los procedimientos remotos.
En rpcfsservice.c se crearon los 2 métodos:

- read_1_svc:
    - retorna tipo read_result (con el buffer y la cantidad de bytes leídos) definido en rpcfs.x y recibe los parámetros read_data *argp (con el nombre del archivo, la posición y la cantidad de bytes a leer) y struct svc_req *rqstp.
    - Se define el resultado y se asigna al buffer de retorno el tamaño definido en el parámetro.
    - Se abre el archivo con el nombre recibido.
    - Finalmente se lee usando pread() y se retorna el resultado.
- write_1_svc:
    - retorna tipo int la cantidad de bytes leídos. Recibe los parámetros write_data *argp (con el buffer, nombre del archivo y cantidad de bytes) y struct svc_req *rqstp.
    - Se abre o crea el archivo con el nombre recibido.
    - Se define el resultado y se escriben los datos con write() guardando el resultado de esta operación (bytes escritos) en el resultado.
    - Finalmente se retorna el resultado.

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.

Para implementar el cliente, primero creo las variables, entre ellas el cliente tipo CLIENT que usare para conectarme al servidor. Recibire como parametros por consola el nombre del archivo original, el nombre de la copia, la cantidad de datos a leer y el offset.
Una vez inicializadas las variables con los datos recibidos, llamo a la funcion server_read, la cual tomara los datos y los reunira en una estructura read_data, la cual sera utilizada en el procedimiento remoto read_1.
Una vez comprobado que el resultado fue recibido con exito, se retorna la estructura read_result, con el buffer con los datos leidos y la cantidad de datos leidos. Esto sera utilizado para escribir mediante write un archivo en el cliente con los datos leidos con el mismo nombre.
Finalmente se llamara a la funcion server_write con el nombre del archivo, la cantidad de caracteres a escribir y los caracteres a escribir, los cuales fueron recibidos anteriormente por server_read. En esta funcion se reuniran los parametros en la estructura write_data para ser enviados al procedimiento remoto write_1, el cual creara la copia en el servidor.

5) Timeouts en RPC:
a.-Desarrollar un experimento que muestre el timeout definido para las llamadas RPC y el promedio de tiempo de una llamada RPC.

El tiempo de timeout es de aproximadamente 25 segundos.
El tiempo promedio para una llamada es 0.000182325. Esto fue calculado con el ejemplo 1.

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 asignar un timeout de 22 segundos (25*0.9) utilizando clnt_control con la opcion CLSET_TIMEOUT y un timeval con tv_sec = 25*.09, efectivamente, el timeout cambio a 22 segundos.
Probado con el ejemplo 1:
22.002513
22.002071
22.002550
22.002727
22.002774
22.002668
22.002743
22.002695
22.003038
22.001886
Se pueden ver que son 22 segundos + el tiempo promedio de llamada aproximadamente.

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 desarrollar el cliente utilicé 2 procedimientos remotos:

- connect: llama a un procedimiento remoto del servidor que no hace nada, de esta forma obtengo una conexión al servidor y con esta consigo el timeout por defecto utilizando clnt_control(clnt, CLGET_TIMEOUT, &tv).
- timeout: este procedimiento remoto recibe un long int con el tiempo de timeout del cliente y utiliza este tiempo + 1 con sleep para esperar a que se pase este tiempo.