Llamadas a procedimientos remotos

Arquitectura y Tecnología de Computadores

Práctica 2

1. Servidor de ficheros

Se programará un servidor que de acceso a los ficheros existentes en cierta carpeta, de modo que un cliente pueda obtener la lista de ficheros existentes, abrirlos, cerrarlos, leerlos, modificarlos, o borrarlos. El cliente también podrá crear ficheros nuevos en esa carpeta.

Para ello, el servidor proporciona una serie de funciones que pueden ser invocadas desde el cliente usando el mecanismo RPC de Sun (programado con ayuda de la herramienta `rpcgen`). El interfaz del servidor se proporciona en un apéndice al final de este documento (y también está disponible para descargar en la página web de la asignatura). A continuación se describe el cometido de cada función:

directorio No recibe parámetros y devuelve una lista con los nombres de fichero que el cliente puede intentar abrir o borrar.

borrar Recibe como parámetro un nombre de fichero y lo borra de la carpeta. Retorna un error en caso de que no sea posible borrarlo, o nada en caso de éxito.

abrir Recibe como parámetro el nombre de fichero a abrir y el modo de apertura, y retorna un manejador (entero) para el fichero o bien un error si no es posible abrirlo.

cerrar Recibe como parámetro el manejador de fichero y lo cierra. No retorna nada.

tamanio Recibe como parámetro el manejador de fichero y devuelve cuál es el tamaño del fichero en bytes, o un error si no es posible averiguarlo.

leer Recibe el manejador de fichero del que se quiere leer, el número de bytes que se quieren leer y la posición del fichero del primer byte a leer. Retorna un buffer con los bytes que han sido leídos, o un error si no es posible la lectura.

escribir Recibe el manejador de fichero en que se quiere escribir, un buffer con los bytes que se quieren escribir, y la posición dentro del fichero en que debe comenzar la escritura. Retorna el número de bytes que se han escrito, o un error si no es posible la escritura.

acabar Este servicio desregistra el programa y finaliza.

En todos los casos anteriores, cuando la función debe retornar un error, lo que retornará será una cadena de texto que contiene un mensaje indicando cuál ha sido el problema (el mismo mensaje que generaría `perror()` en la máquina del servidor).

1.1. Detalles de implementación

El número de programa asociado al servidor será una cantidad hexadecimal de 8 cifras, cuya primera cifra sea 2, y las restantes 7 sean los últimos dígitos del DNI del alumno/a.

En la carpeta en que se ejecute el servidor, habrá un subdirectorio llamado `ficheros`, dentro del cual estarán los ficheros accesibles a los clientes. El servicio `directorio`, listará este subdirectorio. Los servicios `abrir` y `borrar`, comprobará que no se intenta acceder a un fichero fuera de este subdirectorio (rechazarán nombres de fichero que contengan el carácter "`/`", puesto que se trataría de accesos a otras rutas).

Para la implementación de los servicios se usarán las siguientes funciones del sistema operativo UNIX. Para el acceso al directorio y la obtención de los nombres de fichero que hay en él, deben usarse las funciones `opendir()`, `readdir()` y `closedir()`. Para borrar un fichero debe usarse la función `unlink()`. Para mover el puntero de lectura/escritura dentro de un fichero debe usarse la función `fseek()`. Ten en cuenta que `fseek()` puede dar un error si se intenta mover a una posición no válida. Para averiguar el tamaño de un fichero puedes usar la función `fstat()` (preferiblemente), o bien mover el puntero de lectura al final del fichero y usar `ftell()` para averiguar dicha posición[1]

Cuando el servidor abra un fichero con `open()`, debe especificar como segundo parámetro el modo adecuado, según lo que el cliente haya solicitado. `open()` puede llevar además un tercer parámetro opcional, para especificar los permisos con los que el fichero será creado (permisos que afectan a la posibilidad de ser borrado, o de ser abierto más tarde, una vez cerrado, para otras operaciones). Se debe especificar en este tercer parámetro el valor `S_IREAD|S_IWRITE`, que garantiza que el servidor pueda volver a abrir ese fichero, sobreescribirlo, o borrarlo, si un cliente se lo pide.

Para retornar un mensaje de error, se recomienda usar la función `strdup()` que duplica la cadena que recibe como argumento y retorna un puntero a la versión duplicada (la dirección a que apunta la obtiene la propia función llamando a `malloc()`). Si el error ha sido causado por una de las funciones de la librería estándar (ej. si la función retorna -1), se puede invocar `strerror(errno)`, para obtener un puntero a una cadena que contiene el texto del mensaje de error[2].

Algunos de los servicios requerirán que el servidor reserve memoria para los resultados (por ejemplo, el servicio `directorio` deberá crear espacio para los nombres de fichero[3]). Este espacio creado debe ser liberado, sin embargo no se puede liberar antes de que el servicio retorne (pues en ese caso ¿qué retornaría?), pero después de que el servicio haya retornado ¿cómo liberarlo?. La respuesta es que el espacio reservado en una llamada se liberará en la llamada siguiente. El servicio debe comprobar si ya había sido llamado (una variable estática puede mantener esta información) y en ese caso liberar la memoria reservada por la llamada previa. Para liberar esta memoria la opción más simple es usar `xdr_free()`.

1.1.1. Detalles adicionales para la máquina sirio

En esta máquina debe ponerse cuidado en que el compilador sea `gcc` en lugar de `cc` (que es el que usa por defecto la utilidad `make`). Recuerda además enlazar con las bibliotecas apropiadas.

Además, el servidor generado por `rpcgen`, por defecto, se desconecta de la terminal una vez lanzado, de modo que queda ejecutándose como tarea de fondo. Todos los mensajes que el servidor generase en la salida estándar o salida de error, son redireccionados automáticamente hacia `/dev/null` (esto es, son eliminados). En estas condiciones no se puede utilizar `printf` para depurar el servidor. Si quieres evitar este comportamiento, al compilar el servidor (concretamente al compilar el fichero `interfaz_svc.c`) debes especificar la opción `-DRPC_SVC_FG`. En este caso el servidor creado se comportará como en Linux, esto es, no se desconecta de la terminal y sus mensajes serán visibles.

2. Cliente para el servidor anterior

Para probar el servidor desarrollado en la sección previa, se escribirá un cliente que implemente una especie de “FTP”. El cliente se lanzará especificando en la línea de comandos la máquina en que está el servidor y, opcionalmente, su número de programa. Si el número de programa no se especifica, se usará el declarado en el interfaz. En caso de que se especifique, se asumirá que está especificado en hexadecimal. Una vez arrancado el cliente, éste inicializará la estructura CLIENT apropiada. Si encontrara algún error, informará del mismo y terminará. En caso contrario, se entrará en un interfaz de comandos que debe entender las siguientes ordenes:

PROBAR El cliente llamará al servicio número cero del servidor (usando el mecanismo RPC simplificado) para comprobar si está vivo, e informará al usuario del resultado.

[1] Esta segunda opción tiene una implementación más compleja, ya que además deberías asegurarte de dejar el puntero de lectura en su posición original, para no interferir con llamadas a `read`

[2] No se puede retornar directamente el puntero a esta cadena, debe antes duplicarse con `strdup()` y retornar el duplicado

[3] La mayoría de los restantes servicios también reservan memoria cuando crean cadenas de texto con `strdup()`

DIR El cliente mostrará la lista de ficheros en el servidor que el usuario puede manipular.

DEL seguido de un nombre de fichero, intentará el borrado de ese fichero.

TRAER seguido de un nombre de fichero, obtendrá una copia del fichero del servidor y lo almacenará con el mismo nombre en la máquina cliente.

LLEVAR seguido de un nombre de fichero, tomará el fichero especificado de la máquina cliente y creará en la máquina servidora una copia del mismo, con el mismo nombre.

PRINCIPIO seguido de un nombre de fichero y de un entero N, mostrará en pantalla los N primeros caracteres del fichero. Si N es mayor que la longitud del fichero, mostrará el fichero completo. Si se omite el entero N, se asumirá un valor por defecto de 80.

FINAL seguido de un nombre de fichero y de un entero N, mostrará en pantalla los N últimos caracteres del fichero. Si N es mayor que la longitud del fichero, mostrará el fichero completo. Si se omite el entero N, se asumirá un valor por defecto de 80.

QUIT finaliza el cliente.

Observar que para implementar estas funcionalidades, puede ser necesario invocar varios servicios en secuencia. Si el servidor envía un error en cualquiera de estos servicios, el mensaje de error debe mostrarse al usuario, y el cliente abortará el comando en curso (asegurándose de no dejar al servidor ni al cliente en un estado incorrecto, esto es, por ejemplo, debe cerrar ficheros que hubiera abierto, etc..)

2.1. Detalles de implementación

Los ficheros que el cliente cree localmente, como consecuencia de la operación `TRAER`, deben tener los permisos de lectura y escritura, para lo cual deben abrirse especificando el tercer parámetro opcional, como se ha descrito para el caso del servidor.

La transferencia de un fichero mediante `TRAER` o `LLEVAR`, se hará en bloques de 1024 bytes como máximo. El cliente por tanto debe invocar repetidamente al servicio `leer` del servidor (o `escribir` si se trata de un envío de fichero), transmitiendo 1024 bytes de cada vez y actualizando la posición de lectura o escritura remota. La última transferencia será de 0 bytes, puesto que este es el valor devuelto por `read()` al alcanzar el fin de fichero. En las transferencias para los comandos `PRINCIPIO` o `FINAL` no se impone esta limitación. El número de bytes solicitado por el usuario pueden transferirse en una sola invocación remota. Recordar que el cliente debe reservar memoria, o dejar que el filtro XDR la reserve por él, y en todo caso debe liberarla una vez ya no es necesaria, mediante una llamada a `free()`, o `xdr_free()`.

3. Cliente finalizador

Escribe un cliente (`finalizador`) que reciba desde línea de comandos el nombre de una máquina y finalice tu servidor de ficheros que esté corriendo en esa máquina.

El programa intentará localizar al servidor (`clnt_create()`). Si fracasa informará al usuario de que el programa no está registrado. Si lo localiza con éxito, invocará su servicio número cero, mediante el mecanismo RPC simplificado, para verificar si el proceso está vivo. Si encuentra un error, informará al usuario de que el proceso no responde, aunque esté registrado[4]. Finalmente, si el proceso responde, el cliente invocará su servicio `acabar()` con lo que el servidor se desregistrará y terminará.

Utiliza este cliente para desregistrar y finalizar tu servidor una vez hayas terminado de hacer pruebas con el mismo. No dejes servidores corriendo, ni servicios registrados en el localizador.

[4]En este caso, el localizador contiene información obsoleta, fruto de una terminación incorrecta del servidor. Esta información debería ser borrada del localizador, para lo que debe usarse el programa `rpcinfo` con la opción `-d`. Por desgracia, en algunos sistemas operativos sólo el administrador puede invocar esta opción. Otra forma de eliminar la información incorrecta del localizador consiste en lanzar de nuevo el servidor (con lo que el localizador actualizará su información cuando éste se registre) y seguidamente finalizarlo de forma correcta, esto es, usando el cliente finalizador)

NOTA: El servicio `acabar()` no retorna al cliente ningún resultado, por lo que el cliente recibirá un error de RPC. Este error puede ignorarse.

Anexos

A. Interfaz

A continuación se muestra el listado de `interfaz.x`, a partir del cual debes generar los *stubs* de cliente y servidor, y en el que debes basarte para implementar los servicios.

```
typedef int descriptor;
typedef string nombrefich<>;
typedef nombrefich listado<>;
typedef string error<>;
typedef opaque trozo<>;

enum modos_abrir {
  MODO_LECTURA    = 1,
  MODO_ESCRITURA  = 2,
  MODO_AMBOS      = 3,
  MODO_CREAR      = 4
};

union ret_directorio switch(int err) {
  case -1: error msg;
  default: listado dir;
};

struct param_abrir {
  nombrefich nf;
  modos_abrir modo;
};

union ret_abrir switch(int err) {
  case -1: error msg;
  default: descriptor f;
};

union ret_error switch(int err) {
  case -1: error msg;
  default: void;
};


union ret_tamanio switch(int err) {
  case -1:  error msg;
  default:  unsigned int tam;
};

struct param_leer {
  descriptor f;
  unsigned int cuantos;
  unsigned int desde;
};

union ret_leer switch(int err) {
  case -1:  error msg;
  default:  trozo buff;
};

struct param_escribir {
  descriptor f;
  trozo buff;
  unsigned int desde;
};

union ret_escribir switch(int err) {
  case -1:  error msg;
  default:  unsigned int escritos;
};
```

```
program FILESERVER {
  version UNSAFE {
    void           acabar    (void)           = 1;
    ret_directorio directorio(void)           = 2;
    ret_error      borrar    (nombrefich)     = 3;
    ret_abrir      abrir     (param_abrir)    = 4;
    ret_error      cerrar    (descriptor)     = 5;
    ret_tamanio    tamanio   (descriptor)     = 6;
    ret_leer       leer      (param_leer)     = 7;
    ret_escribir   escribir  (param_escribir) = 8;
  } = 1;
} = 0x2PON_AQUI_TU_NUMERO;
```

B. Entrega de ejercicios

La práctica se entregará en las máquinas `sirio` y `orion` en un subdirectorio del directorio raíz de cada alumno denominado `distribuidos/p2`. En ese directorio tiene que haber un fichero comprimido denominado `p2.zip` o `p2.tar.gz` que al ser descomprimido tiene que generar todos los ficheros necesarios para que los distintos ejercicios se puedan compilar y enlazar en la máquina en que se encuentre. Dentro del archivo comprimido se debe incluir algún tipo de mecanismo de compilación automatizada de los ejercicios de la práctica (un fichero de mandatos del *shell* o bien un `Makefile`). Tiene que ser posible descomprimir y ejecutar los programas en cualquier entorno y por cualquier usuario.

Los ejecutables generados por la compilación se deben llamar **`servfich`** (servidor de ficheros), **`rpcftp`** (cliente para el servidor anterior) y **`finalizador`** (cliente específico que invoca el servicio `acabar`). En la carpeta donde esté el ejecutable **`servfich`** debe haber también una carpeta llamada **`ficheros`** que contenga algunos ficheros de prueba para el cliente.

C. Corrección de los ejercicios

El alumno/a puede probar su cliente contra su propio servidor, preferiblemente estando cada uno en una máquina diferente. Para ello lanzará el servidor sin argumentos en una máquina y el cliente en otra especificando el nombre de la máquina del servidor. Por ejemplo:

```
$ ./rpcftp sirio.edv.uniovi.es
Bienvenido al cliente RPCFTP.
La localización del servidor en sirio.edv.uniovi.es ha tenido éxito.
Use el comando PROBAR para verificar que el servidor está activo.
Use HELP para ver otros comandos disponibles.
RPCFTP> _
```

Pero además de esta prueba, el cliente también puede ser probado contra el servidor de otros compañeros/as. Puesto que el interfaz que usan es el mismo (el fichero `.x` suministrado), la invocación remota de servidores programados por otro también debería funcionar. En este caso debería especificarse el número de programa con que se desea conectar. Por ejemplo:

```
$ ./rpcftp sirio.edv.uniovi.es 20843467
Bienvenido al cliente RPCFTP.
La localización del servidor 0x20843467 en sirio.edv.uniovi.es ha tenido éxito.
Use el comando PROBAR para verificar que el servidor está activo.
Use HELP para ver otros comandos disponibles.
RPCFTP> _
```

Téngase en cuenta que el servidor no incorpora mecanismo alguno de autenticación, por lo que cualquier cliente puede invocar sus servicios. Esto implica que cualquier cliente puede borrar tus ficheros (en la carpeta `ficheros`), o crear otros nuevos. Por eso es importante que finalices el servidor una vez hayas realizado las pruebas. Usa para ello el cliente finalizador (si deseas tener un mayor control sobre qué clientes pueden finalizar al servidor, puedes incorporar mecanismos de autenticación en el servicio `acabar` y hacer que el cliente finalizador se autentique).