Sistemas Operativos I - LCC
Trabajo Final - 2012

El trabajo se realizará en grupos de dos o tres estudiantes. Se entregará digitalmente subiéndolo al SVN de la materia junto con un informe explicando (no más de dos páginas) las soluciones y decisiones tomadas. Deberá crear una carpeta en su directorio de SVN y agregar en el informe quienes son los integrantes del grupo.

El trabajo tiene fecha límite el viernes 3 de agosto de 2012. Para obtener un 10 el estudiante debe implementar alguno de los puntos adicionales.

Pueden realizar consultas por email a

`{eruiz0,fbergero,glgrin}@fceia.unr.edu.ar`

Sistema de Archivos Distribuido

Un sistema de archivos (o FS) distribuido o sistema de archivos de red es un sistema de archivos de computadoras que sirve para compartir archivos, impresoras y otros recursos como un almacenamiento persistente en una red de computadoras.

El trabajo consiste en implementar un sistema de archivos distribuidos simplificado con un set de operaciones mínimas. El FS tiene las siguientes simplificaciones:

- No soportará directorios, es decir todos los archivos están en el mismo nivel
- Sólo se compartirán archivos planos (ni impresoras, ni directorios)
- Los archivos no tienen permiso, ni propietario, ni otros atributos
- Los archivos siempre se abren en modo lectura y escritura. No puede haber dos clientes accediendo al mismo archivo a la vez. Los archivos se abren siempre en modo append.
- Las llamadas son siempre síncronas (bloqueantes) desde el punto de vista del usuario
- Los archivos permanecerán en memoria (no serán escritos a disco) en una primera versión

- Los nombres de archivo no tendrán espacios, sólo caracteres `[a-zA-Z0-9.]`
- Las escrituras se realizan completas o fallan

El FS debe actuar como un servidor escuchando en el puerto TCP 8000. El protocolo entre cliente y servidor será mediante envió de strings sobre un socket TCP. La forma general de los pedidos del usuario al servidor será:

```
CMD OP1 OP2 ...
```

donde `CMD` son tres letras que indican la operación a realizar seguida posiblemente de sus argumentos. La respuesta del servidor será de la forma:

```
OK OP1 OP2 ...
ERROR OP1 OP2 ...
```

donde `OK` indica que el pedido se realizó correctamente devolviendo el resultado si lo hubiera como argumento y `ERROR` indica que hubo algún error al ejecutar el pedido.

Caso de uso

El siguiente extracto de una conexión, realiza un `ls`, luego crear un archivo `archivo2.txt`, escribe "hola mundo", intenta leer de un descriptor inválido, cierra el archivo y luego la conexión.

```
C: CON
S: OK ID 5
C: LSD
S: OK archivo1.txt resumen.doc video.mpg
C: RM archivo1.txt
S: OK
C: CRE archivo2.txt
S: OK
C: OPN archivo2.txt
S: OK FD 103
C: WRT FD 103 SIZE 10 Hola mundo
S: OK
C: REA FD 40 SIZE 16
S: ERROR 77 EBADFD
C: CLO FD 103
S: OK
C: BYE
S: OK
```

Las operaciones que permite al usuario son:

- `CON` Inicia la comunicación con el servidor y éste le devuelve un `ID` de conexión

- `LSD` Pide un listado de los archivos existentes en el FS y el servidor responde con la lista de archivos separada por espacios terminada por el caracter nulo.
- `DEL ARG0` Borra el archivo `ARG0` del FS. El servidor responde `OK` si el archivo existe y nadie lo tiene abierto.
- `CRE ARG0` Crea un archivo `ARG0` en el FS. El servidor responde `OK` si es que este archivo no existía previamente.
- `OPN ARG0` Abre el archivo indicado por el primer argumento. El servidor devuelve `OK FD NFD` donde `NFD` es un número entero (representado como string terminada por cero)
- `WRT FD ARG0 SIZE ARG1 ARG2` Pide la escritura del buffer `ARG2` de tamaño `ARG1` en el descriptor `ARG0`. El servidor responde `OK`.
- `REA FD ARG0 SIZE ARG1` Lee `ARG1` bytes del archivo dado por el descriptor `ARG0`. El servidor responde `OK SIZE ARG3 ARG4` cuando la lectura fue correcta indicando que leyó el buffer `ARG4` de `ARG3` bytes de longitud donde `ARG3` $\leq$ `ARG1`. Si se llega al final del archivo, deberá devolver un buffer de tamaño 0.
- `CLO FD ARG0` Cierra el archivo `ARG0` El servidor responde `OK`.
- `BYE` Cierra la conexión (cerrando todos los archivos abiertos). El servidor responde `OK`.

Arquitectura del Servidor

El servidor debe mantener varios procesos/hilos donde cada uno de estos tendrán parte del FS. El servidor consistirá en un módulo encargado de la escucha en el puerto TCP (dispatcher) y varios mini-servidores encargados de las operaciones(workers). El número de workers es fijo (se sugiere 5).

Cuando un cliente se conecta, el dispatcher crea un nuevo hilo que atenderá todos los pedidos de ese cliente. El nuevo hilo escoge aleatoriamente algún worker al que le solicitará los servicios necesarios para atender los pedidos del cliente hasta que se desconecte, momento en el cual el hilo termina.

Los archivos creados en un worker quedarán en ese worker para siempre. Los workers deberán comunicarse entre sí para dar la sensación de que el FS es único. Por ejemplo, cuando un cliente pide abrir un archivo, si el worker que lo atiende no tiene ese archivo, deberá preguntar a los otros workers si ellos lo tienen. Si es así, se devolverá un descriptor y todos los pedidos realizados sobre este descriptor serán reenviados al worker que posee el archivo.

En la figura 1 vemos la arquitectura del sistema.

- Los clientes se conectan al servidor a través del dispatcher (que escucha en el puerto TCP 8000).

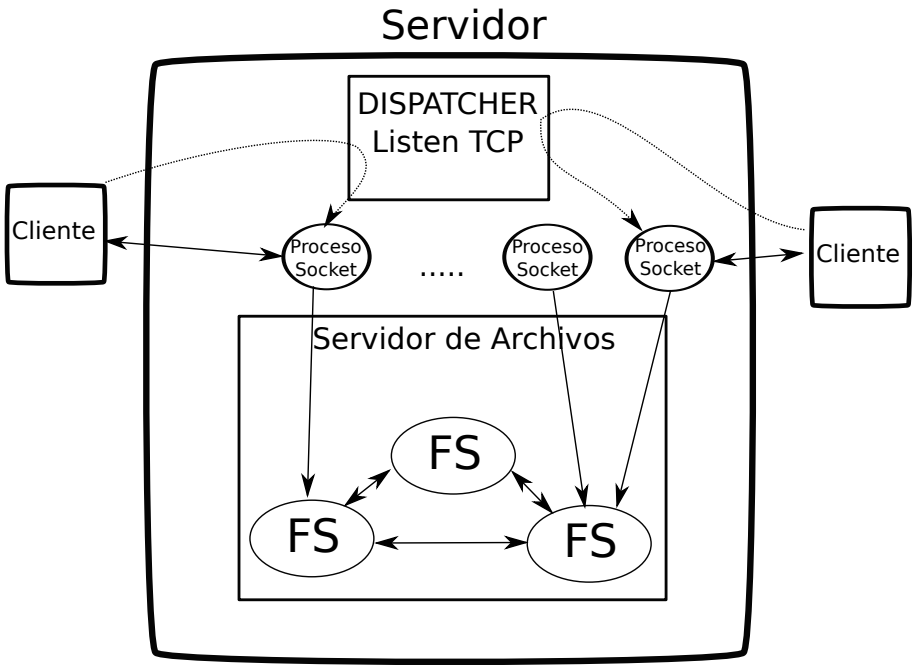

Figura 1: Arquitectura Cliente/Servidor

- Cuando un cliente se conecta al dispatcher, este último crea un hilo (proceso socket) que atenderá todos los pedidos de ese cliente. Cuando el cliente termina la sesión este hilo muere.
- El proceso socket se conectará a algunos de los workers FS (elegido en principio con una política aleatoria) y reenviará todos los pedidos del cliente a ese FS.
- Al ser distribuido el sistema de archivos, para realizar las llamadas al sistema (por ejemplo listar archivos), se debe unir los resultados de la lista de todos los archivos de cada FS. Lo mismo cuando se crea o abre un archivo. En el caso del creado, el FS que lo inicia debe consultar a todos los demás para ver si ese archivo no existe ya. En el caso de la apertura, el FS que lo inicia debe saber cual otro FS posee ese archivo (si es que no lo tiene él).

Cliente del FS

El cliente es dado por la cátedra. Se encuentra en la página web de la materia (aquí). El paquete se compila ejecutando `make` y crea varios ejemplos de uso:

- `cat arch0 arch1 ...` concatena el contenido de los archivos pasados como argumento y los escribe a salida estándar.

- `ls` lista los archivos del sistema de archivos.
- `create arch0` crea el archivo con el contenido que lee de la entrada estándar (terminada con Control+D).
- `cp arch0 arch1` copia un archivo a otro.

Objetivo

El estudiante deberá implementar el servidor de archivos mencionado, incluyendo dispatcher y workers. La implementación debe realizarse tanto en Erlang como en POSIX Threads.

Metodología

El estudiante deberá:

- Definir la estructura de módulos y procesos.
- Analizar y seleccionar las estructuras de datos necesarias para representar tanto archivos, como sesiones, descriptores, pedidos, etc.
- Implementar el manejo de la conexión TCP del servidor en el modulo dispatcher
- Definir el protocolo de comunicación de los workers entre sí y entre los hilos que atienden la conexión y los workers.
- Implementar las llamadas del cliente. Se sugiere inicialmente implementarlas como esqueletos vacíos que siempre devuelven `ERROR` y luego ir completando cada una de ellas.

Requisitos adicionales

Los siguientes puntos serán evaluados para obtener una calificación extra (hacerlo sólo para una de las implementaciones a elección):

- Mensajería tolerante a fallas (con timeouts). Si el FS no responde pasado cierto tiempo (100 ms por ejemplo) el hilo que atiende el pedido devolverá `ERROR 62 ETIME`.
- Permitir la apertura de sólo lectura y sólo escritura dejando múltiples aperturas en modo lectura.
- Implementar `RM` teniendo en cuenta si el archivo está abierto. En este caso se deberá borrar cuando el último cliente lo cierra.
- Hacer que los workers estén en distintas PC.

- Re-inicio del grupo de workers si se cae alguno. Aquí los pedidos que fueron realizados se perderán y el FS se reinicializará vacío.
- Implementar FS sobre archivos del FS local. Puede suponer que se crearán en el mismo directorio pero cada archivo pertenecerá a un worker determinado.

Sugerencias

Puede ser útil realizar un mecanismo de broadcast entre los distintos FS para implementar las llamadas que necesitan comunicación entre todos ellos. Para implementar la conexión TCP entre cliente y el dispatcher puede reutilizar lo realizado para el servidor de eco en POSIX Threads. La comunicación entre hilos POSIX la puede realizar como desee (teniendo cuidado con race conditions) o utilizando POSIX Messages Queues [2, 1].

Para implementar la conexión TCP en Erlang puede consultar el capítulo de sockets en [3].

Referencias

[1] David R. Butenhof, *Programming with POSIX Threads*.

[2] Bill O. Gallmeister, *Posix.4: Programming for the Real World*.

[3] Francesco Cesarini y Simon Thompson, *Erlang Programming A Concurrent approach to software development*.