Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Trabajo Final

Sistemas Operativos I

SISTEMA DE ARCHIVOS DISTRIBUIDO

Biasín Franco, Rodríguez Jeremías, Meli Sebastián

17 de junio de 2016

Índice

1. Introducción

Nuestro trabajo consiste en la implementación de un sistema de archivos (FS) distribuido, que permite compartir archivos en una red de computadoras.

El FS actúa como un servidor, escuchando en el puerto TCP 8000, que se comunica con los usuarios (clientes) mediante el envío de strings sobre un socket TCP. Implementamos un set de operaciones básico que permite a los usuarios manipular archivos.

Realizamos dos implementaciones en distintos lenguajes: Erlang y C con POSIX Threads. A continuación las describiremos individualmente.

2. Implementación en Erlang

2.1. Estructura General

Veamos cómo funciona nuestro FS una vez que ya está iniciado y funcionando, para presentar su estructura general.

Siguiendo la idea del enunciado, el nuestro FS es un servidor compuesto (entre otros) por 5 mini-servidores llamados workers. Cada worker contiene algunos de los archivos del FS, y deberá trabajar cooperativamente con los demás para satisfascer las demandas de los usuarios, de tal modo que desde el exterior se perciba un FS no distribuido.

Los clientes se conectan al puerto TCP 8000, donde el proceso principal (mon) les asignará alguno de los 5 workers al azar. Por cada cliente nuevo, se creará un hilo *handler* que se encargará de gestionar las solicitudes del cliente en cuestión e interactuar con el worker que le fue asociado.

El cliente inicia la comunicación con el comando CON. Luego, podrá utilizar alguno de los siguientes comandos:

```
LSD
RM <Filename>
CRE <Filename>
OPN <Filename>
WRT FD <FileId>
SIZE <Int> <Buffer>
REA FD <FileID> SIZE <Int>
CLO FD <FileID>
BYE
```

El handler se encargará de recibir por el socket correspondiente la cadena de texto, parsearla, determinar si corresponde a un comando válido y reenviarla al worker asignado. Finalmente, recibe la respuesta y la comunica al usuario.

Un worker es un proceso que recibirá solicitudes de procesos handler, y se encargará de obtener la respuesta. Posee un conjunto de archivos propios, que junto a los archivos de los demás workers conforman todos los archivos del FS. Se comunica con los demás workers mediante pasaje de mensajes (al igual que con los handlers, y con todos los demas procesos del FS).

Para analizarlo con un ejemplo, supongamos que un handler H se comunica con un worker W solicitándole la creación del archivo A. El worker deberá verificar que el archivo no exista ni en su espacio de archivos ni en el de los demás workers. Caso afirmativo, creará el archivo en su espacio de archivos. Caso negativo, comunicará al handler la imposibilidad.

De este modo, creamos tres categorías de mensajes que puede recibir un worker:

- from handler: un mensaje recibido de un handler con un comando aún no atendido. (e.g. crear el archivo A)
- from worker: un mensaje recibido de un worker compañero, solicitando alguna información. (e.g. ¿Existe el archivo A entre tus archivos?)
- from rworker: un mensaje recibido de un worker compañero en respuesta a una solicitud hecha. (e.g. No, no tengo ningún archivo de nombre A).

Cada mensaje recibido es manejado por una función homónima a su categoría.

Finalmente mencionar que en muchos casos se requiere que el worker mantenga información del estado de una consulta (¿Cuántos workers han respondido a una dada consulta?) o del estado de un archivo (Abierto, marcado para ser borrado, etc). Para solventar esta necesidad, cada worker mantiene una lista de tracks, que tiene estos datos.

2.2. Inicialización

En un principio se invoca a server:start(), que se encarga de spawnear el proceso principal (mon) y levantarlo si cae anormalmente. El proceso mon spawnea los 5 workers y se encarga de darle los datos necesarios para comenzar a funcionar (los Pids de los demás workers). Finalmente, se mantiene gestionando el primer contacto con cada cliente y la asignación de handlers que les atiendan.

2.3. Extensiones Opcionales

De entre las extensiones opcionales, elegimos:

- Implementación de RM de tal forma que, si un archivo está abierto, sea borrado cuando lo cierren.
- Re-inicio del FS si se cae.

2.4. Algunas Decisiones de Diseño Importantes

- El método de comunicación entre workers, utilizando broadcasting y contadores. Esto clarificó mucho el trabajo.
- La inclusión de una estructura "Track"dentro de cada worker, fácilmente manipulable, que nos permitió almacenar todo tipo de información útil ordenadamente (archivos abiertos, comandos en procesamiento, archivos marcados para ser borrados).
- El hecho de que cada worker conoce qué usuarios tienen abierto cada archivo de su espacio de archivos, y qué archivos tienen abiertos cada usuario que le han sido asignados. Esta pequeña redundancia facilita la respuesta de solicitudes, aunque requiere mucho cuidado para que no haya inconsistencias.

2.5. Módulos y Código

Finalmente, exponemos brevemente los distintos archivos y directorios:

- **src/server.erl** Módulo principal, donde está definida start.
- **src/server/mon.hrl** Define la función mon, que inicia los workers y recibe las conexiones.
- **src/server/wait_connect.hrl** Auxiliar para tratar con conexiones en mon.
- **src/server/handler.hrl** Interfaz entre el usuario y los workers.
- **src/server/fider.hrl** Facilita la obtención de ids únicos para archivos.
- **src/server/workers_utils/** Utilidades generales (e.g. manipulación de archivos, tracks).
- **src/server/worker/** Implementación de workers.

3. Implementación en C

La estructura del FS es completamente análoga a la recién vista en Erlang. Las secciones 2.1, 2.2 y 2.3 siguen siendo válidas para la implementación en C, con la salvedad de que en C sólo implementamos el punto adicional de marcar archivos para ser borrados.

3.1. Decisiones de Diseño Importantes

- El método de comunicación entre workers, usando broadcasting y contadores (igual que en Erlang).
- En esta implementación, los workers poseen una estructura mucho más compleja:

```
typedef struct _workerdata {
int id;
queue * myqueue;
queue * queues[5];
pthread_t thread;
File * files;
Tracker * cre_tracker;
Tracker * opn_tracker;
Tracker * lsd_tracker;
Tracker * rm_tracker;
Opened_Files * opened_files;
} workerdata;
```

Que incluye cuatro estructuras Tracker (una para cada comando), una lista de archivos abiertos (opened_files), una lista de los archivos que posee (files) y su ID (id).

3.2. Módulos y Código

Comentamos brevemente los distintos archivos y directorios:

datagram Expone toda la información sobre los datagramas[1]
files Aqui se maneja toda la información sobre los archivos. Todas las funciones (agregar, crear, chequear existencia, obtener fid, buscar, inicializar, leer, abrir, reservar espacio, borrar, escribir)
functions Funciones auxiliares muy triviales, que ayudan mucho a ordenar el código
headers Algunas cabeceras y definición de constantes
note Define las notas y la función para crearlas
opened Todas las funciones asociadas a los archivos abiertos (marcar para borrar, chequear si está abierto, chequear si está abierto por un usuario en particular, etc)
storage Aquí se guardarán los archivos que manejan los workers
tracker/tracker.h Definición de lo que es un Tracker
tracker Todas las funciones asociadas a los Tracker
worker/workerdata.h Estructura de los datos que tiene un worker (ya comentado arriba)
worker Aquí están los archivos más complejos (from_handler, from_worker, from_rworker) que permiten a los workers comunicarse entre ellos y trabajar cooperativamente
client.c El cliente
server.c El servidor, con el main
worker.c Los workers

3.3. Diferencias con la implementación en Erlang

Las principales diferencias son respecto a la implementación en Erlang son:

[1] Un datagrama es una unidad de transferencia de base asociada con una red de conmutación de paquetes

3.3.1. Comunicación entre procesos

La comunicación entre los procesos (workers) fue mucho más compleja.

Mientras que en Erlang contamos con todas las primitivas para hacer (por ejemplo) el broadcast en sólo 3 líneas, en C tuvimos que usar toda una estructura aparte, las colas.
Mediante las funciones enqueue y dequeue podemos manejar las colas de cada worker, pero estas funciones no son triviales.

En sí, la estructura de una cola en nuestro trabajo es esta:

```
typedef struct queue_t {
void **buffer;
int capacity;
int size;
int in;
int out;
pthread_mutex_t mutex;
pthread_cond_t cond_full;
pthread_cond_t cond_empty;
} queue;
```

Mediante mutexs es que podemos lograr manejar la zona crítica, al igual que hicimos durante el cursado en las prácticas.

Los enteros nos ayudan a saber el estado de la cola (y este estado, para saber cuando bloquear y cuando no al proceso).

Ahora, con las colas y las funciones enunciadas, podemos hacer que los workers trabajen de forma cooperativa y sin fallas ni inconsistencias.

3.3.2. Manejo de los comandos

Mientras que en Erlang alcanzaba con una tupla con un átomo como primer elemento, en C tuvimos que implementar la estructura command de esta manera:

```
typedef struct _command {
uint user_id;
cmd_name name;
lilString file;
uint fid;
uint size;
lilString text;
queue * user_queue;
char linea[BUFF_SIZE];
} command;
```

Donde:

- cmd_name es un tipo numerado, con los nombres de los comandos que se pueden usar (CON, LSD, RM, CRE, OPN, WRT, REA, CLO, BYE)
- Una lilString es una cadena larga de caracteres
- User_queue es la cola del usuario que usa el comando
- Fid es el id del archivo

- Size es el tamaño que tiene el archivo
- User_id es el id del usuario

Así es como podemos llevar toda la info necesaria en el datagrama para que funcione. Sólo a modo de curiosidad, la estructura del datagrama es esta:

```
typedef struct _datagram {
entity from;
uint from_id;
queue * respond_to;
command cmd;
void * message;
} datagram;
```

Donde:

- From es la entidad que lo envía
- From_id es el identificador de dicha entidad
- Respond_to es la cola donde deberá responder
- Cmd es el comando recibido