Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Licenciatura en Ciencias de la Computación
Sistemas Operativos I

Sistema de archivos distribuído

Alumnos:
BORRERO, Paula (P-4415/6)
IVALDI, Ángela (I-0561/4)
MISTA, Agustín (M-6105/1)

Docentes:
MACHI, Guido
GRINBLAT, Guillermo
DIAZ, José Luis

4 de Julio de 2016

Introducción

Un servidor de archivos distribuído es un componente de software que le ofrece al usuario final las operaciones necesarias para trabajar con un sistema de archivos virtual, aparentemente centralizado, donde todos los archivos parecen estar en una misma ubicación, cuando en realidad es probable que los mismos estén dispersos en varias unidades de disco, o más aun, en varias computadoras.

En éste informe analizaremos las implementaciones tanto en **C** como en **Erlang** de un servidor de archivos distribuido simple, esto incluye profundizar sobre algunas cuestiones de diseño tales como comunicación entre hilos, concurrencia y performance.

Implementación con Posix Threads

Estructuras de datos

- Session: representa las sesiones. Contiene datos que relacionan un cliente con su respectivo worker, seleccionado de forma aleatoria por el dispatcher cuando el usuario se conecta.
- Request: se usa para modelar los pedidos a los workers. Si el campo external (int) es 0 el pedido proviene de otro worker, en otro caso el pedido proviene del handler.
- Reply: representa las respuestas del worker al handler. Contiene dos campos: uno representa errores al procesar un comando (Es de tipo Error. Si no hay errores es NONE.) y el otro sirve para agregar información extra sobre el resultado.
- File: modela cada archivo.
- Worker_Info: permite guardar datos de cada worker, entre ellos: sus archivos, identificador, cola de mensajes y pool de descriptores de archivos.

Módulos

- Server: inicializa la conexión TCP, el dispatcher y los workers.
- Dispatcher: espera por nuevas conexiones. Cuando alguna ocurre, se ocupa de crear una sesión (Session) y lanzar un handler.
- ClientHandler: se encarga de parsear lo que recibe de cada cliente, convertirlo en un external request y mandarlo al worker correspondiente. Cuando recibe una respuesta (Reply) de algún worker, la procesa y envía el resultado al cliente.
- Worker: recibe del handler o de otros workers diferentes peticiones, las cuales intenta llevar a cabo. Opera sobre sus archivos cuando es necesario.

Decisiones de diseño

- Usamos **POSIX Messages Queues** para la comunicación entre los distintos nodos. La misma forma un anillo entre los workers.

- Para optimizar las operaciones, **no es necesario que una petición pase por todos los workers**.
  Al recibir una petición externa se la va pasando en anillo, el primero que pueda llevarla a cabo lo hará. Si es el worker principal (el que puede comunicarse con el cliente a través del handler) envía su respuesta inmediatamente. Sin embargo, si no lo es, no necesita terminar la vuelta, puede mandarle directamente los resultados al principal para que éste los comunique al exterior. Esto se puede hacer gracias a que se mantiene entre los pedidos la referencia al worker principal (*main_worker* dentro de *Request*).

- Para mostrar los mensajes del servidor se usaron **banderas de compilación condicional**. Con la bandera DEBUG, se muestran mensajes de estado del servidor. Con DEBUG_REQUEST podemos también ver la comunicación entre workers.

Implementación en Erlang

En este caso los workers forman un anillo similar a la versión de C. La diferencia es que siempre se realiza toda la vuelta (no hay optimización). Si se resuelve la petición antes de llegar al origen (exitosamente o con error) se propaga la respuesta hasta que llegue al origen. Si se da toda la vuelta sin resolver la petición se retorna error.
Cada worker conserva sus archivos, su pool de descriptores de archivo y el pid del siguiente worker en el anillo. Cada uno recibe mensajes de la forma {*Req, Pid, Count*}, donde Req representa a los comandos y argumentos, Pid es el PID del cliente solicitante y Count es un contador de saltos, que al llegar al número total de workers (*?N_WORKERS*) indica que se dio toda la vuelta.
El handler se divide en dos funciones, *handle_unlogged* se ocupa de conectar al cliente (*CON*) y *handle_logged* se ocupa del resto de las peticiones una vez que se tiene un identificador de cliente.
Cada archivo es representado mediante una tupla de la forma {*Name, Fd, Opener, Cursor, Size, Content*} y el conjunto de archivos de cada worker es una lista de tuplas de ese tipo.

Característica adicional

Implementamos en Erlang el punto adicional de mensajería tolerante a fallas (con timeout). Luego de 300 milisegundos se lanza el error: ERROR 62 ETIME.