Universidad Nacional de Rosario

Sistemas Operativos I

Trabajo Final

Integrantes:
Crosetti Mariano
Rinaldi Lautaro

Índice

1. Introducción

Nuestro *sistema de archivos distribuidos* fue especialmente diseñado de modo que no posea elementos centralizados que violen la concepción de ser distribuido. Además se realiza de modo transparente al usuario.
Se hizo especial énfasis en un diseño modular y flexible a futuras ampliaciones/modificaciones.

2. Requisitos adicionales implementados

Los puntos opcionales realizados (en el código de erlang) fueron:

- Sistema de archivos real.
- Capacidad de posicionar los workers en distintas PC's.
- Opción de abrir los archivos como lectura o escritura.

2.1. Nueva sintaxis para lectura y escritura

OPN ARG0 deja de existir, en su lugar se utilizan:

- OPNW ARG0 Abre el archivo ARG0 en modo escritura.
- OPNR ARG0 Abre el archivo ARG0 en modo lectura.

3. Decisiones tomadas

Con el fin de simplificar partes engorrosas del trabajo y poderse focalizar en otras que juzgábamos más importante (diseño y manejo de concurrencia) hemos acordado los siguientes puntos con la cátedra:

- Usar C++ para la implementación en vez de C.
- Los archivos no pueen contener espacios
- Hemos cambiado sutilmente el protocolo de respuestas para con los clientes para que sea más detallado e informativo.

4. Manual de uso

4.1. Erlang

1. Configurar correctamente el archivo worker_list.hrl.
2. Compilar todos los .erl ejecutando el script compall.sh
3. Ejecutar la función *dispatcher:init()* desde erlang en la PC donde el dispatcher sea instalado.
4. Corroborar que exista la carpeta fs dentro del directorio a montar los workers (pues aquí se crean los archivos).
5. RECORDAR: el Worker de ID 0 es el que empieza con el Token, por lo que debe ser lanzado cuando ya están todos funcionando.
6. Ejecutar la función *worker:init(MyId)* desde erlang en la PC donde residirá el Worker con ID especificada.
7. Los Clientes se conectarán al dispatcher por el puerto 8080.

4.1.1. Formato worker_list.hrl

-define(WORKER_LIST, L).
Donde L es una lista de cuatruplas de la forma:
{
IDWorker,
PuertoComunicacionDispatcher,
PuertoComunicacionWorkers,
IPWorker
}

4.2. C++

1. Compilar todo el proyecto con make.
2. Corroborar que la exista la carpeta fs (pues aquí se crean los archivos).
3. Ejecutar el programa main <cantida de workers>.
4. Los Clientes se conectarán al dispatcher por el puerto 8080.

5. Conceptos Preliminares

Aquí explicamos algunos conceptos necesarios para entender el trabajo:

- **Cliente local:** desde la perspectiva de un worker, los clientes que él atiende de manera directa.
- **Cliente remoto:** desde la perspectiva de un worker, los clientes que él atiende de manera indirecta: por consecuencia de pedido de tasks de otros workers.
- **ClientId:** ID del cliente local al worker que lo atiende.
- **WorkerId:** ID que identifica al worker de los otros.
- **GlobalId:** ID que identifica al cliente de manera única del resto de los clientes conectados al Sistema de Archivos Distribuidos. Internamente es una combinación del ClientId y el WorkerId del worker que lo atiende.
- **LocalFd:** identificador de archivo que le da nuestro sistema a los archivos locales abiertos.
- **GlobalFd:** identificador de archivo único en nuestro sistema. Internamente es una combinación del LocalFd y el WorkerId del worker que posee el archivo abierto.
- **Task:** son las tareas a resolver por los workers. Son generados a partir de los comandos de los usuarios (UserTasks) y como consecuencia de la interacción entre workers (WorkerTasks). Cada una puede entenderse como una orden específica, con campos particulares (datos de entrada). Se procesan de forma atómica.

6. Líneas generales de nuestro diseño

Cada Worker almacena información acerca del sistema de archivos y los clientes conectados a él.
El dispatcher a medida que recibe las conexiones y comandos de los clientes actúa de pasarela con los Workers.
Cada comando del usuario se transcribe en un Task para el Worker que lo atiende.
Los Workers procesan los Task de a uno a la vez. Muy a menudo el procesamiento de un Task genera que se le envíe un nuevo Task a algún otro Worker del anillo.
Los Workers forman un anillo virtual por donde circula un Token que sirve para manejar la concurrencia en el caso de las creaciones y borrados de archivos.

Un panorama más detallado se puede obtener leyendo la guía de módulos.

7. Guía de Módulos

El trabajo fue realizado por módulos, cada uno encargado de funciones bien definidas. En el código de Erlang los módulos están brevemente comentados. Sus funciones se listan a continuación.

7.1. Comunicación

Aquí agrupamos a los módulos que encapsulan funciones de comunicación entre los Workers y con los Clientes:

- **comunic:** encargado de la comunicación entre distintos workers(PC's) y entre workers y clientes.
- **dispatcher:** encargado de asignar (o redirigir) un cliente a un worker. En la implementación de Erlang podríamos ubicarlo en una PC diferente a la de los workers, probablemente en una DMZ.

7.2. Estructuras del Worker

Aquí agrupamos a los módulos que encapsulan las estructuras persistentes en cada Worker:

- **globalfiles:** guarda la imágen del sistema global que tiene el worker: los archivos que existen y sus respectivos dueños.
- **localfiles:** guarda el estado (lectores, escritores actuales) de los archivos locales (los que el worker es dueño).
- **fdmanage:** registra los archivos locales abiertos, asignando los LocalFd y llevando la correspondencia de los mismos con el Handle del Sistema de Archivos Real subyacente, el nombre del archivo en cuestión y el dueño del manejador.
- **openedfiles:** lleva la cuenta de los identificadores tomados por clientes locales.
- **localconections:** para cada cliente conectado al worker, almacena la correlación entre su Id. Esto permite que dado el Id se posea la información necesaria

para contestarle. En la implementación en Erlang se guarda el Pid del proceso que se encarga de atender la conexión. En la implementación de C++ se guarda un puntero a una syncQueue que actúa como inbox.

- **ids:** contiene la información necesaria referente a los identificadores descriptos. Además tiene el Id del worker en cuestión y del siguiente en el anillo virtual.
- **tokenqueues:** almacena la Lista de Creaciones Requeridas (y el cliente que espera por su creación) y la Lista de Borrados a Informar que actúan en al funcionamiento de los borrados/creaciones.
- **tokencontrol:** encapsula el almacenamiento del Token y las funciones que indican cuándo procesarlo y con qué frecuencia. También las funciones para acceder a la Lista de Creaciones a Notificar y Lista de Borrados a Notificar del Token.

7.3. Auxiliares

Aquí agrupamos a los módulos que realizan tareas de propósito general auxiliares a nuestro programa o encapsulan funcionalidades muy específicas:

- **mensaje:** encapsula las contestaciones de los Workers a los Clientes.
- **realfs:** encapsula la interacción con el sistema de archivos real subyacente.
- **parser:** función que tokeniza una cadena por espacios como separadores.
- **sockaux:** función que lee una conexión hasta el fin de línea o caracter nulo.
- **task:** encapsula y modela los Tasks, la forma en la que se acceden a sus campos, la generación de WorkerTasks a partir de los campos y la generación de UserTasks a partir de los comandos de usuarios.
- **workerdirs:** funciones para acceder a puertos y las direcciones de los workers.

7.4. Sistema

- **worker:** contiene las funciones encargadas de inicializar el worker, y el loop principal que ejecuta. Además contiene la función encargada de procesar al Token.
- **proctask:** se encarga de cómo procesar cada Task.

8. Manejo de Concurrencia - Token Ring

El manejo de concurrencias es uno de los puntos importantes, para este caso decidimos tratarlo implementando Token Ring: los workers se agrupan en un anillo virtual por donde circula un Token.

Cuando llega un pedido de creación de un archivo a un worker, éste verifica que no exista en su imagen del sistema un archivo con dicho nombre y añade el pedido a la cola de Lista de Creaciones Requeridas (manejada por el módulo tokenqueues), dejando esperando al cliente la respuesta para cuando llegue el Token y esta lista sea procesada.

Similarmente cuando llega un pedido de borrado de un archivo local, se verifica que exista y no esté en uso, se lo borra y se añade a la Lista de Borrados a Informar (en tokenqueues).

Dentro del Token circula la Lista de Creaciones ha Notificar y la Lista de Borrados ha Notificar de los workers para que otros workers actualicen su imagen del sistema. Así cuando un worker quiere informar un alta o baja lo pone en el Token Ring y lo saca cuando le llega nuevamente el Token (el mensaje habrá pasado por todos los demás worker una vez).

Cuando un worker procesa el Token quita los pedidos que le correspondan a él de ambas listas (que ya dieron una vuelta entera al anillo). Luego agrega sus Borrados ha Informar a la lista que el Lista de Borrados a Notificar del Token. Los pedidos de creación de archivos almacenados en la Lista de Creaciones Requeridas que no aparezcan ya en la Lista de Creaciones a Notificar del Token, se contesta al Cliente que la creación ha sido exitosa y se agrega a esta última lista del token, de aparecer contesta que el archivo ya existía con anterioridad.

Luego con este nuevo Token generado actualiza la imagen del sistema y pasa este nuevo Token al siguiente worker del anillo.

Como se puede observar, con este mecanismo se mantiene la consistencia del sistema y se resuelven eventuales condiciones de carrera.