Universidad Nacional de Rosario

Ciencias de la Computación

Sistemas Operativos I

Trabajo Final

Sistema de Archivos Distribuido

(con anillos dinámicos)

Autor

Martín Villagra

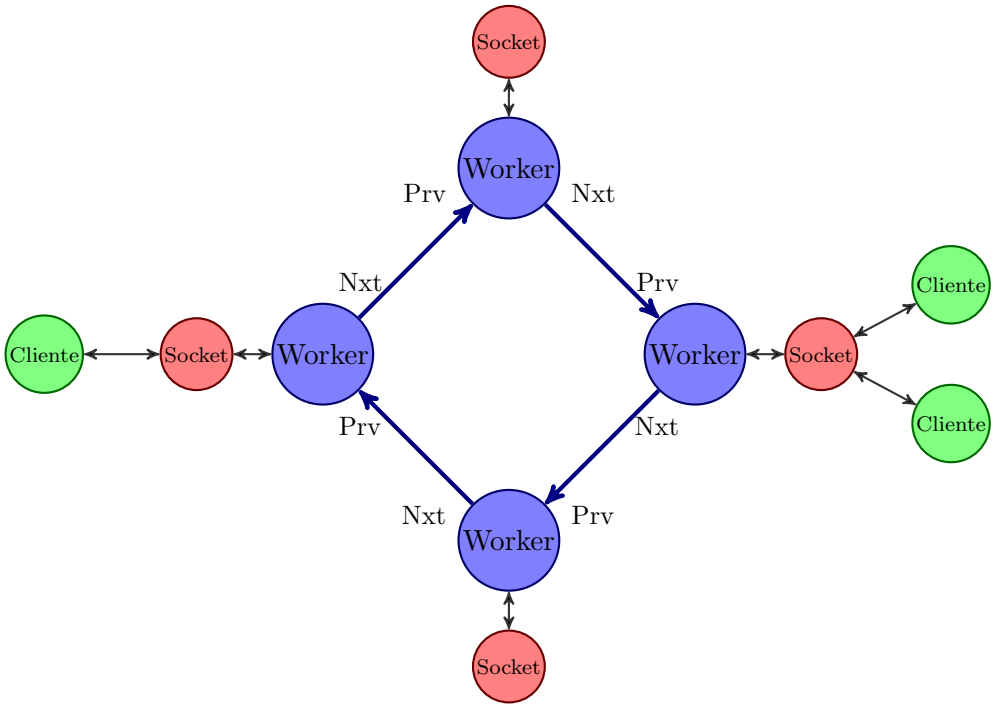
17 de diciembre de 2015

1. Introducción

Se presenta un sistema de archivos distribuido totalmente descentralizado escrito en Erlang, junto con detalles de su funcionamiento. El mismo se diseñó para que requiera la mínima intervención del usuario para configurarse.

2. Topología

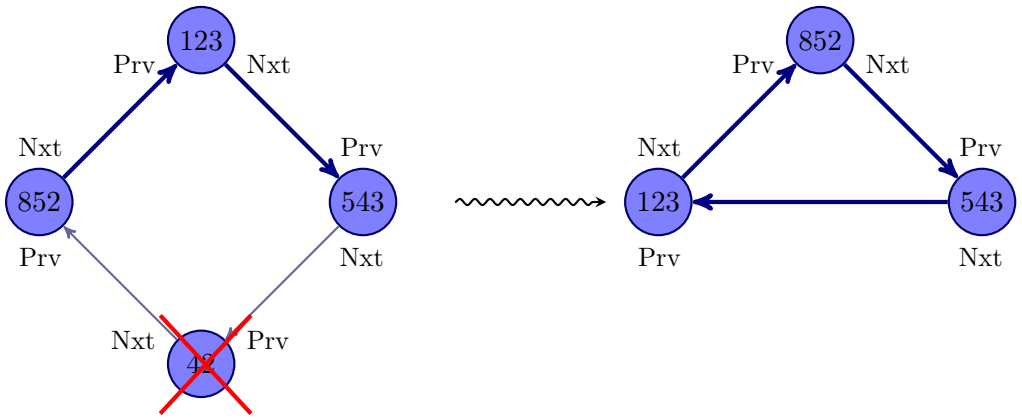
El sistema que aquí se muestra es diferente al diseño sugerido en el enunciado. Si bien los nodos[1] se siguen comunicando en forma de anillo, cada uno cuenta con un socket por el cual los clientes se conectan y no existe el dispatcher. En particular cada nodo tiene dos conexiones TCP, una al nodo anterior (Prv) y otra al siguiente (Nxt).



El sistema **sólo** está formado por estos workers. Cada nodo consiste en un programa totalmente autónomo.

[1] En nuestro sistema tomaremos nodo y worker como sinónimos.

Como la idea es requerir la mínima intervención del usuario, se utiliza un sistema de "anillos dinámicos". Los nodos se detectan entre sí automáticamente y en un tiempo finito forman la topología de anillo deseada.
La idea es que el anillo detecte cuando un nodo se caiga o cuando un nodo quiere incorporarse y el anillo se modifique apropiadamente.



Esto da lugar a una gran flexibilidad y reduce la configuración del sistema a casi nula. De hecho para ejecutar tanto el servidor como el cliente no es necesario ningún argumento.

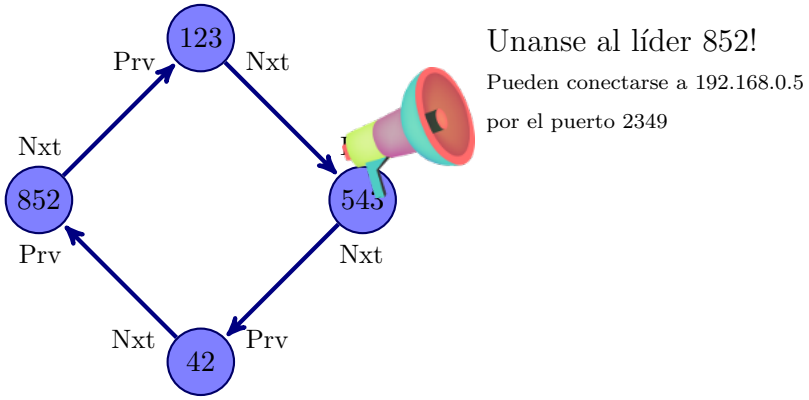
2.1. Funcionamiento

Inicialmente cada nodo genera para sí mismo un *ID* aleatorio con el cual identificarse que no va a cambiar. A fines prácticos se va a suponer que nunca hay dos nodos con el mismo *ID*.

Como al iniciar todavía no sabe nada del sistema[2], supone que es el único en la red y crea un anillo donde él es el único que pertenece (auto ciclo). Si bien esto parece absurdo, permite crear un invariante: un nodo está siempre en un anillo.

[2]fuera de saber que está siendo ejecutado

Cada anillo va a poseer un líder, el cual será el nodo con mayor *ID*. También hay un encargado de anunciar el anillo, el vocero. El vocero envía periódicamente un paquete multicast UDP[3] informando la existencia del anillo, junto con el *ID* de su líder y como incorporarse al mismo. El vocero a su vez recibe los nodos y clientes que quieran comunicarse con el anillo. El rol de vocero está simbolizado en el dibujo por quien tiene el megáfono. El megáfono va a ser periódicamente pasado como si fuera un token al siguiente en el anillo cada vez que un cliente se conecte, distribuyendo los clientes entre los workers.



Cada nodo va a estar permanentemente escuchando por mensajes que anuncien anillos nuevos. Los nodos prefieren estar con líderes con *ID*s más grandes. **Si un nodo detecta un anillo cuyo líder posee un *ID* más grande que el de el líder del anillo actual entonces abandona su anillo y procede a unirse al anillo detectado.**

Este último comportamiento junto con la únicidad de los *ID*s asegura que luego de un tiempo va a haber un único anillo conformado por todos los nodos en la red.

Si por alguna razón algún nodo se desconecta de sus vecinos, los vecinos proceden a desconectarse de sus otros vecinos y vuelven al estado inicial[4]. Se genera un comportamiento en cascada donde el anillo entero se rompe y todos los nodos quedan en el estado inicial, que conduce a que el anillo se forme nuevamente.

[3] `https://en.wikipedia.org/wiki/Multicast#IP_multicast`

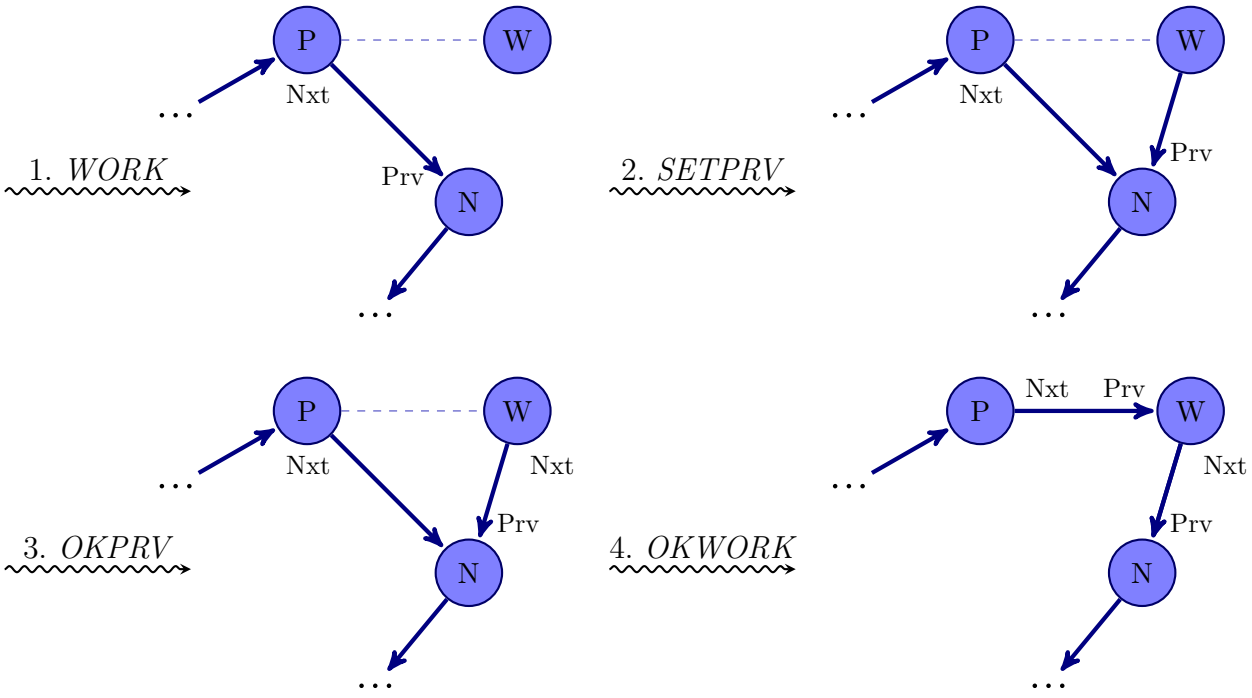
[4] Cada nodo vuelve a formar el auto ciclo, pero las conexiones con el cliente, los archivos abiertos y demases estados no se reinician.

La simpleza del sistema descripto tiene la ventaja de que la única operación importante que debemos implementar es adicionar un worker al anillo, la cual se verá a continuación.

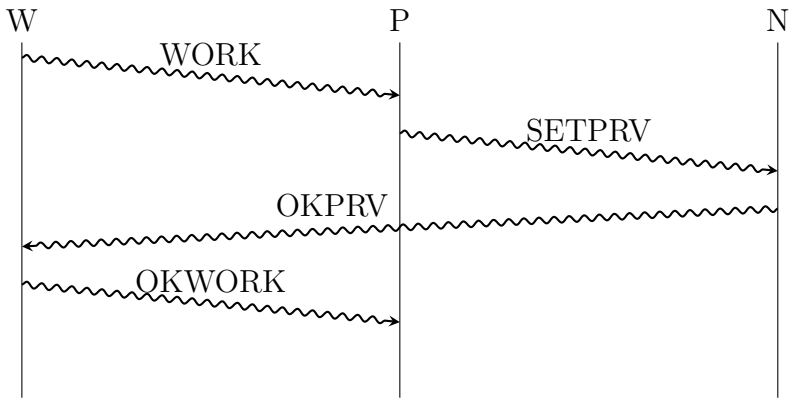
2.1.1. Protocolo para unirse

Existen tres nodos involucrados en el proceso **P**, **N** y **W**. Aquí **W** quiere incorporarse al anillo de **P** y **N**, estableciendo una conexión primero con **P**. Básicamente el procedimiento es:

1. **W** le dice a **P** que quiere unirse entre **P** y **N**.
2. Luego **P** le dice a **N** que ponga a **W** como su anterior (Prv).
3. **N** informa a **W** de la conexión y este lo pone como su siguiente (Nxt).
4. Por último se establece el enlace entre **W** y **P**.



Este esquema muestra los mensajes enviados en función del tiempo:



Además *WORK* debe contener información de como conectarse a **W**, así esta le es pasada a **N** a través de *SETPRV*.

Existen casos especiales que hay que manejar cuando el anillo es de tamaño uno (**P**=**N**). También está contemplado cualquier error que pueda ocurrir durante el procedimiento (en cuyo caso se aborta la operación).

3. Sistema de archivos

Una vez establecido el anillo, el sistema distribuido tiene que tener alguna comunicación entre los workers.

Para esto el worker implementa los mensajes de anillo. Un mensaje es iniciado por algún worker y el mismo viaja por todo el anillo en orden secuencial. Cada nodo recibe el mensaje, lo procesa de alguna forma y vuelve a enviarlo al siguiente del anillo. De esta forma al finalizar todos los nodos habrán procesado el mensaje. Cuando el mensaje da una vuelta completa el mismo es devuelto al proceso que creó el mensaje. Esto permite crear "funciones" que operan sobre la totalidad de la información del sistema distribuido.

Por ejemplo para el mensaje *lsd*, el worker primero crea el mensaje {''lsd'', [ ]}. Cada nodo que procesa el mensaje agrega a la lista los archivos locales que posea y lo vuelve a reenviar. Así al finalizar el worker recibirá el mensaje {''lsd'', XS}, con XS conteniendo la lista de todos los archivos globales.

3.1. Problemas de concurrencia

Surgen problemas de concurrencia pues, imaginemos el hecho de que dos workers A y B intenten abrir el mismo archivo F a la vez, que está actual-

mente cerrado. Cada uno mandaría un mensaje de anillo para ver si alguien tiene abierto a F. Alguno de los dos tiene que percibir que F no estaba abierto. Sin perdida de generalidad suponemos que es A. Como A percibe que no está abierto, lo abre. El problema está en que B no necesariamente percibe que F fue abierto por A, pues B podría haber procesado el mensaje antes de abrir el archivo. Al no percibir que está abierto, lo abre. Esto lleva a un estado erroneo pues **dos workers tienen abierto el mismo archivo**.

Ocurren cosas similares con otros comandos. Bajo el mismo argumento un archivo podría ser creado en dos workers distintos. Podría hasta abrirse un archivo que justo fue borrado.

3.2. Solución

La idea sería que para realizar cambios en el estado de un archivo[5], un worker primero debe adquirir un lock sobre el mismo (incluso si el archivo no existe y se desea crearlo). Mientras se posea un lock ningún otro worker va a poder operar sobre ese archivo. Sólo cuando se obtiene el lock el worker puede operar sobre ese archivo y una vez que termina libera el lock.

Los lista de locks son el único estado global que mantiene cada worker. Los cambios en los locks se tienen que propagar por todos los nodos para mantener esta lista.

3.2.1. Procedimiento para adquirir un lock

Para que el nodo A obtenga un lock sobre un archivo primero se fija en la lista de locks globales que posee. Si no aparece, entonces no está bloqueado y procede a intentar adquirirlo.

Para hacerlo envia un mensaje de anillo M preguntando a cada nodo si se opone a la adquisión de tal bloqueo y si no, que lo agregue a su lista de locks. Si nadie se opone, el lock se adquiere y las listas de locks quedan actualizadas con este nuevo lock.

Si en cambio un nodo B recibe este mensaje y se encuentra con que el archivo está bloqueado por otro nodo, la única explicación para esto es que ese lock todavía no había llegado a A al momento de emitir M (pues sino A hubiera sabido que el lock ya fue adquirido mirando su lista de locks). Es decir estamos en el caso de dos locks que se emitieron 'casi' a la vez. En este

[5]renombrar, abrir, escribir, crear, borrar, etc

caso debemos decidir cual va a prevalecer y elegimos que prevalezca el lock proveniente del worker con *ID* más grande, descartando el otro.

Este comportamiento nos asegura que el lock va a ser dado a exactamente un worker:

- El de *ID* más grande va a recibir que su lock fue aceptado, pues si no, significaría que otro worker con mayor ID ya obtuvo ese lock recientemente, lo cual es imposible porque no existe.
- Los nodos con *ID*s más chicos van a recibir que su lock fue denegado, pues si no es denegado significaría que el de ID más grande no se opuso a la petición lo cual significaría que no intento adquirir el lock recientemente.

El estado global de los nodos queda consistente, pues el mensaje del de *ID* más grande va a reemplazar de la lista de locks de cada uno de los workers lo que modificaron los mensajes de los de *ID*s más pequeños.

Este sistema tiene semejanzas con el protocolo de commit en dos fases[6] en el sentido de que hay una votación para modificar el estado global (los archivos bloqueados). Pero la diferencia está en que no se necesario deshacer los cambios de las operaciones fallidas.

4. Casos Especiales

4.1. Estabilidad

Antes de formarse el anillo final con todos los nodos pueden formarse varios anillos más pequeños. Un nodo puede unirse a un nodo pensando que es el más grande solamente porque todavía no recibió ningún anuncio del verdadero líder.

Es por esto que es preferible esperar algún tiempo desde la creación del anillo para que todos los nodos se unan antes de comenzar a mandar mensajes de anillo. Esto provoca que cuando un anillo se rompe justo cuando el cliente manda un comando, el usuario lo perciba como un ligero retardo en la respuesta de su comando, pues su consulta está esperando a que el anillo se estabilize.

[6] `https://en.wikipedia.org/wiki/Two-phase_commit_protocol`

4.2. Archivos huérfanos

Cuando un cliente se desconecta, el worker al cual estaba conectado cierra todos los archivos que dejó abiertos. Ahora propongamos otro escenario: supongamos que el cliente se conecta al worker A y abre un archivo ubicado en B. Luego si A deja de funcionar el cliente obviamente queda desconectado y el archivo de B queda abierto. Es decir en el resto de los workers figura como si alguien tiene un lock sobre él.

4.2.1. Solución

La solución implementada consiste en que cada vez que se rompa el anillo (y pasado el período de estabilidad) cada worker se cerciore de que los archivos que tiene abiertos localmente efectivamente estén abiertos por algún cliente. Si hay algún archivo abierto pero no es poseído por ningún cliente, se procede a cerrarlo y liberar su lock.

4.3. Escritura sin respuesta

Existe un caso especial que es cuando un cliente manda una instrucción para escribir en un archivo y justo se rompe el anillo. Por defecto los mensajes de anillo que circulaban por el anillo antes de romperse se destruyen. Entonces es imposible saber si el archivo fue modificado o no. Si reenviamos la petición de escritura podríamos estar escribiendo dos veces.

4.3.1. Solución

La solución fue crear un caché que guarda las respuestas. Primero, se numeran los paquetes de cada worker. Así un paquete es identificado por el par (worker *ID*, número de paquete). Para cada paquete el caché guarda su respectiva respuesta. Ahora cuando se recibe un mensaje antes de procesarlo primero hay que fijarse en el caché si ya había sido procesado anteriormente y si es así devolver la respuesta guardada en el caché.

De esta forma cuando se reenvie el paquete, la respuesta va a ser la correcta y la escritura no se va a realizar dos veces.

5. Código

El código del servidor y de los clientes está disponible en el respositorio:
`http://dcc.fceia.unr.edu.ar:81/svn/lcc/R-322/Alumnos/2015/mvillagra/TrabajoFinal/`

En el archivo *README.md* encontrará información de las dependencias y de como ejecutarlo.

A continuación se detallan los módulos principales por los cual está formado.

- **main.erl** Contiene la entrada principal, donde se inician todos los procesos y se queda esparando a que haya algún error.
- **worker.erl** Es el proceso que recibe los mensajes del anillo y de las nuevas conexiones y los maneja apropiadamente. Su estado y operaciones sobre él están en *workerstate*. También provee una interfaz para mandar un mensaje a través del anillo, que es usada intensivamente por *fs.erl*
- **ring.erl** Es un módulo que provee funciones relacionadas al anillo (unirse, incorporar un nodo, crear auto ciclo, etc). Es utilizado sólo por *worker.erl*.
- **fs.erl** Es la parte más importante del servidor, aquí se proveen funciones para trabajar con el sistema de archivos. Su estado y operaciones sobre él están en *fsstate*. Trabaja un nivel más arriba que *worker.erl*.
- **client.erl** Contiene el handler que recibe los comandos del cliente, los procesa (posiblemente usando *fs.erl*) y genera una respuesta. Trabaja un nivel más arriba que *fs.erl*. Su estado y operaciones sobre él están en *clientstate*.
- **megaphone.erl** Ejecuta un proceso que representa el megáfono del vocero. Al iniciar inicializa el socket UDP. Puede ser activado y desactivado.
- **cache.erl** Ejecuta un proceso que contiene la cache del worker.
- **cmd.erl** Contiene operaciones para parsear los mensajes entrantes y para transformarlos a binario así pueden ser enviados por los sockets. Es utilizado desde el lado del worker y desde el cliente.