# Trabajo Práctico Final
# R-322: Sistemas Operativos I

Felipe Andrés Tenaglia Giunta
Legajo: T-2658/1

20 de junio de 2014

# Sistema de Archivos Distribuído

## Arquitectura

En ambas versiones, el sistema se compone de las siguientes partes:

- Al inicio, un hilo atiende las conexiones entrantes (dispatcher) y otros 5 hilos representan a los workers (Estos últimos conforman un anillo lógico)
- Al recibir una conexión, se crea un nuevo hilo (proceso socket) que recibirá las peticiones de ese cliente, enviará la petición a uno de los workers que responderá en consecuencia, para luego responderle finalmente al cliente. Al terminar la conexión, este hilo finalizará.

### Dispatcher

**En Erlang, función *esperaConexion*. En C, *main* de dispatcher.c**

Escucha el puerto 8000/tcp, esperando una conexión entrante. Cuando se recibe, crea un nuevo hilo para que atienda exclusivamente a ese cliente[1].

### Proceso socket

**Función *handleCliente* (en server.erl y en dispatcher.c)**

Recibe los comandos del usuario y los traduce en peticiones al worker asignado. En cada caso espera la respuesta y la envía al cliente.

En ambas implementaciones, cada hilo, además de guardar información sobre la sesión actual (i.e: id, socket, worker asignado), también almacena una tabla de archivos abiertos en donde se realiza la traducción: descriptor del usuario → descriptor del worker.

Al tener cada cliente una asignación propia de descriptores, nos libramos del problema de que un cliente intente acceder a un archivo abierto por otro, ya que sólo podrá utilizar los descriptores que figuran en su tabla. Además, haciendo esta verificación en el proceso socket se reducen las peticiones al worker en el caso de que un descriptor no exista, pudiendo retornar un error (BADFD) inmediatamente.

### Worker

**Función *worker* (en workers.erl y en workers.c)**

Espera una petición del proceso socket o de otro worker. Se decidió no hacer distinción *a priori* de los mensajes entre workers de los mensajes entre workers y procesos socket: Para ambos casos se utilizan los mismos.

La función se encarga de procesar la petición y/o derivarla a otro worker. Hay operaciones que requieren ciclar alrededor del anillo (Por ejemplo, LSD, OPN, CRE, DEL) y otras pueden ser redirigidas (si se requiere) a un worker en particular (como es el caso de REA, WRT o CLO).

La comunicación entre workers es asíncrona: Si un worker depende de otro no se bloquerá esperando la respuesta, sino que continuará procesando peticiones. Esto es posible gracias a que

[1] En Erlang esto no es literalmente así, sino que el hilo actual atenderá al cliente y se crea un nuevo hilo para que escuche al puerto

la respuesta acarrea suficiente información como para poder procesarla posteriormente en el tiempo.

Esta función también se ocupa del manejo del sistema de archivos real (ver Puntos adicionales). Para la implementación en C se decidió encapsular estas tareas en un archivo aparte (*filelist.c*), cuya interfaz provee funciones para la apertura, el cierre, la modificación, la lectura y la eliminación de archivos, además de actuar de contenedor de los archivos presentes en el worker.

Se almacena una lista con los archivos del worker, una con los descriptores abiertos, información sobre quien es el proximo worker en el anillo y una lista de nombres reservados.

La lista de archivos abiertos que almacena es una tabla de indirección, pudiendo determinar, dado un descriptor en particular, si el archivo es local o debe redirigirse al solicitud a un worker remoto.

La lista de nombres reservados se utiliza en el proceso de creación de un nuevo archivo, para resolver el problema de que un archivo sea creado al mismo tiempo en dos workers distintos. El sistema se comporta de la siguiente manera:

Ante una petición de crear un archivo pueden darse distintos escenarios:

- Puede que el archivo exista localmente (i.e, en el mismo worker), en este caso se rechaza directamente.
- Puede que el archivo ya este reservado (por otro cliente) para la creación, tambien será rechazado.
- Puede que el archivo exista en otro worker, por lo que se agrega a la lista de reservados, y se consulta, a través del anillo, a los demás workers. Aquí pueden darse mas situaciones:
  - Que el archivo exista realmente en otro worker, por lo que le comunicará inmediatamente al worker original que el archivo ya existe y éste le comunicará al proceso socket la situación.
  - Que el archivo esté reservado en otro worker, por lo que se deberá decidir cual de los dos es el que efectivamente creará el archivo. Para esto se *'desempata'* comparando los PIDs: quien tenga mayor PID se impondrá sobre el otro.
  - Que el archivo no exista ni este reservado en ningun otro worker: oportunamente el worker que inició el ciclo en el anillo recibirá su propio mensaje, detectando así esta situación y dando curso a la creación efectiva del archivo.

## Semántica de los mensajes

En Erlang: `{Origen, Cliente, Id, Tipo, Operacion, Datos}`

En C:

```
typedef struct {
    mqd_t cliente;
    mqd_t origen;
    int id_origen;
    TCode tipo;
    OpCode operacion;
    int arg0;
    int arg1;
    char* arg2;
} Message;
```

Los campos `Origen`, `Cliente`, `Tipo` y `Operacion` coinciden en su semántica en ambas implementaciones:

- `Origen`: Indica quien originó el mensaje (puede ser un worker o un proceso socket)
- `Cliente`: Es siempre un proceso socket. Es quien originó el pedido y es el que espera la respuesta final.
- `Tipo`: Tipo del mensaje: REQUEST (indica un pedido), OK (informa finalización con éxito), ERROR (informa un error)
- `Operacion`: Indica el verdadero motivo del mensaje: OPN, CRE, CLO, REA, WRT, LSD, DEL, BYE

El campo `Id` de Erlang es el número identificador del mensaje dentro del proceso socket, el worker no altera este número, ya que se utiliza para diferenciar el pedido actual de los pedidos que ya expiraron y que deben descartarse.

El miembro de estructura `id_origen` en C se utiliza para la comunicacion entre workers. Identifica que worker originió el mensaje. En la comunicación Cliente ↔ Worker, `id_origen` = -1.

`Datos`, `arg0`, `arg1`, `arg2` dependen del tipo del mensaje y de la operación asociada.

## Puntos adicionales

Los puntos adicionales elegidos para este trabajo fueron:

- **Implementar el FS con el sistema de archivos local** (En ambas implementaciones): Los archivos se almacenan en las carpetas`"filesystem/#"` donde `#` corresponde al número de worker correspondiente (Cada worker tiene un directorio particular). Las implementaciones comparten esta carpeta, por lo que los archivos se comparten. Además, al inicializarse el sistema, cada worker revisa su directorio particular y carga la lista de archivos presentes allí, por lo que son persistentes a lo largo de distintas ejecuciones.
- **Mensajeria tolerante a fallas** (Sólo en Erlang):
  En la implementación de Erlang, el proceso socket esperará la respuesta una determinada cantidad de milisegundos (100 en este caso, modificables mediante la macro TIMEOUT, línea 7 de server.erl). Si la respuesta arriba pasado el plazo, será descartada, exceptuando las correspondientes a una llamada a OPN o CLO, que modificarán la tabla de archivos abiertos para que el sistema se mantenga consistente.