Sistemas Operativos

Trabajo Práctico Nro. 1: IPC (Inter Process Communication)

Grupo 7

Integrantes

Integrante	Legajo
Lautaro Gonzalez	54315
Martin Goffan	55431
Eric Horvat	55564

Índice

Introducción

En este trabajo práctico planteamos simular un sistema de archivos básico en base a dos tipos de IPC: Sockets y FIFOs. Además, se implementa una base de datos para el manejo de usuarios y localización de archivos. Finalmente, implementamos también un servidor de logging, que se comunica con nuestro servidor a partir de una cola de mensajes y un monitor de los servidores que se comunica mediante shared memory.

Instrucciones

Para compilar el trabajo práctico, se debe indicar si se desea utilizar FIFO o sockets, y si se desea ejecutar el servidor de logging. El comando de compilación es:

```
make clean
make FIFO=1/SOCKET=1 [LOGGING=1] all
```

Para ejecutar el server, se utiliza el comando: `./server.bin`. Así mismo, para ejecutar el cliente, el comando es: `./client.bin`. Finalmente, para ejecutar el monitor se utiliza el siguiente comando: `./monitor.bin`.

Cliente

El cliente tiene una única función: recibir instrucciones por línea de comandos, interpretarlas, hacer un pequeño control sobre los datos y pedirle al servidor que los procese. Luego, espera la respuesta, y se la muestra al usuario.

Para tener un código más prolijo, se delegó a una librería `commands.h`, que se encarga de parsear los comandos recibidos, y aplicar controles como chequear si está conectado, si ya inició sesión, si los argumentos son los correctos, o si se llama a la ayuda.

Cada comando tiene su función que además de aplicar los controles ya mencionados cuando corresponda, manda la info al servidor, espera la respuesta, los procesa en caso de ser necesario y le devuelve el control al usuario.

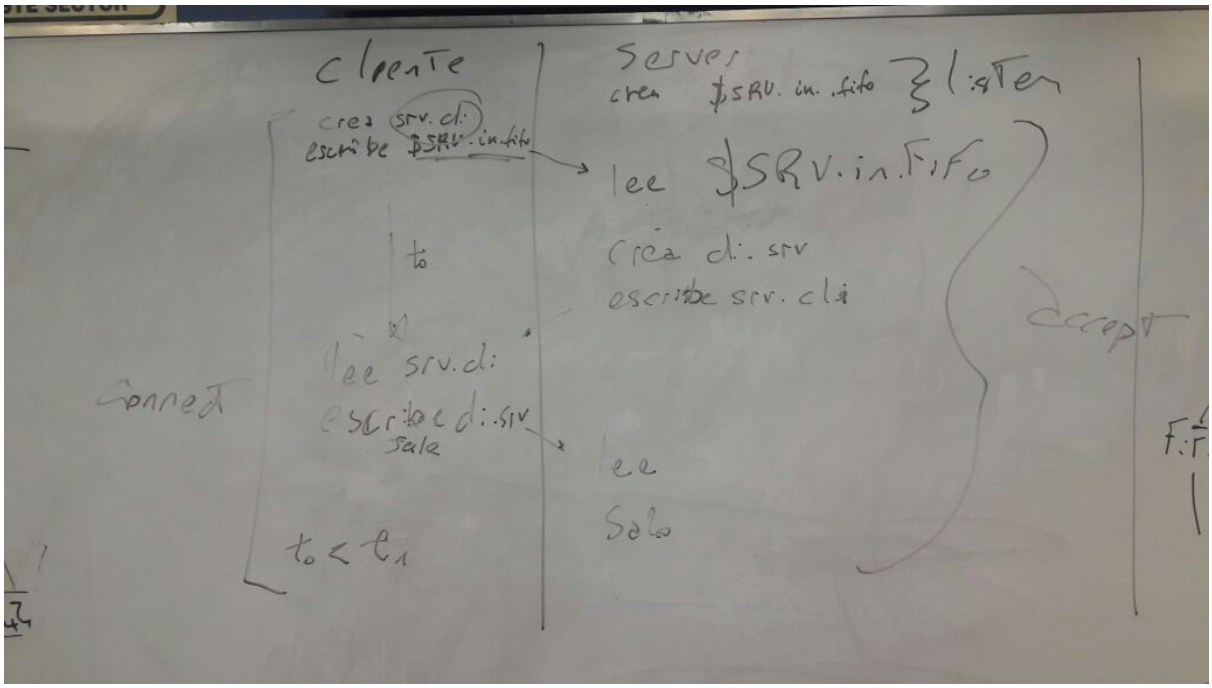
Librerías

Comunicación

Esta librería se encarga de abstraer tanto al cliente como al servidor de cómo comunicarse entre sí. Se definen qué funciones deben aplicarse, y la implementación de sockets y fifos se encargan de su aplicación.

En un principio, fue complejo abstraer ambos protocolos bajo una misma librería, por lo que se tuvo que pensar de manera más genérica como modelar este problema. Se decidió de la siguiente manera `comm_listen` se encarga de iniciar el IPC correspondiente (en este caso es FIFO o sockets, pero se pensó para que sea cualquier tipo de IPC) en el servidor. En el caso de sockets, hace el `socket`, `bind` y `listen`; en el caso de FIFO, crea un FIFO de nombre específico según cual sea el nombre del servidor, así el cliente sabrá cual es el FIFO a indicar que quiere conectarse, con solo saber el nombre del servidor. Este proceso se hizo basado en el threeway-handshake de TCP.

Luego, el servidor escuchará, según el IPC, solicitudes de nuevas conexiones en la función `comm_accept`; en caso de sockets, en el puerto, en el caso de FIFO, en el FIFO de nombre del servidor. En el caso de sockets, se hace simplemente un `accept`, en caso de FIFO se modela un three-way-handshake con el cliente, validando mediante “echos” que la información enviada sea la misma que la se recibe para verificar que del otro lado no se haya corrompido la información.



Del lado del cliente se hace `connection_open`, que en el caso de sockets, tan solo crea el socket y hace `connect`, mientras que en el caso del FIFO, tiene que avisar al FIFO de nombre del servidor que quiere conectarse, crear su FIFO de lectura, esperar que el server responda, indicando que creó el FIFO donde el cliente puede escribir, le aviso que puedo escribir.

Finalmente hay un `comm_send_data`, y `comm_send_async_data`, para el envío de datos; un `comm_receive_data` para recibirlos y un `connection_close` para cerrar los IPC correspondientes.

Serialización

Esta librería se encarga de abstraer la capa de marshalling y unmarshalling tanto al servidor como al cliente. Se decidió utilizar JSON para esto, por lo que las funciones se encargan de convertir los datos a ser enviados a formato JSON en la fuente, y que en el destino se logre volver a convertirlos en datos para ser procesados.

El mayor problema que encontramos en esto fue algo respecto a lo que se puede enviar. Los archivos, en su gran mayoría, contienen 0s (ceros) en su contenido, y su traspaso traía complicaciones. Para esto se decidió codificar el mensaje que se está mandando y decodificarlo en el destino, incluyendo este algoritmo en la capa de marshalling.

Para evitar utilizar los '\0', que causan el cierre de los FIFO, y se envían archivos de contenido desconocido, decidimos utilizar un esquema basado en Multipart Content-Type (RFC 1341). Este consiste en agregar un texto que es el substring más corto que no aparece en el contenido a enviar, tanto en el comienzo como en el fin del archivo, para indicar los límites del mismo.

Asumimos que `json-c` esta instalado en el sistema. Para instalarlo:

- En OSX: `brew install json-c`
- En Linux: `apt-get install libjson0 libjson0-dev`

Logging

La librería de logging se utiliza para abstraer la comunicación con el servidor de logging, logserver de ahora en adelante, de nuestro servidor. De esta manera únicamente se tiene que arrancar el servicio de logging a partir de `init_mq`, y luego establecer el tipo de log cada vez que se desea según las funciones de log que se encuentran en la librería `log_helpers`, que agregan un tag al mensaje y lo mandan a la cola de mensajes.

Al tratar de implementarlo, a partir de cola de mensajes, nos encontrábamos que la misma no se podía abrir ya que no había sido creada, cuando esto era tarea del logserver. Para solucionar esto, decidimos que si el error era que la cola de mensaje no estaba creada, inicie el logserver.

Por cuestiones de funcionalidad, decidimos asumir que el logserver no va a estar corriendo cuando se inicie el servidor de archivos. De esta manera, `init_mq` se encarga de forkear, e iniciar el logserver, y de esta manera este se encuentra corriendo desde casi el mismo momento que el servidor de archivos. El logserver se encarga de crear la cola de mensajes, leer de la misma, y agregar a un archivo lo que recibe.

Algo a tener en cuenta es que no es posible compilar las colas de mensajes de POSIX en OSX, por eso se incluye la opción de incluir o no el servicio de logging al momento de compilar el trabajo práctico.

SQLite

La librería de base de datos abstrae la comunicación con el servidor de base de datos, DBserver a partir de ahora, del servidor de archivos. De igual manera como el logserver, solo se debe iniciar el servicio del DBserber con la función `open_sql_conn`, y este esperará por medio de un pipe las sentencias de SQL, y responderá por otro pipe las posibles respuestas seguidas de un string indicador de que la consulta fue ejecutada. Se asume que el archivo de base de datos no está abierto por ningún otro software y se puede abrir con privilegios de escritura.

Para acceder a este, se deben utilizar las funciones que se encuentran en `sql_helpers`, que ya están diseñadas para las operaciones que se necesitan ejecutar en la base de datos, abstrayendo al servidor del manejo de estas operaciones. Estas funciones, a su vez, utilizan las funciones que se encuentran en la librería sqlite, que abstrae de la creación de strings, siendo solo necesario indicar que dato de la consulta se desea establecer.

Los problemas al momento de implementar este servicio fue tener una única variable que contuviera los datos de la conexión con el DBserver, que logramos solucionar haciendo a esta variable con el modificador `extern`. Siguiendo con los problemas encontrados, el manejo de datos fue uno de ellos por dos razones: primero, como obtener los datos de un solo string, que se solucionan con debidas funciones que crea un vector de argumentos; segundo, los límites de las consultas, como por ejemplo la cantidad de columnas o el largo máximo de la misma consulta, decidimos indicar esto a través de constantes y asumir que no superan ese límite.

Se asume que sqlite está instalado en el sistema. De no tenerlo en el sistema, descargarlo de la pagina http://www.sqlite.org/download.html, e instalarlo. O realizando `apt-get install libsqlite3-dev`.

Monitor

El monitor se encarga de mostrar el estado de los servidores, que le avisan al mismo mediante shared memory. Sea este estado entre los siguientes:

- OK
- WARN
- ERROR
- IDLE
- DOWN

Se utilizó la librería ncurses para poder mostrar las actualizaciones de los servidores en una misma pantalla. Para instalarla:

- En Linux: `apt-get install libncurses-dev`

Los estados mencionados anteriormente se representan mediante colores en la pantalla, donde se muestran todos los workers del servidor, con sus threads o responders correspondientes.

Con el mismo patrón que la librería de logging se arranca el servicio de monitor a partir de `init_monitor`, y luego establecer el tipo de estado cada vez que se desea según las funciones de monitor que se encuentran en la librería `monitor_helpers`.

A nivel implementación, el monitor muestra información a partir del momento que fue iniciado. Para que la información sea verídica, es recomendable iniciarlo antes de iniciar el servidor.

Servidor

El servidor tiene un clara función: recibir datos del cliente, procesarlos y responder. Obviamente necesita de otras herramientas ya mencionadas como el DBserver, o el logserver.

Para esto, el servidor inicia todas las herramientas que necesita, y espera conexiones. Cuando una conexión llega, crea un hijo (slave) que se encargará de responder en este IPC, mientras que él (master), sigue esperando conexiones. Este hijo espera solicitudes de este cliente en específico y crea un thread por cada una de ellas, ya que esta puede ser asincrónica.

Al igual que el cliente con `commands.h`, el servidor tiene la librería `responder.h`, la cual tiene las funciones para parsear las solicitudes y luego, ejecutar según lo que le sea pedido. El thread es aquel que se encarga de esto, y de aplicar los controles necesarios sobre los datos recibidos.

El servidor toma su configuración de un archivo JSON por defecto o se puede indicar otro por línea de comando con la opción `-c`.

Fuentes Utilizadas

- man pages
- https://www.sqlite.org/docs.html
- http://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/printw.html#PRINTWCLASS
- https://github.com/Jeshwanth/Linux_code_examples/tree/master/POSIX_shm_sem
- La cátedra
- http://www.gnu.org/software/libc/manual/html_falsede/Example-of-Getopt.html#Example-of-Getopt
- StackOverflow