Universidad Carlos III

Sistemas Distribuídos

Curso 2023-24

Práctica 1

Colas de mensajes POSIX

Ingeniería Informática, Tercer curso

Adrián Fernández Galán (NIA: 100472182, e-mail: 100472182@alumnos.uc3m.es)
César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)

Prof . Félix García Caballeira
Grupo: 81

Índice

1 Introducción

El desarrollo de este proyecto consiste en implementar una aplicación cliente-servidor, donde los diferentes clientes podrán guardar información en tuplas a través del servidor, de forma transparente. Para que esto pueda darse se pide que la comunicación entre los clientes y el servidor se de a través de colas POSIX. Es importante destacar que para el tratamiento de las solicitudes el servidor será concurrente, utilizando hilos.

2 Diseño

La aplicación constará de dos partes diferenciadas: los clientes y el servidor.

2.1 Cliente y Biblioteca dinámica

El cliente estará formado por dos partes, el main de cliente, que corresponde con el fichero *src/cliente.c*, y la comunicación con el servidor, que corresponde con el fichero */src/clave.c*. Esta último se compilará como una librería dinámica *lib.so* de forma que la comunicación sea independiente de la implementación de cliente. De esta forma, logramos una aplicación que, en caso de cambiar la forma de comunicación o implementación de los servicios no sea necesario volver a compilar siempre que se respete la interfaz actual.

Para compilar el archivo *claves.c* se ha empleado el siguiente comando en un archivo *Makefile*.

```
gcc -c -fPIC -shared -o lib.so claves.c
```

El cliente podrá llamar a las las diferentes funciones ofrecidas por la interfaz, de esta manera se podrá interactuar con las distintas tuplas almacenadas. Los servicios, no obstante, estarán implementados en el servidor de forma transparente al usuario. En esta implementación cada cliente creará una peticición que corresponda con la funcionalidad invocada y creará una cola POSIX que le corresponde, donde se recibirá la respuesta, y abrirá la cola POSIX que le corresponde al servidor, donde volcará su petición y procederá a esperar la respuesta.

2.2 Servidor

En el servidor se encuentran las funcionalidades que se encargan de la comunicación con el cliente, es decir *src/servidor.c*, y la correspondiente implementación de las tuplas, es decir *src/imp_clave.c*.

2.2.1 Estructura del Servidor

En cuanto a la parte de comunicación con el cliente, el servidor abre su cola POSIX y espera de forma indefinida a que los clientes realicen sus peticiones. En el momento en el que llega una petición el servidor analizará la petición y llamará a la funcionalidad que el cliente haya pedido. Cuando la funcionalidad se haya realizado se volcará en contenido de respuesta en la cola de cliente. Para la implenetación de las distintas funcionalidades se ha optado por guardar las tuplas en ficheros, lo que nos permite acceder a las tuplas en distintas sesiones, ya que se guardan en la memoria permanente. Para poder crear y destruir simplemente creamos los ficheros y los borramos, y para leer y escribir abrimos el fichero y volcamos los datos de manera binaria.

Implementación en el servidor

Como hemos mencionado las tuplas se almacenan en ficheros, por lo que la implementación se basará en las interfaces que C nos proporciona para gestionar ficheros. Vamos a profundizar cómo hemos usado estas interfaces durante la implementación de cada una de las funciones:

- `init()`: Para la función *init* creamos el directorio *tuple*, en el que se encontrarán todas las tuplas, en caso de que no exista y en el caso de que exista se utilizan
- `set_value()`: Para la función *set_value* escribimos en este orden la clave, el string, el tamaño del array y el array de doubles siempre y cuando no exista la tupla que corresponda con la clave proporcionada, a traveś de *fwrite*.

- `modify_value()`: Para la función *modify_value* escribimos en el mismo orden que para *set_value* pero siempre y cuando exista la tupla que corresponde con la clave proporcionada, a traveś de *fwrite*.
- `get_value()`: La función *get_value* realiza la lectura de la misma manera que ocurre en el *modify_value* pero en vez de utilizar *fwrite* utilizamos *fread* para leer todo el fichero
- `delete_key()`: La función *delete_key* realiza un unlink del fichero que corresponde con la clave proporcionada
- `exist()`: La función *exist* trata de abrir un archivo con permisos de lectura y si consigue hacerlo devuelve un 1, en caso de no poder hacerlo devuelve un 0

2.2.2 Concurrencia del Servidor

Para la concurrencia del servidor se utilizan hilos por petición, lo que significa que existe un hilo principal que acepta las peticiones y crea hilos para gestionar estas peticiones. Tras crear un hilo principal se duerme hasta que el hilo creado tome el mutex y copie los valores de la petición recibida.

Existen secciones críticas en los ficheros, pero no se van a dar condiciones de carrera debido a que las escrituras son más pequeñas que el buffer de escritura de los ficheros. Además, el sistema operativo se encargará de gestionar a los hilos para que no haya entrelazamiento de escrituras simultáneas dado que los ficheros son más pequeños que una página.

2.3 Mensajes

Para el proceso de comunicación necesitamos realizar un paso de mensajes, que se vuelquen en las correspondientes colas POSIX. Para almacenar distintos valores los mensajes utilizan arrays ya que para no pueden usarse punteros debido a que los mensajes viajarán entre procesos y por lo tanto no pueden referirse a la memoria de un proceso en concreto. Para ello se han construido 2 estructuras: la petición y la respuesta

2.3.1 Petición

Para construir la peticición nos hemos basado en los argumentos de entrada que tenían todos los prototipos de las funciones. A continuación nos pararemos en cada uno de los atributos de la estructura.

- *q_client*: Este atributo es necesario para identificar el nombre de la cola del cliente que ha realizado la petición con el objetivo de que el servidor pueda devolver una respuesta al cliente.
- *key*: El valor que corresponde a este atributo se utiliza en las funciones para identificar la tupla de forma inequivoca.
- *op*: El valor que se coloca en este atributo corresponde al tipo de función con la que se debe de trabajar
- *value1*: El valor que se almacena en este atributo es la cadena que se almacena en la tupla.
- *N_i*: Esta variable de tipo entero se utiliza como longitud para escribir el array de doubles o como índice para leer este array.
- *V_value2*: Esta variable almacena el array de doubles que corresponde con el valor que se almacena en las tuplas.

2.3.2 Respuesta

Para construir la respuesta nos hemos basado en los valores que devuelven las funciones. A continuación profundizaremos en cada atributo de la respuesta.

- *success*: El valor que contendrá este atributo será un 0 en caso de que la función se haya completado con éxito o un -1 en caso de error

- *value1*: Este atributo se usará para la función *get_value* con el objetivo de devolver el string de la tupla consultada
- *value2*: Este atributo se usará para devolver el array de doubles de la tupla consultada a través de la función *get_value*
- *n*: Este atributo contendrá el tamaño del array de doubles de la tupla consultada cuando se llama a la función *get_value*

2.4 Uso de ficheros

Para almacenar las tuplas se han utilizado ficheros. Hemos decidido que estos ficheros sean binarios, ya que conocemos a la perfección la estructura que tendrán estos ficheros. De esta manera no es necesario analizar la estructura de estos, sino que simplemente hay que leer y escribir lo que es necesario. Dado el carácter de los ficheros, nos permite mantener el contenido de las tuplas entre distintas ejecuciones del servidor.

3 Descripción de pruebas

Pruebas que hemos ejecutado sobre nuestra aplicación: ejecución con varios clientes, validación de las funciones... Creo que lo mejor sería dividirlo en subsecciones, una para cada función, otra para las comunicaciones y otra para la concurrencia.

4 Conclusiones

Durante el desarrollo de esta práctica hemos comprendido el funcionamiento de una aplicación distribuida constituida por varios clientes y un servidor. Para ello nos hemos visto en la necesidad de desarrollar el código modular para que tanto el servidor como el cliente tenga su parte de comunicaciones y su parte de implementación.