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 las peticiones se tratarán de forma concurrente, utilizando procesos ligeros.

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, la comunicación y las llamadas a los servicios de la aplicación. La primera se corresponde con el fichero *src/claves.c*. Este ofrecerá la interfaz para los servicios especificados en el enunciado y se compilará como una biblioteca dinámica (*libclaves.so*)de modo que el mecanismo de comunicación pueda variar sin afectar a los ejecutables del cliente. La segunda, se corresponde con el fichero *src/cliente.c*, esta sólo contendrá llamadas a las funcines ofrecidas por la API de la aplicación distribuída.

De esta forma, logramos una aplicación distribuída en la que la ejecución de los servicios en el servidor sea transparente al usuario. Además, la comunicación será independiente de la implementación del cliente, de forma que se podrá actualizar siempre que se respete la misma interfaz sin necesidad de recompilar el ejecutable de un cliente.

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
```

En esta implementación cada cliente creará una peticición que corresponda con la funcionalidad invocada, creará una cola POSIX, donde se recibirá la respuesta, y abrirá la cola POSIX del servidor, donde volcará su petición.

2.2 Servidor

En el servidor se encuentran las funcionalidades que se encargan de la comunicación con el cliente, en el fichero *src/servidor.c*, y la implementación de los servicios para las tuplas, en el fichero *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 lanzará un hilo que la procesará, enviando al cliente la respuesta con los datos oportunos.

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 en binario.

2.2.2 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. A continuación profundizamos en 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

2.2.3 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.

Existirán secciones críticas al acceder a 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 unívoca.

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 en binarios para definir su estructura y adaptarla a la perfección al servicio. Dado el carácter de los ficheros, nos permite mantener el contenido de las tuplas entre distintas ejecuciones del servidor.

La estructura de los ficheros será la siguiente: 256 bytes reservados para texto, un número entero N de 4 bytes que se corresponderá con el número de elementos que vendrán a continuación, y N números en coma flotante de doble precisión.

3 Descripción de pruebas

Hemos realizado dos clases de pruebas para evaluar el correcto funcionamiento de nuestra aplicación. Existen tres puntos críticos en el sistema: la **comunicación** entre el proceso cliente y el servidor, la **implementación** de lo servicios sobre las tuplas y la **concurrencia** en el lado del servidor.

Estas pruebas se recojen en 2 ficheros y un *script* en *bash* contenidos en el directorio tests: tests_concurrency.c, tests_imp.c y tests_imp.sh. Han sido preparados para su fácil ejecución a partir del siguiente comando.

```
make testing
```

Las pruebas de implementación siguen una aproximación sencilla. Ejecutamos las funciones para unos casos del que conocemos el resultado final y compararemos el resultado esperado con el obtenido. Las pruebas de concurrencia y comunicación se realizan en conjunto.

Para las pruebas de comunicación simplemente comprobaremos que uno o varios clientes son capaces de enviar y recibir mensajes de un servidor. Para las de concurrencia se han realizado los siguientes tests:

Dos escritores: comprobar el acceso concurrente de dos clientes a un mismo fichero donde prevalece la última escritura.

Dos escritores creando el mismo archivo: comprobar que, al intentar crear dos veces el mismo archivo por dos clientes concurrente, uno de ellos recibe un error.

Escritor-lector: comprobar el acceso concurrente de un escritor y un lector en ambos órdenes posibles. El lector debe acceder a los valores previos a la escritura o posterior a la escritura sin darse condiciones de carrera.

Varios escritores: comprobar el acceso concurrente de 5 escritores donde debe prevalecer la última escritura.

Varios escritores y lectores: comprobar el acceso concurrente de 3 escritores y dos lectores a un mismo archivo.

Estos test imprimen por la salida estándar los datos necesarios para validar la existencia de concurrencia y la asuencia de condiciones de carrera. Además, en algunas de ellas se generan archivos de texto que nos permiten comprobar los valores almacenados en los ficheros en puntos de la ejecución.

Los ficheros de tuplas generados en estos tests cuentan con una esctructura concreta que depende del pid del proceso que lo modifica, de forma que seamos capaces de evaluar los conceptos descritos anteriormente. Todas las pruebas han resultado exitosas.

4 Conclusiones

Esta práctica ha sido nuestra primera aproximación al desarrollo de aplicaciones distribuídas. Nos ha permitido profundizar en los conceptos introducidos en la asignatura de Sistemas distribuídos, a la par que descubrir el funcionamiento de las bibliotecas dinámicas y su utilidad en esta clase de sistemas.