SISTEMAS DISTRIBUIDOS

EJERCICIO EVALUABLE 1: COLAS DE MENSAJES

VICTORIA GUZMÁN CLEMENTE, 100495844
100495844@alumnos.uc3m.es

ALEJANDRO SÁNCHEZ, 100495744
100495744@alumnos.uc3m.es

INTRODUCCIÓN

En esta práctica hemos desarrollado un sistema de guardado y manejo de claves que funciona a través de una comunicación concurrente entre un servidor, que realiza todas las operaciones, y un cliente que sólo tiene acceso a una API. Esta API, a través de un proxy, realiza las llamadas al servidor.

DISEÑO REALIZADO

Nuestro diseño contiene los siguientes elementos:

- servidor-mq.c → Contiene el loop infinito que espera peticiones y la función tratar_peticion() para ejecutarlas y contestarlas
- claves.h → Contiene headers para todas las operaciones posibles
- claves.c → Contiene el código que ejecuta las operaciones
- proxy-mq.c → Manda las peticiones del cliente al servidor
- libclaves.so → Librería dinámica para el proxy

Además, una carpeta, “DataBase”, que será creada durante la ejecución del programa y contendrá archivos de texto con las claves guardadas y sus valores.

Para el diseño del contenido de los archivos, seguimos un desarrollo progresivo que nos permitió identificar fallos antes de pasar a un sistema más complejo.

De esta forma comenzamos con un sistema simple, que sólo contenía claves.c y un main que ejecutaba y probaba todas las operaciones. Una vez que su correcto funcionamiento estaba asegurado, pasamos a la siguiente fase.

Ampliamos el sistema convirtiéndolo en distribuido. Creamos el servidor, que se ejecutaba indefinidamente esperando peticiones, y las trataba según su tipo. Además, comenzamos a desarrollar el proxy, que sólo era un prototipo que simulaba usar colas. Para ello creamos la estructura “peticion” y la estructura “respuesta”, que serían utilizadas para la futura comunicación:

```
typedef struct {
    int op; //Operación a realizar
    int key;
    char value1[256];
    int N_value2;
    double V_value2[32];
    struct Coord value3;
    char q_name[256]; //Nombre de la cola de mensajes
}peticion;


typedef struct {
    int key;
    char value1[256];
```

```
    int N_value2;
    double V_value2[32];
    struct Coord value3;
    int status; // Éxito o fracaso
}respuesta;
```

Aunque este sistema por sí sólo no era funcional, nos permitió sentar las bases para el siguiente paso: las colas de mensajes.

Añadimos colas de mensajes al proxy, que con unas funciones globales (*abrir_colas()* y *limpiar_colas()*) nos permite mandar las peticiones al servidor y extraer de las respuestas todos los datos necesarios. A su vez, el servidor espera constantemente a que le llegue una petición y la trata adecuadamente, añadiendo el resultado a la cola de respuestas.

Una vez comprobado su buen funcionamiento, sólo nos quedaba añadir concurrencia para que el servidor pudiera manejar múltiples clientes al mismo tiempo.

La concurrencia añadida sigue dos pasos muy básicos, y todos ocurren en el servidor.

- Primero, hacemos que cada petición se cree como un hilo que ejecuta la función “TratarPeticiones” al que le pasaremos la struct Petición como un puntero. Luego, en la creación del hilo ponemos una variable condicional que nos confirme que el hilo fue creado correctamente y que el valor puntero mandado ha sido copiado.

- La segunda parte de la concurrencia simplemente consiste en hacer atómicas los cambios en la base de datos, conseguimos esto cerrando la operación con un mutex.

Algo a destacar del servidor, es que tanto cuando este se ejecuta, como cada vez que va a coger una petición, revisa que la carpeta de “DataBase” está. Esto lo hemos hecho así para prevenir que en medio de que un cliente esté trabajando, por error humano, o de la máquina, o de otro cliente, se borre dicha carpeta y afecte al funcionamiento del servicio.

COMPILACIÓN Y EJECUCIÓN

Para la compilación del programa hemos usado un archivo “compilar.sh” para ir haciendo la compilación de todo. No es la manera más eficiente pero ya que el programa se compila rápidamente hubiéramos perdido más tiempo haciendo compiladores solo para clientes o solo para servidor.

Importante destacar que para que el cliente funcione correctamente hay que anclar la biblioteca dinámica a la terminal correspondiente, cosa que hace el “compilar.sh” pero si usaramos otra terminal para ejecutar un cliente habría que hacerlo de nuevo.

Finalmente, la ejecución es muy básica: primero en una terminal se ejecuta el servidor como cualquier programa compilado de C, “./servidor-mq “, y en otra u otras terminales ejecutamos los clientes que queramos.

Pruebas

Para las pruebas hemos hecho dos clientes que comprueban diferentes aspectos del servidor.

Cliente1: está enfocado en comprobar que las funciones devuelven el resultado correcto, ya sea 0 o -1. Las situaciones previstas son:

- Destroy
- 2 set de la misma key
- get de esa key
- dos del seguidos
- get de key borrada
- modify de key borrada
- prueba de modify correcta

No hacemos pruebas directas del Exist ya que está implementado y probado en las otras funciones.

Cliente2 busca llevar al límite al servidor tanto en concurrencia como en instrucciones y tamaño de cola. Por ejemplo tenemos un loop que guarda 50 keys distintos sin problema. Y ejecutándose a la vez con Cliente1 podemos comprobar el buen funcionamiento de la concurrencia.

CONCLUSIONES

Este trabajo nos ha permitido entender de forma práctica cómo funciona una aplicación distribuida. A pesar de que al principio nos costó entender cómo los archivos se debían comunicar entre ellos, finalmente logramos una interacción fluida y correcta de todo el programa en conjunto.

Gracias al uso de archivos de texto, las claves se guardan de forma segura y eficiente, proporcionando un sistema eficaz que se podría escalar en el futuro con el uso, por ejemplo, de un sistema de cifrado que añadiera seguridad a la aplicación.