Universidad Carlos III

Curso Sistemas Distribuidos

Práctica 1

Curso 2024-25

Entrega: **1**

GRUPO: **80**

ID de grupo de prácticas:

Alumnos: **Jorge Agramunt / Pablo Navarro**

Correo de todos los alumnos: **100495764@alumnos.uc3m.es** y **100495879@alumnos.uc3m.es**

Indice

1. Diseño realizado

Esta práctica consiste en un sistema que implementa una comunicación mediante cola de mensajes entre un cliente y un servidor dicho genéricamente, a continuación se específica cómo está hecho el diseño de este ejercicio:

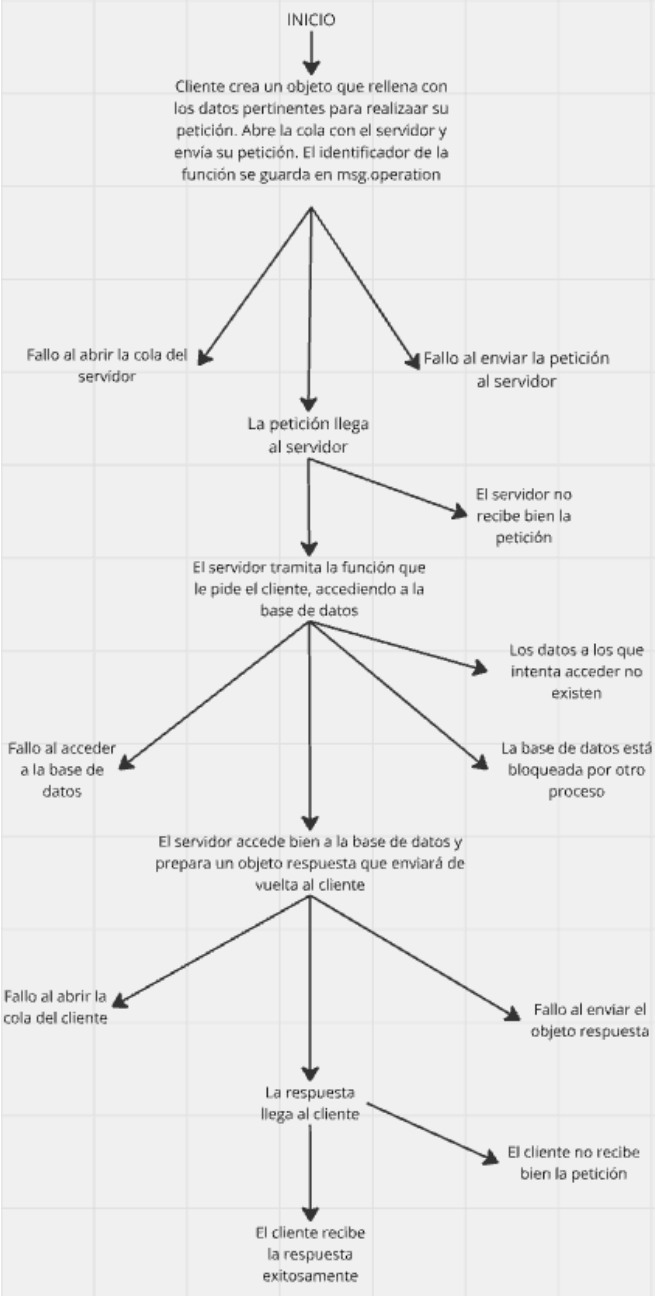
- **Cliente(app_cliente.c):**
  Llama a las funciones del API: set_value, get_value, exist, destroy, modify_key y delete_key. Una cosa interesante de esta implementación es que el cliente no sabe cómo se comunica con el servidor así que se podrían implementar un sistema realmente distribuido sin que el cliente tuviera que hacer ningún cambio. Esto se consigue gracias a la biblioteca dinámica que contiene *proxy-mq.c* que es la encargada de manejar la comunicación.
- **Proxy(proxy-mq.c):**
  Envía las peticiones del cliente al servidor mediante una cola de mensajes. Cada cliente abre una cola que es la que van a usar todos los clientes para comunicarse con el servidor (q_servidor) que tiene esta estructura: "/SERVIDOR-5764-5879". Una vez la cola está abierta con éxito, se envía la información y espera a recibir una respuesta por la cola del cliente (q_cliente) que es única para cada cliente con esta estructura "/Cola-id_cliente-5764-5879", siendo id_cliente el ID de proceso de la máquina en la que se ejecute asignado para ejecutar ese cliente.
- **Servidor(servidor-mq.c)**
  Recibe las peticiones desde la cola del servidor (q_servidor), llama a claves.c que es el que las ejecuta y espera su respuesta. Una vez tiene esta respuesta, abre la cola con el cliente (q_cliente) y envía la respuesta de claves.c al cliente de vuelta. Cada vez que un cliente se comunica con el servidor, el servidor crea un hilo para tratar la petición. Una vez completada la misión del thread se muere.
- **Base de datos(claves.c)**
  Se ha implementado una base de datos en sqlite3 para simular una base de datos lo más realista posible. Se ha diseñado esta base de datos de la manera más simple que se nos ha ocurrido para evitar errores de FK a la hora de crear o borrar filas entre diferentes tablas y simplificar su manejo haciéndolo con sólo una única tabla: "datos", que contiene las columnas "key(PK), value1, value2, value2_len, vale3_x, value3_y".
  Aquí se realizan todas las operaciones relacionadas con buscar, añadir, modificar o eliminar filas en la BD.
- **Paso de mensajes**
  A través de las colas se pasa un único mensaje que contiene una estructura Obj que tiene el siguiente formato:

```
Obj {
  int key;
  char value1[MAX_BUFFER];
  int N_value2;
  double V_value2[32];
  struct Coord value3;
  int operation;
  char q_name[MAX_BUFFER];
}
```

  El elemento *operation* es el número de operación para saber qué función ejecutar (set_value, get_value, exist, destroy, modify_key y delete_key). El elemento *q_name* es el nombre de la

cola del cliente (q_name) para que el servidor sepa por qué cola enviar la respuesta a la petición.

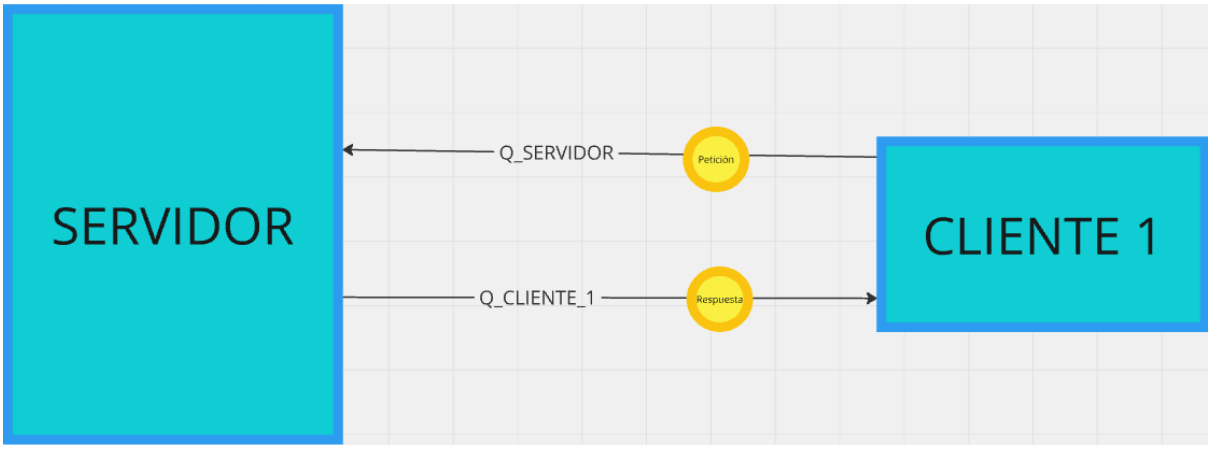
2. Diagrama de flujo



Estructura del objeto respuesta y petición

```
typedef struct Obj {
    int key;
    char value1[MAX_BUFFER];
    int N_value2;
    double V_value2[32];
    struct Coord value3;
    int operation;
    char q_name[MAX_BUFFER];
} Obj;
```

Key, Value1, N_value2, V_value2, value3 son argumentos que pasa el cliente al servidor para añadir datos en la base de datos por ejemplo.
Operation es un entero que representa por medio de un número el tipo de operacion que debe realizar el servidor. Cuándo el servidor envía su respuesta, es aquí donde se guarda la salida de la función realizada en el mismo.
Q_name es el nombre de la cola cliente que viene determinado por el pid() de cada uno de los procesos cliente.



3. Batería de pruebas

Para la realización de la batería de pruebas nos creamos varias programas clientes:

1. app-cliente.c: Este programa ejecuta un set_value, enviando una serie de valores a la base de datos para su posterior inserción en la base de datos. La key por la que se identificarán estos valores es 234.

2. app-cliente-1.c: Este programa hace lo mismo que el anterior pero la key por la que se insertan estos valores es 44.

3. app-cliente-2.c: Este programa ejecuta un get_value, el cuál guarda en las posiciones de memoria que le pasamos como argumentos, los valores que le manda el servidor en función de la key proporcionada. En este caso serán los valores que hemos introducido con app-cliente ya que la key es la misma, 234.

4. app-cliente-3.c: Este programa ejecuta un modify_value, cuyo cometido es modificar los datos caracterizados por la key 44.

5. app-cliente-4.c: Este programa ejecuta un exist, que consiste en averiguar si existen datos con la key 44.

6. app-cliente-5.c: Este programa ejecuta un delete_key, borra aquellos datos asociados a la key 234.

7. app-cliente-6.c: Este programa ejecuta un destroy, borrando la base de datos por completo.

Ya creados los distintos programas cliente, procedimos a la ejecución de cada uno de ellos por separado, lo cual resulta en éxito si los programas son tratados en el orden correcto. Por ejemplo, si ejecutamos el exist, el modify_value, el get_value o el delete_key sin antes haber hecho los set_value, las funciones no tendrán éxito ya que la base de datos carece de aquellos valores a los que las funciones tratan de acceder. Lo mismo pasa si ejecutamos

el destroy y luego tratamos de acceder a datos inexistentes. Todo esto es correcto ya que refleja el comportamiento correcto de un servidor con base de datos operativa y eficiente.

Después de ejecutar todos los programas uno por uno, procedimos a la ejecución de todos ellos al mismo tiempo. Esto lo conseguimos abriendo dos terminales, en una de ellas ejecutamos el servidor (./servidor) y en la otra todos los clientes (LD_LIBRARY_PATH=. ./cliente & LD_LIBRARY_PATH=. ./cliente1 & LD_LIBRARY_PATH=. ./cliente2 & LD_LIBRARY_PATH=. ./cliente3 & LD_LIBRARY_PATH=. ./cliente4 & LD_LIBRARY_PATH=. ./cliente5 & LD_LIBRARY_PATH=. ./cliente6). De esta forma comprobamos que efectivamente, tenemos un servidor concurrente que ejecuta las distintas peticiones de los clientes una por una y carecemos de fallos de carrera de datos.

4. Compilar código

Hay un fichero makefile, ejecutar “make” en la terminal.

5. Ejecutar código

Abrir dos terminales, en una ejecutar: ./servidor, en la otra ejecutar: LD_LIBRARY_PATH=. ./cliente.
En caso de querer ejecutar varios clientes al mismo tiempo se podría así:LD_LIBRARY_PATH=. ./cliente & LD_LIBRARY_PATH=. ./cliente1 & LD_LIBRARY_PATH=. ./cliente2 & LD_LIBRARY_PATH=. ./cliente3 & LD_LIBRARY_PATH=. ./cliente4 & LD_LIBRARY_PATH=. ./cliente5 & LD_LIBRARY_PATH=. ./cliente6

6. Conclusión

Esta práctica nos ha servido para comprender un sistema de comunicación mediante paso de mensaje por colas y una comunicación básica entre cliente y servidor, además del trato con bases de datos.

Con esta estructura, en un futuro se podrá hacer un sistema distribuido en el que se ejecuten partes del código en distintas máquinas sin necesidad de que el cliente haga nada.