Universidad Carlos III

Curso Sistemas Distribuidos

Práctica 2

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**

Índice

1. Cambios realizado

Esta segunda práctica consiste en un realizar lo mismo que en la primera pero cambiar el código para que la comunicación entre cliente y servidor se haga a través de sockets y no de colas:

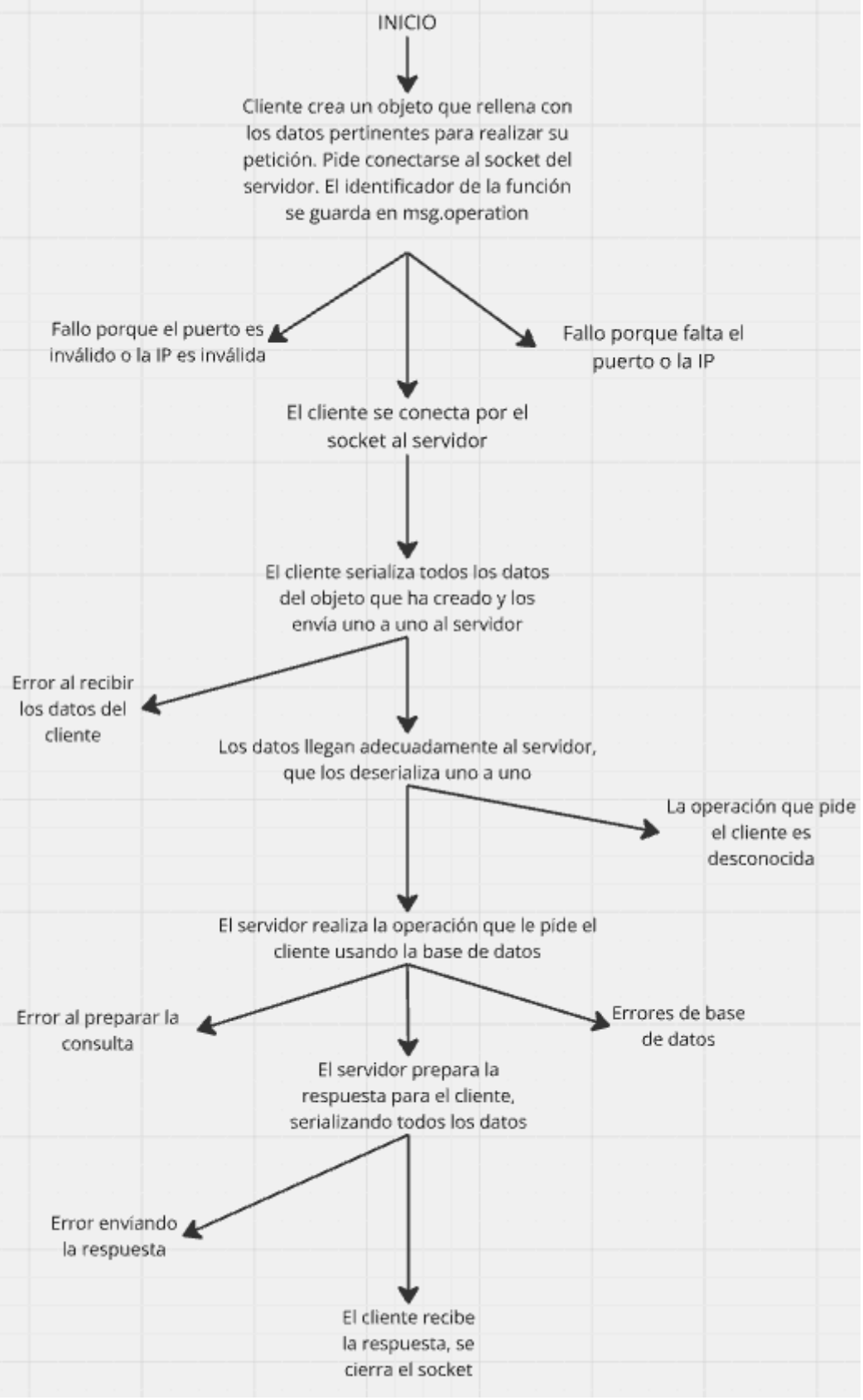
- **Proxy(proxy-sock.c):**
  Envía las peticiones del cliente al servidor mediante uso de sockets. Cada cliente abre el socket que es el que van a usar todos los clientes para comunicarse con el servidor (sockfd). Una vez el socket se inicia con éxito, se envía la información y se espera a recibir una respuesta del servidor.
  Para enviar estos datos, usamos unos métodos de serialización y deserialización que explicamos en el siguiente punto.

- **Servidor(servidor-sock.c)**
  Los recibos se realizan por socket con la función: recv_obj_fields que deserializa uno a uno todos los datos que recibe del cliente y los introduce uno a uno en una estructura objeto para realizar las operaciones necesarias. Lo hemos hecho de esta manera para reciclar la estructura objeto que teníamos ya creada y que el código se mantuviera de una forma similar a la de la práctica 1.
  Los envíos se realizan por socket con la función: send_obj_fields que envía todos los campos del objeto respuesta serializados al cliente. Operation y key como enteros de 4 bytes en formato big-endian, value1 enviando primero su longitud y luego la propia cadena de texto, N_value2 como un entero de 4 bytes y cada elemento de V_value2 como 8 bytes en formato big-endian. Por último los valores de value3.x y value3.y como enteros de 4 bytes en formato big-endian.

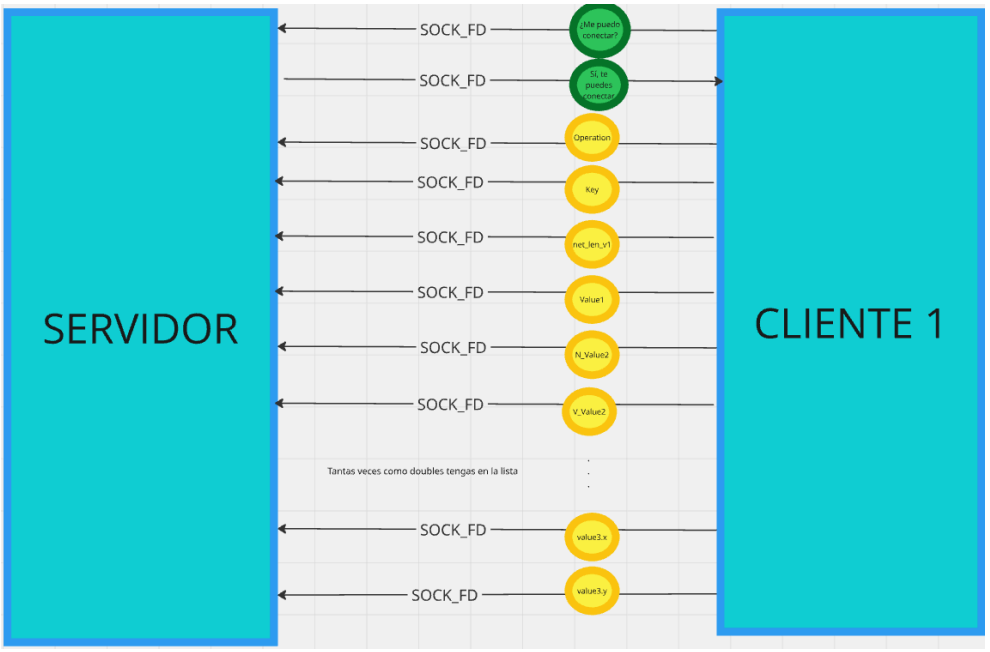
- **Base de datos(claves.c)**
  Los únicos cambios que hemos realizado en la base de datos han sido los relacionados con los mutex, ahora hacemos un lock_mutex justo antes de abrir la base de datos y se hace un unlock_mutex después de realizar las operaciones pertinentes.

- **Funciones auxiliares**
  En proxy-sock.c y en servidor.sock.c hemos creado funciones auxiliares tales como recvAll y sendAll que aseguran que se reciban o envíen exactamente n bytes a través del socket, manejando posibles interrupciones parciales en la transmisión.
  Además de eso hemos añadido funciones de conversión de double a 8 bytes en formato big-endian y viceversa: double_to_be64 y be64_todouble.

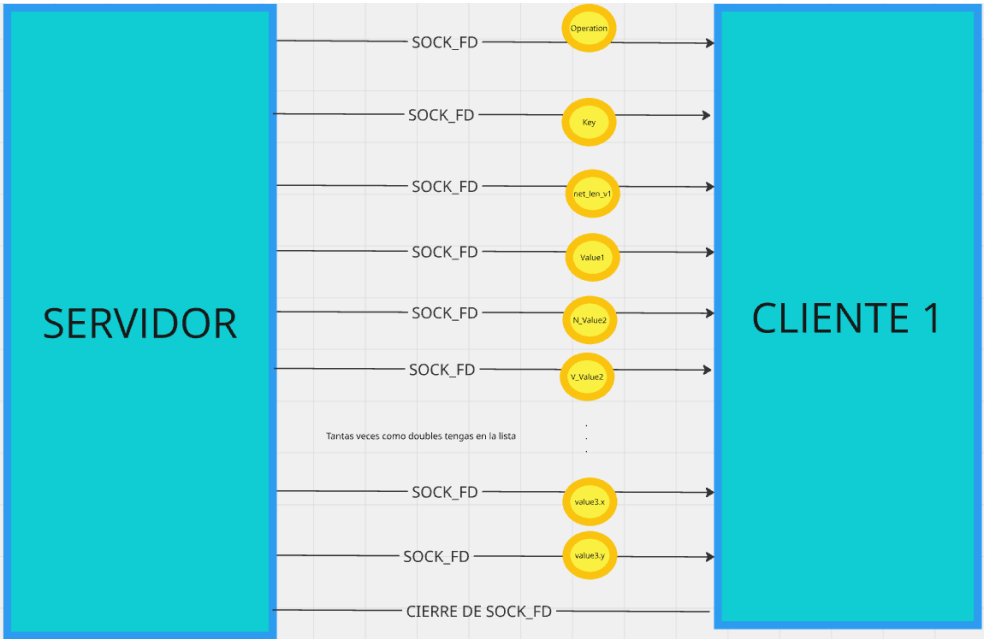
2. Diagrama de flujo



PRIMERA CONEXIÓN CLIENTE CON SERVIDOR



RESPUESTA SERVIDOR 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.

Hemos realizado pruebas de clientes queriendo insertar la misma key al mismo tiempo, siendo un cliente éxito y otro cliente error porque ya existe esta key. Lo mismo con el resto de funciones.

También hemos realizado una prueba de estrés, poniendo en el bash.sh 54 clientes al mismo tiempo, saliendo exitosa la operacion.

4. Compilar código

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

5. Ejecutar código

La ejecución del código ha cambiado, lo primero es averiguar la dirección IP usando ip addr en la terminal desde la que vas a ejecutar el servidor. Después de obtenerla debe ejecutarse el siguiente comando desde la terminal desde la que vas a ejecutar los clientes: export IP_TUPLAS=<dirección IP obtenida>, para guardar en el proxy-sock.c la IP correcta. A su vez deberá ejecutarse el comando export PORT_TUPLAS=<puerto>, donde puerto es un número entero como por ejemplo 5000.

Para ejecutar el servidor se escribirá el siguiente comando en la terminal: ./servidor <puerto>, siendo el puerto el mismo que hemos introducido antes para el proxy-sock.c

Para ejecutar los clientes nos hemos creado un archivo bash.sh que ejecuta todos los distintos clientes que se detallan en la batería de pruebas, bastará con ejecutar el comando: ./bash.sh. Si se deseara ejecutar un cliente individualmente hay que poner el siguiente comando: LD_LIBRARY_PATH=. ./<cliente>. Por otro lado, si se ejecutar clientes concretos al mismo tiempo, hay que poner el siguiente comando: LD_LIBRARY_PATH=. ./<cliente> & LD_LIBRARY_PATH=. ./<cliente> & LD_LIBRARY_PATH=. ./<cliente>

6. Conclusión

Esta segunda práctica nos ha servido para comprender las distintas diferencias entre colas y sockets a la hora de enviar mensajes. También hemos corregido los errores de concurrencia que teníamos al poner los mutex. Ahora los mutex se encuentran dentro de cada una de las funciones (get_value, modify_value…) por lo que los distintos hilos pueden hacer sus funciones a la vez de forma correcta.