Universidad
Carlos III de Madrid

Universidad Carlos III

Curso Sistemas Distribuidos 2024-25

Ejercicio 1

Curso 2024-25

Sistemas Distribuidos

Grado Ing Informática

Fecha: **16/03/2025** - ENTREGA: **I**

GRUPO: **80**

Alumnos:

Hector Álvarez Marcos 100495794

Angela Elena Serrano Casas 100475053

Introducción

Este documento contiene toda la documentación acerca del entregable 1 de la práctica de sistemas distribuidos. En ella se explican todas las decisiones de diseño tomadas, así como una explicación de cómo se ejecuta el proyecto y una detallada descripción de los 2 principales actores involucrados en el proyecto.
Esta práctica pretende simular un esquema Cliente-Servidor, en el que se nos proporcionaba un esqueleto con un archivo de cabeceras claves.h, que contiene las principales funciones que un cliente podría implementar. El sistema pretende emular una especie de "API" para el cliente, que podrá realizar distintas acciones sobre un sistema de guardado de datos realizado en un servidor. Se nos pedía implementar tanto el código asociado a los posibles clientes como el asociado al servidor, usando para ello un sistema de colas de mensajes POSIX. Todo esto desarrollado en el lenguaje de programación c.

Diseño e implementación

En esta parte explicaremos en profundidad cómo funcionan ambos elementos, tanto cliente como servidor, así como detallar el por qué de las principales decisiones tomadas para mejorar tanto el rendimiento, como la gestión de todo el sistema, en pro de realizar una implementación lo mejor posible.

Decisiones principales

Dentro del enunciado de la práctica, se nos permitía tener total libertad a la hora de escoger el método de guardado de los valores. Como se ha explicado, el objetivo era generar una especie de API para un cliente que le permitiera acceder, ver, modificar, guardar o eliminar distintos datos. Ese sistema, en bajo nivel, vendrá implementado por un servidor, en el cual teníamos la libertad de elegir cómo pretendíamos guardar los datos. Una opción fácil de implementar sería por ejemplo la creación de una lista enlazada dentro del servidor, ya que permite fácil acceso a los datos y sencillez en la implementación. Este método no obstante tiene problemas, ya que si por lo que sea el proceso que ejecuta el servidor finaliza, todos estos datos serán borrados, por lo que optamos por usar un fichero externo para almacenar los datos. Una vez tomada esta decisión, falta saber qué tipo de fichero se usaría. Optamos por usar un fichero de tipo .db, que actúa como base de datos "SQL", lo que nos permite tener un sistema de tablas con garantía de acceso atómico, gestión de duplicados, y fácil manejo a la hora de tratar los datos. Se ha utilizado el siguiente diagrama relacional para las tablas(imagen 1). Este sistema, basado en 2 tablas, se ha generado con el único propósito de guardar con facilidad el "*value_2*", ya que es una lista de entre 0 y 32 elementos. Ante la complejidad de insertar esto en una misma tabla, se ha optado por generar una tabla secundaria "*value2_all*", con referencia de FK[1] a la PK[2] de data, que es la tabla principal, y se realiza una inserción por cada elemento del array *value_2*. La PK de cada elemento corresponde con la concatenación numérica de la PK referenciada y el índice del elemento del vector. Así mismo, si la PK de data es 33 y quiero insertar 4 valores el *value2_all*, la PK de cada uno será 330, 331, 332 y 333 respectivamente.
Esto garantiza que no se dupliquen PK en la segunda tabla tampoco.

[1] FK: Foreign Key(Identificador de una fila que referencia a un identificador único de otra tabla)
[2] PK: Primary Key(Identificador único de una fila en una tabla)

Aparte de esto, la segunda tabla tiene una relación UC/DC[3], lo que garantiza que si se elimina una fila en la tabla principal, todas las referencias a su PK en la tabla secundaria se eliminarán también, así se elimina el tener que ir buscando todas las referencias a esa fila en la tabla secundaria.

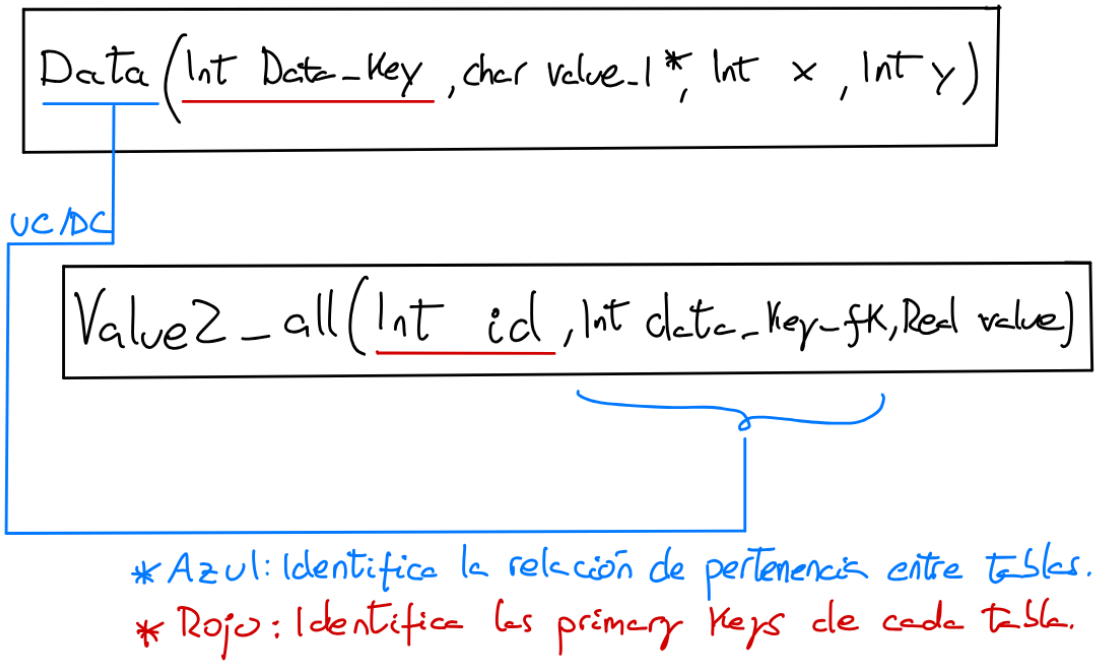

Imagen 1

Servidor

El servidor (servidor-mq.c) se encarga de gestionar las solicitudes de los clientes, operando sobre una cola de mensajes centralizada. Para garantizar la eficiencia y la capacidad de atender múltiples clientes de manera simultánea, se ha implementado un pool de hasta 25 hilos que procesan las peticiones concurrentemente. Esta decisión garantiza un uso de memoria controlado, frente a la creación de un hilo "detached" por cada petición que se realice, garantizando también así un buen tratamiento de los datos. Durante la fase de pruebas nos hemos dado cuenta de que mayoritariamente nos basta con un máximo de 2 hilos, ya que las peticiones se realizan tan rapido que da tiempo a que se liberen esos hilos, ya que el servidor siempre buscará los primeros que hayan acabado para realizar las nuevas tareas. No obstante dejamos 25 en caso de que se recibieran muchas peticiones de distintos clientes a la vez, aunque el factor limitante si se reciben múltiples peticiones es más la propia cola del servidor por culpa del sistema operativo en sí que el número de hilos en nuestra pool.
Hemos implementado para la reutilización correcta de cada hilo un join no bloqueante, mediante la función pthread_tryjoin_np. Esto junto con la realización de la lectura de mensajes no bloqueante también garantiza que mientras no se estén recibiendo peticiones, se pueda realizar la "recogida" de aquellos hilos que hayan terminado su tarea, garantizando la disponibilidad completa del máximo número de hilos dentro de la pool.

[3] UC/DC: Update Cascade, Delete Cascade. Se actualizan los valores referenciados en la segunda tabla según lo que pase con la tabla principal.

El proceso comienza cuando un cliente envía una solicitud a través de la cola de mensajes. El servidor extrae este mensaje utilizando la función mq_receive y lo interpreta para determinar la operación a realizar *(set_value, get_value, modify_value*, ...). Posteriormente, se invoca la función correspondiente definida en la API, y tras completar la operación, el resultado es enviado de vuelta al cliente mediante la función *mq_send*, asegurando así la correcta transmisión de la respuesta a través de la infraestructura de colas de mensajes.

Para la gestión de datos, el servidor hace uso de las funciones definidas en *treat_sql.c*, que encapsulan las operaciones de consulta sobre la base de datos. Este módulo implementa funciones como recall_row_data y recall_row_value2_all, que permiten recuperar los datos de una clave específica. Para los posibles fallos dados en la comunicación y ejecución de las operaciones, el servidor implementa mecanismos de control de errores basados en el uso de códigos de retorno en todas las funciones clave.

Cliente

El cliente (*proxy-mq.c*) se encarga de proporcionar una interfaz sencilla para que los usuarios interactúen con el sistema sin necesidad de preocuparse por la comunicación con el servidor. Para ello, encapsula las funciones de la API (*set_value, get_value, modify_value, delete_key, exist y destroy*), permitiendo que el cliente realice operaciones sobre las claves almacenadas sin conocer los detalles del intercambio de mensajes.

Cada cliente establece su propia cola de mensajes exclusiva, a la cual el servidor enviará las respuestas correspondientes. De este modo, se garantiza que cada solicitud reciba una respuesta sin interferencias de otros procesos en ejecución. Las funciones implementadas en proxy-mq.c estructuran los mensajes en un formato estándar antes de enviarlos a la cola de mensajes del servidor, asegurando compatibilidad con el sistema.

Cabe mencionar que a petición del enunciado, el cliente se ha empaquetado en una biblioteca dinámica denominada libclaves.so de esta manera se obtiene una notable mejoría en la portabilidad además de facilitar su uso en otras aplicaciones.

Compilación

Para compilar este proyecto se ha usado un archivo “makefile” ubicado en el directorio principal. Dicho archivo se usa para automatizar la compilación o limpieza de la aplicación.
En nuestro caso hemos empezado definiendo una serie de variables como que compilador se iba a usar, en este caso gcc, las distintas flags necesarias o la ubicación de los includes, es decir, en qué directorio tenemos todos los archivos de cabeceras. También definimos variables necesarias para localizar la ubicación de todos los archivos. Finalmente se realiza una compilación de todo esto, teniendo en cuenta que se tiene que compilar una librería dinámica y que la tienen que usar todos los clientes. Al final del todo se incluye una regla para que si se ejecuta el comando make clean, se realiza un borrado de todos los objetos y los ejecutables generados, así como una limpia del archivo generado como base de datos si este existiera.

Pruebas

Guernika

Primeramente hemos de indicar una decisión que tuvo que ser tomada al realizar las pruebas del proyecto en el cluster “guernika”. Al intentar ejecutar nuestro código del servidor nos encontramos con un problema con la apertura del archivo para la base de datos. Abría bien, pero no dejaba realizar ninguna operación sobre él, mostrando el error de “database is locked”. Esto suponemos que puede ser debido al uso del sistema de archivos compartidos dentro del servidor, aunque desconocemos completamente su causa. Por ello decidimos ubicar la base de datos en el directorio temporal /tmp. Desconocemos cada cuanto se elimina esto ni si es una solución completa, pero era la única forma que encontramos de garantizar el correcto funcionamiento de nuestra aplicación. Cabe destacar también que a veces en la terminal donde compilamos el proyecto, si ejecutamos desde ahí el código, daba a veces lugar a error no generando la cola del servidor o del cliente. Esto lo solucionamos simplemente usando otra terminal conectada a “guernika”, pero nos gustaría que quedara constancia de este problema, por si en algún momento se enfrenta a este problema también. Para aportar más información al problema, hemos notado que si ejecutamos la conexión desde ssh con el comando “ssh -l a0495794 guernika.lab.inf.uc3m.es” a veces se conecta a una versión de guernika -gpu y otras veces no.. Solo en los casos que incluye el “gpu” nuestro codigo funciona. Para el correcto funcionamiento completo se tiene que realizar una conexión con el comando “ssh -l a0495794 guernika-gpu.lab.inf.uc3m.es”. Desconocemos el por qué, pero si ejecutas desde una version no gpu no da error al abrir la cola, pero no la crea, lo que ocasiona que el codigo no funcione correctamente.

Para evaluar el correcto funcionamiento del sistema en diferentes condiciones se han realizado una serie de pruebas con el objetivo de profundizar y examinar los límites de nuestra implementación. Estas pruebas se encuentran distribuidas en varios archivos, cada uno con un propósito específico.

Clientes Generados

Hemos generado 3 clientes distintos, pero con una estructura prácticamente idéntica. Realiza 2 set_value, un get_value, un modify_value, un delete_key y un destroy, todo en ese orden. Generamos 3 clientes similares para luego poder realizar pruebas de estrés con más facilidad.
A parte de esto, hemos creado una implementación de un “cliente infinito”, que va pidiendo por terminal que operación se desea realizar y los campos necesarios para dicha operación todo repitiendo en bucle mientras se desee.

Pruebas de estrés

Hemos querido realizar pruebas de estrés para evaluar el comportamiento de nuestro sistema, implementando un archivo bash que ejecuta de forma simultánea distintos clientes, con el objetivo de analizar y llevar al límite a nuestro servidor.
Hemos dividido las pruebas de estrés en 2 partes:

test_easy_stress.sh

Implementa una prueba de estrés automatizada que simula un bajo volumen de operaciones simultáneas, incluyendo inserciones, modificaciones y eliminaciones de claves. Se analizaron métricas de rendimiento y la capacidad del sistema para manejar grandes volúmenes de solicitudes sin degradación significativa.

test_hard_stress.sh

Implementa una prueba de estrés automatizada que simula un gran volumen de operaciones simultáneas, incluyendo inserciones, modificaciones y eliminaciones de claves. De nuevo, se analizaron métricas de rendimiento y la capacidad del sistema para manejar grandes volúmenes de solicitudes sin degradación significativa. Esta prueba se realiza con el único proposito de probar que el servidor no realiza goteo de información por el gran volumen de mensajes que llega a recibir, denotando como factor limitante al propio sistema operativo

Durante las pruebas de carga, se observó una limitación del sistema al intentar manejar más de 50 clientes simultáneamente(Esta prueba depende del sistema en el que se realice, en algunos sistemas soportaba hasta 50 clientes, en otros a los 18/20 paraba). Aunque las operaciones básicas de inserción, recuperación y modificación se realizaron correctamente bajo carga moderada, cuando se superó este umbral, se presentó un problema en el que la cola de mensajes alcanzaba su capacidad máxima, lo que impedía la recepción de nuevas solicitudes y generaba un mensaje de error indicando que la cola estaba llena.

Conclusión

La implementación desarrollada transforma con éxito la aplicación monolítica en un sistema Cliente-Servidor basado en colas de mensajes POSIX. Se ha conseguido una arquitectura eficiente que permite la concurrencia de múltiples clientes, asegurando la persistencia y consistencia de los datos.

El sistema ha sido sometido a un plan de pruebas exhaustivo, confirmando su correcto funcionamiento en condiciones normales y ante escenarios de error. La modularidad del diseño y la implementación de una biblioteca dinámica facilitan la integración y reutilización del código en otras aplicaciones.

El desarrollo de esta práctica nos ha ayudado a comprender mejor los conceptos abordados en la asignatura, permitiéndonos no solo reforzar la teoría estudiada en clase, sino también enfrentarnos a los retos que supone llevar a la práctica estos conocimientos. Implementar un sistema distribuido real nos ha permitido experimentar con la comunicación entre procesos, el uso de colas de mensajes y la gestión de concurrencia, aspectos fundamentales en la materia.

En definitiva, esta práctica nos ha servido para consolidar lo aprendido en clase, pero sobre todo, para desarrollar una perspectiva más realista y aplicada de los sistemas distribuidos. Ha sido una experiencia desafiante y a su vez enriquecedora.