Universidad Carlos III

2023-24

Ejercicio 1 - Sistemas Distribuidos

Pablo García Rius 100428976

José Antonio Barrientos Andrés 100451290

Curso 2023-24

Diseño

Servidor

Estructura de datos

Se especifica en el enunciado del ejercicio que el único requisito para la selección de la estructura de datos es que no limite el número de claves que se puedan almacenar, por tanto, nuestra elección ha sido un árbol binario, en esta implementación del árbol binario, cada nodo está compuesto por la información de la tupla a almacenar, es decir, una key, la cadena de caracteres, la longitud N del vector y el vector de doubles. Además de punteros a los nodos izquierdo y derecho que salen de él.
La estructura utiliza como clave de ordenación la key, ya que es irrepetible, y hace como cualquier árbol binario, compara el valor de la key y si es menor buscará en el nodo izquierdo, si es mayor en el derecho, hasta encontrar un hueco vacío en el que colocar el nuevo.
Esta estructura de datos nos proporciona una cantidad ilimitada de nodos, además de una velocidad bastante considerable a la hora de hacer una búsqueda.

Estructura del código

Se han declarado unas constantes que corresponden a los códigos de operación que el "cliente" puede mandar, estas son un entero que va del 0 al 5, siendo los valores: INIT, GET, SET, MODIFY, DELETE y EXIST.
Lo primero que hace el servidor es definir la cola en la que los clientes van a mandar las peticiones, hemos utilizado para los mensajes de los clientes, un struct Mensaje, que contiene: el código de operación, la key, la cadena de caracteres, la N, el vector, y otra cadena de caracteres llamada queue, en la que el cliente manda el nombre de la cola que ha abierto para esperar la respuesta del servidor.
Después de abrir la cola, se inicializan las variables tipo mutex y condiciones.
El servidor entra en un bucle infinito en el que: espera una petición de un cliente, hace un switch para mandar la petición a la función que corresponda, crea un hilo para el cliente con la función correspondiente al código de operación que mandó. Si el código de operación está fuera de rango, se descarta la petición.

Concurrencia

Hay una función por cada operación que el cliente puede hacer sobre el servicio de tuplas, pero la estructura se comparte prácticamente.
Se pasa a la función el struct Mensaje que mandó el cliente (inciso: no todas las funciones necesitan todos los parámetros del struct Mensaje, por tanto, este puede tener parámetros vacíos, dependerá de la operación que quiera hacer el cliente) el hilo recoge el mutex para

copiar el valor del mensaje a una variable local, una vez copiado, suelta el mutex y notifica al hilo principal de que ha terminado de copiar para que pueda crear el siguiente hilo. Después, procede a ejecutar la función correspondiente sobre la estructura de almacenamiento, para evitar que se introduzcan varias tuplas a la vez y haya incongruencias, o problemas en la ejecución, el hilo adquiere otro mutex mientras hace la operación sobre la estructura, y lo suelta cuando ha terminado.

Respuesta

Por último, para mandar la respuesta de vuelta al cliente, se utiliza el parámetro queue que este incluyó en la petición y se abre la cola de mensajes como escritura. Se utiliza otra estructura auxiliar que hemos creado llamada Respuesta, en la que se incluye: el código de respuesta, que dependerá del resultado de la operación, si ha sido exitoso o no, la key, la cadena, la N y el vector de doubles.
Se manda el struct por la cola y esta se cierra y se desliga del proceso.
El código en la respuesta es igual en las 4 primeras funciones: init, get, set, modify y delete devuelven 0 en caso de éxito y -1 en caso de error. Exist devuelve 0 en caso de no existir la tupla, 1 en caso de sí existir, -1 en caso de error.

Claves

Para implementar **int init** lo primero que hace es inicializar la cola de mensajes, definir la cola de respuesta y definir unos atributos que nos sirven para el tratar con las colas. Abrimos las colas y hacemos un control de errores. producimos el datos y lo copiamos a la estructura Mensaje, que contendrá los distintos atributos que necesita (en este caso solo en código de operación y el nombre de la cola). Se envía el mensaje producido y después se lee la respuesta del servidor, una vez leída la respuesta se cierra la cola.

Para el resto de claves a implementar se sigue un desarrollo muy parecido. En **set_value** el dato que producimos para el mensaje contiene todos los datos de la estructura del mensaje. En el caso del **get_value** en la respuesta del servidor nos llegan los datos de la key consultada, que después devolvemos. En el caso de **modify_value** es prácticamente inguinal que en **set_value**. Para **exist_key** enviamos al servidor un mensaje con la key que queremos comprobar si existe. Para tratar las colas en estas funciones siempre utilizamos la misma implementación.

Cliente

Para la parte de cliente hemos generado unos datos para insertar y probar las distintas funciones de las claves.

Compilación

El fichero Makefile genera el ejecutable de clientes y de servidor utilizando el comando make en la línea de mandatos. Además creará la librería dinámica libclaves.so dentro del directorio lib del proyecto (Si esto directorio no existe al momento de intentar compilar, sucederá un error)
Para poder utilizar la librería enlazada, hay que ejecutar dos comandos más en la terminal que se esté utilizando, de no hacerlo así, el ejecutable clientes dará error al intentar ejecutarse.
Estos comandos son:
LD_LIBRARY_PATH=$(pwd)/lib
export LD_LIBRARY_PATH

Esto modifica el valor de la variable de entorno LD_LIBRARY_PATH para que primero intente buscar librerías dinámicas en el directorio de trabajo actual/lib, que es donde se encuentra libclaves.so