Universidad Carlos III

2023-24

Ejercicio 2 - 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 el socket en el que los clientes mandan las peticiones, para los mensajes de los clientes, se envía un mensaje por el socket para cada argumento, tanto servidor como claves saben cuántos argumentos se envían y reciben dependiendo del tipo de operación, por tanto, no se esperará más tiempo del necesario a argumentos innecesarios.
Después de iniciar el socket servidor, 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 socket por el que se inició la comunicación con el cliente, el hilo recoge el mutex para copiar el valor del descriptor del socket 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 envía un último mensaje desde el servidor a través del socket en el que se especifica el código de resultado 0 o -1 (o 1 en el caso de exist),la operación get se comporta de una forma un poco diferente, primero mandará el código de operación al cliente, en caso de que este sea positivo, enviará el resto de argumentos (y el cliente los estará esperando) en caso de que la operación no haya sido positiva, no se mandarán el resto de argumentos, el cliente sabrá esto y continuará con su ejecución sin esperarlos.
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 **init** lo primero es inicializar el socket, enviamos el código de operación, comprobamos que no haya error en el envío y esperamos la respuesta, comprobamos si hay un error al recibir la repuestos y por último cerramos el socket.

Para la función **set_value** la primera parte es igual que para **init**. después de pasarle el código de operación le vamos enviando todos los datos uno a uno, haciendo un sendMessage por cada dato. Cuando se han enviado todos esperamos la respuesta y cerramos el socket.

En la función **get_value** envía solo el código de operación y la key, después espera la respuesta y con recvMessage se encarga de recibir todos los argumentos. Se comprueba que lleguen todos y en caso de que se detecte que no ha llegado todo salta un error.

La función **modify_value** es muy parecida a **set_value,** ya que envía todos los argumentos que se le pasan para que se pueden modificar.

En **delete_key** se envía el código de operación y la key a eliminar, después solo espera a que reciba la respuesta del servidor.

Para **exist** es igual que para delete, ya que solo envía la operación y la key y espera a la respuesta.

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 este directorio no existe al momento de intentar compilar, sucederá un error)
Para poder utilizar la librería enlazada, hay que declarar varias variables de entorno en la terminal que se esté utilizando, de no hacerlo así, los ejecutables darán error al intentar ejecutarse.
Estos comandos son:
IP_TUPLAS=localhost
export IP_TUPLAS
PORT_TUPLAS=4200
export PORT_TUPLAS
LD_LIBRARY_PATH=$(pwd)/lib
export LD_LIBRARY_PATH

Para ejecutar cada programa, se hará de la siguiente manera:
El valor que se haya especificado para PORT_TUPLAS debe ser el mismo que se utilice a continuación para ejecutar el servidor de la siguiente manera:
./servidor <puerto_elegido>
Y para ejecutar el cliente (debe ser en la terminal en la que han definido las variables de entorno), se hará:
./cliente