# Documentation for key-value store server implementation

## Introduction:

The key-value store server implementation is a simple implementation of a server that can store key-value pairs in memory and allow the client to interact with the stored data. The server is implemented in C language using sockets and pthreads.

## File Structure:

The server implementation is divided into two main files:

- main.c: This file contains the main function and is responsible for setting up the server socket, listening for incoming connections, and creating new threads to handle each connection.

- keyValStore.c: This file contains the implementation of the key-value store. It provides functions for adding, retrieving, and deleting key-value pairs.

## Compilation and Execution:

To compile the server, simply run the command "make" in the root directory of the project. This will create the binary file "kv\_server".

To run the server, execute the binary file by running "./kv\_server".

## Implementation:

The server listens for incoming connections and creates a new thread to handle each connection. The maximum number of threads that can be created is set by the NUM\_THREADS constant in the main.h header file.

When a new connection is accepted, the server creates a new thread and passes the socket file descriptor as an argument to the thread. The thread then handles the communication with the client by receiving client requests, parsing the requests, and sending back the appropriate responses.

The server stores key-value pairs in a hash table implemented in the keyValStore.c file. The hash table is implemented as an array of linked lists, with each linked list containing all the key-value pairs that have the same hash value. Each linked list node contains a key and a value.

## Multiclient:

The program is made multiclient able by creating a new thread for each client connection.

When a new client connection is accepted, a new thread is created using the pthread\_create() function, passing the function handle\_client as the thread routine. The handle\_client function receives the client socket file descriptor as its argument, and it is responsible for reading and processing the client's requests.

The pthread\_create() function creates a new thread, and the new thread executes the handle\_client() function, allowing multiple clients to be served concurrently. Each thread has its own stack and execution context, allowing them to operate independently of each other.

The main thread of the program remains in the while(1) loop, waiting for new client connections. The program terminates when a client sends the QUIT command or when an error occurs. When the QUIT command is received, the main thread cancels all the other threads and exits the program.

The use of threads allows the server to handle multiple clients simultaneously, increasing the system's efficiency and improving the user experience.

## The key-value store implementation provides the following functions:

- void initKeyValStore(): Initializes the hash table.

- int put(char\* key, char\* value): Adds a new key-value pair to the hash table or updates the value of an existing key.

- int get(char\* key, char\* value): Retrieves the value of a key from the hash table.

- int del(char\* key): Deletes a key-value pair from the hash table.

## The client can interact with the server by sending requests over a socket connection. The server expects requests to be in the following format:

**PUT command (case 0):**

The put() function is called with the key and value obtained from the command.

If put() returns 0, indicating a successful operation, a response string is formatted using snprintf() to indicate the successful PUT operation: "PUT:key:value".

If put() returns a non-zero value, indicating an error, a response string is formatted to indicate the error: "PUT ERROR".

**GET command (case 1):**

The get() function is called with the key obtained from the command.

If get() returns 0, indicating a successful operation, a response string is formatted using snprintf() to include the retrieved value: "GET:key:value".

If get() returns a non-zero value, indicating an error or key not found, a response string is formatted to indicate the error: "GET ERROR".

**DEL command (case 2):**

The del() function is called with the key obtained from the command.

If del() returns 0, indicating a successful operation, a response string is formatted using snprintf() to indicate the successful deletion: "DEL:key".

If del() returns a non-zero value, indicating an error or key not found, a response string is formatted to indicate the error: "DEL ERROR".

**QUIT command (case 3):**

The server receives a QUIT command, indicating the termination of the server.

A message is printed to indicate the received command: "Received QUIT command, terminating server".

All threads are canceled by calling pthread\_cancel() for both the main\_thread and other threads in the threads array.

The server socket is closed using close(sockfd).

The process is exited with a status of 0.

Default case:

If none of the recognized commands (PUT, GET, DEL, QUIT) are received, the default case is executed.

A response string is formatted to indicate an invalid command: "INVALID COMMAND".

## The server responds to requests in the following format:

- PUT:key:value

- PUT ERROR

- GET:key:value

- GET ERROR

- DEL:key

- DEL ERROR

- INVALID COMMAND

## Conclusion:

The key-value store server implementation provides a simple way to store and retrieve key-value pairs using socket communication. The implementation can be easily extended and modified to fit the specific requirements of different applications.