To create the key value store, the task was split into 3 components: Sockets, Parser and Queue. Each component was then combined to create the full key value store. The steps used to create each component are analysed below.

Sockets – This involved initialising the sockets that would be used by the control and data ports. A function was used to incorporate the necessary implementation of the sockets called initSocket [107].

initSocket [107] – This function takes 4 arguments, the port of the socket, the socket address, the socket length and the backlog. The socket function is used to create the socket for the two-way channel with the appropriate domain, socket type and protocol set [111]. This produces a file descriptor that is used by the bind function to switch the socket from client to server mode on the appropriate ports [119], the file descriptor is also used by the listen function to listen to incoming pending clients [124]. Using the given control and data ports, the sockets are initialised [398,399] and then stored in a poll structure [413,414] and then polled [420].

Parser – This involved using the parser function implemented in the parser files to parse the appropriate commands for the key value store. Functions handle_data [138] and control_data [334] are used to handle the parsed commands for the data and control ports respectively.

handle_data [138] – This function works in hand with the worker threads to handle request from the data port, it takes the accepted connection and a buffer that stores the request and response to be sent to and fro from the telnet client and the server as arguments. The request from the client is read into the buffer, using the parse_d function [147], the buffer is parsed into command, key and value. Using the command supplied from the client, the request is handled appropriately

*GET [154] – using the key and the findValue function [156], if the value exists it is returned, otherwise an error message is returned

*END [166] – The connection is ended with the client [170]

*COUNT [174] – using the countItems function [177], the number of items in the store is returned to the client

DELETE [182] – using the key and the deleteItem function [185], the key is deleted and response sent to the client

EXISTS [193] – check if the key exists using the key and the itemExists function [194]

*PUT [203] – create an allocated memory to store the text [205] then using the key, text and the createItem function [213] store the key and its value.

control_data [334] – This function handles incoming request from the control port, the request from the accepted connection is stored onto a buffer, using the parse_c function [347], the buffer is parsed into a command, handled appropriately and the response is sent to the client.

*COUNT [349] – the number of items is counted using the countItems function [351] and the connection is closed

*SHUTDOWN [367] – A shutdown response is sent to the client; the connection is closed and the server is shut down by posting the s_shutdown semaphore to the worker threads

Queue – This involved creating a queue [queue.c] that is used to store incoming connections from the data port for the worker threads to handle. This involved using semaphores and a producer consumer strategy

*s_queue_lock: mutex lock for the queue

*s_space_availabe: controls the space available for the queue

*s_work_available: used to wake sleeping worker threads that work is available

*s_data_lock: controls the number of data ports that can send requests

In the main function once the socket for the data port has been polled [440], it then checks if space is available on the queue [444] and pushes the connection to the queue [464] and signals the worker threads that work is available [466]. The worker threads are handled using the worker function [266], on initialisation the threads wait till work is available i.e. a client connection has been pushed to the queue [275], once work is available the thread pops the connection off the queue [296] then informs the main thread that a space has been freed off the queue [298] and passes the connection to the handle_data function to handle the client request [318].

The implementation is correct, efficient and thread safe by using semaphores as mutex locks, they were locked and released appropriately to allow for smooth synchronisation between threads. The implementation is split into several functions for good code structure and seamless integration of the key value store, error handling was taken care for any function that returns a value.

The program was tested on snowy for different scenarios, on simple cases where only one port is being used (either control or data) it handles the request appropriately. Where a data port is still sending request to the server and a control port tells the server to shutdown, no other data parts can be opened and once the opened data port has finished the server shuts down fully. If multiple data ports are open they work synchronously so long as there is a free worker thread available to handle the request, if no worker is available the data port is blocked and waits till another data port is closed to send requests.