Coursework 2 – a key/value store server

In this coursework we will put together all the topics that we learnt in the unit and write a server of our own.

Introduction

Key/value stores

A key/value store is a simple kind of database where the main operations are

- put KEY VALUE – stores the value under the given key
- get KEY – returns the value stored under this key, if any

A simple key/value store is implemented in the files kv.c and kv.h in the ZIP file for this coursework. The API that you should use is in the header file; all other methods in the source file are implementation details. Note that this is a "toy" implementation – it is not particularly efficient, it does not save data to disk and there is a limited number of slots available. More importantly, the implementation is not thread-safe so you will need to manage synchronization yourself.

Telnet and TCP/IP

TCP/IP are two of the protocols used for internet traffic. IP allows each computer to gain a unique address and TCP allows, among other things, each computer to use many different port numbers to handle different protocols. The main job of TCP is turning a packet-based network into a stream-based one. The combination of an IP address and a TCP port number is called a socket.

In your server, you will bind to two sockets – one for control messages and one for data. To connect to and test your server, you can use the telnet command-line tool. The syntax to start it is "telnet HOST PORT". We will be running our server on the same host as our telnet client so we use localhost as the host. For example, if a server is listening on port 8000 then you can do this:

```
$ telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
```

The IP address 127.0.0.1 is special and refers to "this computer". Now that you have an open connection, any time you press ENTER the line you typed gets sent to the server and the response from the server is shown underneath in your terminal.

It is up to the protocol that you run on top of TCP to determine when to close the connection – telnet doesn't do this for you. If the server closes the connection, telnet prints

```
Connection closed by foreign host.
```

and exits, causing you to return to the terminal in which you started it.

Protocols and parsing

To run a service over TCP, we need to define our own protocol – what kinds of commands will our server accept, how will it reply to them, how will it deal with errors and how does a client tell the server to end a session. The protocol that we will implement is line-based and supports the commands defined later on. A line can be maximally 255 characters long, including the newline and 0-terminator characters.

In our protocol, the server begins by sending the client a "welcome" line. Then, the client sends the server a single line. The server replies with a single line – either the answer to the client's query or an error message. This procedure – client sends a line, server replies – continues until the client sends an empty line, then the server prints a "goodbye" message and closes the connection.

Since this assignment is about handling the concurrency and networking part of a server, a parsing function is provided in parser.c and parser.h. To use it, read any pending data from the client into a buffer e.g. char buffer[LINE] and set up the following:

```
char buffer[LINE]; /* put your data in here */
enum DATA_CMD cmd;
char* key;
char* text;
parse_d(buffer, &cmd, &key, &text);
```

This function modifies the buffer. cmd, key and text are output parameters – you pass them by reference and the parser sets them correctly. After this you can match cmd against the following enum values (defined in the header file):

D_PUT	The line contained a "put key value" command. The pointers key and text point to 0-terminated strings containing the key and value.
D_GET	The line contained a "get key" command. The pointer key points to the key and text is NULL.
D_COUNT	The line contained a "count" command. key and value are NULL.
D_DELETE	The line contained a "delete key" command. The pointer key points to the key and text is NULL.
D_EXISTS	The line contained an "exists key" command. The pointer key points to the key and text is NULL.
D_END	The line was empty. This is a signal that the client wishes to close the connection.
D_ERR_OL	Error: the line is too long.
D_ERR_INVALID	Error: invalid command.
D_ERR_SHORT	Error: too few parameters (e.g. "put 1", which is missing a value).
D_ERR_LONG	Error: too many parameters (e.g. "count x", count does not take a parameter).

The pointers key and text point to positions within the buffer. This means that if you allocated the buffer on the heap, you can free the buffer when you are finished processing the command but you do not have to free key and text separately.

So your procedure for handling client requests should be something like:

1. read() up to 255 bytes from the client into the buffer (on the client side, telnet will send data only when you press ENTER, so it arrives a line at a time).
2. Parse the buffer and handle the command.
3. write() the response line to the client (you can reuse the same buffer).
4. If the command was D_END, close the connection. Otherwise go back to 1.

Of course you may have to do some locking/synchronisation while you're handling the command. Make sure you do not hold a lock while doing client I/O – this can block.

Example

Here is an example session with two terminals and a completed key/value server. The text that you type is indicated in bold. Text printed in the server terminal 1 is for information only – you can print whatever you want here. Text printed in the client terminal 2 is the server's response to the last command.

Terminal 1:

```
$ ./server 8000 5000
Server started.
(blocks)

Got a connection.
Delegating to worker 1.
Worker 1 executing task.
(blocked)









(unblocked)
Shutting down.
```

Terminal 2:

```
$ telnet localhost 8000
(some telnet output)


Welcome to the KV store.
COUNT
0
PUT name David
Success.
COUNT
1
GET name
David
EXISTS name
1
EXISTS office
0
(empty line)
(telnet says connection closed)

$ telnet localhost 5000
(some telnet output)
SHUTDOWN
Shutting down.
(telnet says connection closed)
```

When you have completed your server, you should be able to connect to it from multiple clients simultaneously and values that you save in one client should be visible to others.

Assignment

1. Write a server program matching the specifications in this document and submit the source code of your program (including a Makefile) as a ZIP file to SAFE.
2. Write a short report (1 sheet A4, PDF format, min. 2cm margins and 10pt fonts – these limits are strict, anything else will not be marked) explaining how thread synchronisation works in your program.

To build your program, I will unzip your submitted file in a new folder and type "make" on a machine with the same set-up as the lab ones. This must build an executable file named "server". It is your responsibility to ensure that this procedure works, for example that your makefile is correct and all required files are present. If your program does not compile, you may receive 0 marks for the entire assignment. Do not submit an executable program that you have compiled yourselves – this will incur a penalty.

Your program must run when called as "server CPORT DPORT" where CPORT and DPORT are port numbers for the control and data ports. Your program must bind to both these ports (you can assume that they are not in use) and listen for the following commands. Each command is ended by a newline.

I recommend that you use the template server.c file provided to start coding.

Worker thread pool

When your program starts, it must create a thread pool of NTHREADS threads (this is currently set to 4 – you can experiment with this but set it back to 4 when you submit). This thread pool must be created before you accept any client connections. When you get a client connection on the data port, your program must delegate this connection to one of the free worker threads. If no workers are currently free, you have two options: wait, or store the connection request somewhere and have workers check for pending requests whenever they complete a task.

One way to do this is to start all worker threads and have them immediately wait for instructions. Then in a loop, accept() new connections in the main thread and whenever a connection succeeds you pass the new file descriptor for the connection to one of the waiting worker threads and wake it up. If no worker threads are waiting, the main thread waits until one becomes free. During this time the server cannot accept any more connections of course.

Another way is to have the main thread accept() connections and put them into a queue – the worker threads then wait on this queue and handle the connections. This is a producer-consumer situation where the main thread "produces" connections and the worker threads consume them. A fixed-size queue is fine – if it's full, then the main thread can wait.

Control protocol

On the control port, the available commands are

COUNT: in response, your program should display a line containing an integer that represents the number of items currently in the store.

SHUTDOWN: your program should stop accepting any new connections on the data port and terminate as soon as all current connections have ended.

After reading and processing a single command on the control port, your program should terminate the connection on this port. While a connection on the control port is open, your program does not have to listen for new connections on the data port (i.e. you can handle the control port in your main thread). However, while there is no connection open on the control port, your program must be able to handle multiple simultaneous connections on the data port. You will probably want to poll() the control and data ports in your main thread and then handle whichever port you get a connection on. Use the provided function

```
enum CONTROL_CMD parse_c(char* buffer);
```

to parse the line you get on the control port. It returns one of C_SHUTDOWN, C_COUNT or C_ERROR. Note that this function may modify the buffer contents.

Data protocol

On the data port, your program should accept new connections whenever the control port is not busy and delegate them to one of the free worker threads – or a queue that is watched by the worker threads.

In the worker threads, you should process the following commands. Each command is a single line and you may assume that no line is longer than the constant LINE=255. Use the parser provided as this is not an assignment about writing parsers. Remember that the provided key/value store is not thread-safe so you have to synchronise all access to it.

put KEY VALUE – store the given value under the given key. This should succeed whether or not the key already exists (if it does, overwrite the old value). Your program should return a single line explaining whether the operation was successful or not, e.g. "Stored key successfully." or "Error storing key."

get KEY – if this key exists, your program should return a single line to the client containing the stored value. If not, print a single line error message e.g. "No such key."

count – return a single line containing the number of items in the store.

delete KEY – if the key exists, delete it and print a single-line success message. Otherwise print a single-line error message.

exists KEY – print a single line containing 1 if this key exists in the store, otherwise 0.

(empty line) – print a "goodbye" message and close the connection.

Error handling

An invalid command, overlong line or broken connection from a client must not crash your server. Proper error handling on the socket and connection is therefore essential and you cannot abort() or exit the program if something goes wrong that is the client's fault.

You must also remember to free resources properly when a client connection closes.

Plagiarism

All work that you submit must be your own, except that you may use the template files provided for this coursework without referencing them.

You may not share code with or show your code to other students. You are welcome to discuss ideas with other students but not your code.

You should not need any third-party libraries to complete this project. You may use pthreads and any other functions or libraries that are part of the POSIX standard as long as your program compiles and runs on the lab machines. Linking your code against existing libraries (e.g. pthreads) is not plagiarism.