CS214 Spring 2021
Project III

David Menendez

Due: Monday, May 3, 2021, at 11:00 PM (ET)

This is a group project. You may work alone, or with a partner. Your partner may be in a different section.

If you are working with a partner, both partners must declare the partnership on or before Monday, April 26, using the form at `https://forms.gle/n8hQKZy9yXEkyWSUA` or by e-mailing your TA.

1 Summary

You will write a server for a simple key-value storage service. Client programs may use this server to set and retrieve the *values* associated with *keys*, where the keys and values are strings. For example, one client may set the key "day" to the value "Sunday". If a second client then requested the value of "day", the server would reply "Sunday".

Keys and values may be strings of any length, but may not include the terminator (`\0`) or newline (`\n`).

2 Server program

The server program has one argument, an integer identifying which port it will use to listen on. For testing purposes, choose a port number between 5000 and 65 536 that is not currently in use. (The program `netstat` shows all ports currently in use by some program.)

The server program opens and binds a listening socket on the specified port and waits for incoming connection requests. For each connection, it creates a thread to handle communication with that client. The thread terminates once the connection is closed.

The server also maintains the key-value data in a data structure shared by all threads. You may choose any convenient data structure, provided that it supports adding and removing keys and changing the value associated with the key. A linked list maintained in alphabetical order is sufficient, but you may use a binary tree or hash table or some other structure. Be sure to synchronize access to the structure to avoid non-deterministic behavior.

3 Communication protocol

Clients initiate and terminate connections to the server. Once a connection is open, a client will send zero or more requests to the server. For each request, the server will take some action and

respond. After receiving a response, the client may send another request or close the connection.

3.1 Messages

Our communication protocol has three requests that the client may send to the server.

GET *key* Requests the current value associated with *key*. If *key* is set, the server returns its value. Otherwise it returns key-not-found error.

SET *key*, *value* Associates *value* with *key*.

DEL *key* Deletes *key* from storage. If *key* is set, it returns the value that was associated with it and unsets the key. Otherwise, it returns key-not-found error.

3.2 Message format

Requests sent by the client are broken into three or four fields, each terminated by a newline character (`\n`). Each message begins with a code identifying the message type, followed by a decimal integer specifying the length (in bytes) of the payload, and finally the payload itself.

The message codes are `GET`, `SET`, and `DEL`. The payload for `GET` and `DEL` is one field, indicating the key to retrieve or delete. The payload for `DEL` is two fields, indicating the key and value.

The message length is the length of the fields making up the payload (in bytes), *including* the newlines that terminate each field. Thus, the string literal `"GET\n4\nday\n"` represents a request for the key "day".

Successful responses For each client request, the server has a corresponding response message indicating success. Each response begins with a response code followed by a newline. The response codes are `OKG` (for get), `OKS` (for set), and `OKD` (for delete).

The `OKG` and `OKD` codes will be followed by a payload length field and the payload itself, which will be the value associated with the key requested or deleted. The string literal `"OKG\n7\nSunday\n"` represents a successful response to a `GET` with the value "Sunday".

Unsuccessful response If a `GET` or `DEL` request asks for a key that is not set, the server will respond with `KNF` followed by a newline.

Error responses Error messages from the server begin with the code `ERR` followed by a newline, a three-character error code, and the final newline.

The error codes are `BAD`, indicating a malformed message, `LEN`, indicating that the message length was incorrect, and `SRV`, indicating an internal error in the server.

We consider messages containing an unknown message code, or one that is inappropriate in context, or fields containing inappropriate data (such as non-digit characters in the length field) to be malformed. Messages that contain the wrong number of newlines will generally have the wrong length.

The server will close the connection after sending an error response.

General features Messages always begin with a three-character code followed by a newline. Depending on the code, the message may be followed by one or more additional fields. If any of the fields do not have a fixed length, the first field is a decimal integer followed by a newline. The integer gives the length in bytes of the remainder of the message.

The last byte of the message will always be a newline. The number of newlines that will occur in the message is determined by the message code. This allows programs to detect messages with incorrect lengths. Once the length field is parsed, the final newline must come exactly that many characters later. If the newline comes too early, the message is too short. If the newline does not come within the specified number of bytes, the message is too long. In both cases, the server should send ERR with LEN and close the connection as soon as the error is discovered.

3.3 Examples

The client opens a connection to the server and sends this message, setting the value of "day" to "Sunday".

```
SET
11
day
Sunday
```

Note that each field in the message ends with a newline character.

The server sets "day" and reports success.

```
OKS
```

Next, a client requests the current value associated with "day":

```
GET
4
day
```

The server responds with the current value.

```
OKG
7
Sunday
```

A client requests to delete the key "day".

```
DEL
3
day
```

The server unsets the key and responds with its most recent value.

```
OKD
7
Sunday
```

A client requests the value associated with "day".

```
GET
4
day
```

Because "day" has been unset, the server responds with key-not-found.

```
KNF
```

4 Advice

The general advice for Project II applies here. Always check whether a function that might fail has failed. If you can't be bothered, then write a macro and use that instead.

Reading and writing a socket is very similar to working with a file, but be aware of the streaming model that TCP uses: when you call `read()` and get some bytes back, this is just the bytes that are currently available. Calls to read *do not* correspond to "messages". This is why our application protocol specifically notes where messages end. It is entirely possible for a single batch of bytes to contain part of a message, or more than once message.

Prioritize getting your server to work with correct inputs first, but design your code to make it easier to add checks for malformed and incorrect messages from the client. Ideally, your server will report an error message and close the connection as soon as it discovers a problem. For example, if the first byte in a message is `X`, it should report an error immediately, without calling `read()` again.

Reading input from the socket 1 byte at a time is inefficient, but may be a good first approach.

A realistic protocol would require that a complete message arrive within some time limit. As it is, a client that connects to your server, sends the start of a message, and then keeps the connection alive without finishing will tie up a socket and thread in the server. For the purposes of this assignment, we will not worry about such a scenario.

4.1 Testing

You are not required to hand-in client code, but you may find it helpful to write one or more simple client programs to test your server.

You may also use `telnet` to open a connection to your server and enter messages directly.

5 Submission

Submit a Tar archive containing your source code, makefile, and a README. The README must contain:

- Your name and NetID[1]
- Your partner's name and NetID, if you worked with a partner.
- A brief description of your testing strategy. How did you determine that your program is correct? What sorts of files and scenarios did you check?

[1] If you prefer not to share your NetID with your partner and/or Google, contact your TA and request a unique identifier to use for this project.

Do not include executables, object files, or testing data. Before submitting, be sure to confirm that your archive contains the correct files!

For safety, both partners should create and submit an archive.

6 Grading

Your program will be scored out of 100 points: 50 points for each part. Your grade will be based on test cases and on manual examination. Testing will be performed on iLab machines, so be certain that your code will compile and execute on the iLab!

Your code should be free of memory errors and undefined behavior. We may compile your code using AddressSanitizer, UBSan, Valgrind, or other analysis tools. You will lose points if these tools report memory errors, space leaks, or undefined behavior. You are advised to take advantage of AddressSanitizer and UBSan when compiling your program for testing.

Late submissions Because this assignment is due on the last day of class, we will not be able to accept late submissions.

Plan to complete your assignment early! Build slack into your schedule so that unexpected delays or difficulties do not result in a late submission.