Project README file

This is YOUR Readme file.

Note - You will submit a PDF version of this readme file. This file will be submitted via Canvas as a PDF. You may call it whatever you want, and you may use any tool you desire, so long as it is a compliant PDF - and for us, compliant means "we can open it using Acrobat Reader".

Project Description

Your README file is your opportunity to demonstrate to us that you understand the project. Ideally, this should be a guide that someone not familiar with the project could pick up and read and understand what you did, and why you did it.

Specifically, we will evaluate your submission based upon:

- Your project design. Pictures are particularly helpful here.
- Your explanation of the trade-offs that you considered, the choices you made, and *why* you made those choices.
- A description of the flow of control within your submission. Pictures are helpful here.
- How you implemented your code. This should be a high level description, not a rehash of your code.
- How you *tested* your code. Remember, Bonnie isn't a test suite. Tell us about the tests you developed. Explain any tests you *used* but did not develop.
- References: this should be every bit of code that you used but did not write. If you copy code from one part of the project to another, we expect you to document it. If you *read* anything that helped you in your work, tell us what it was. If you referred to someone else's code, you should tell us here. Thus, this would include articles on Stack Overflow, any repositories you referenced on github.com, any books you used, any manual pages you consulted.

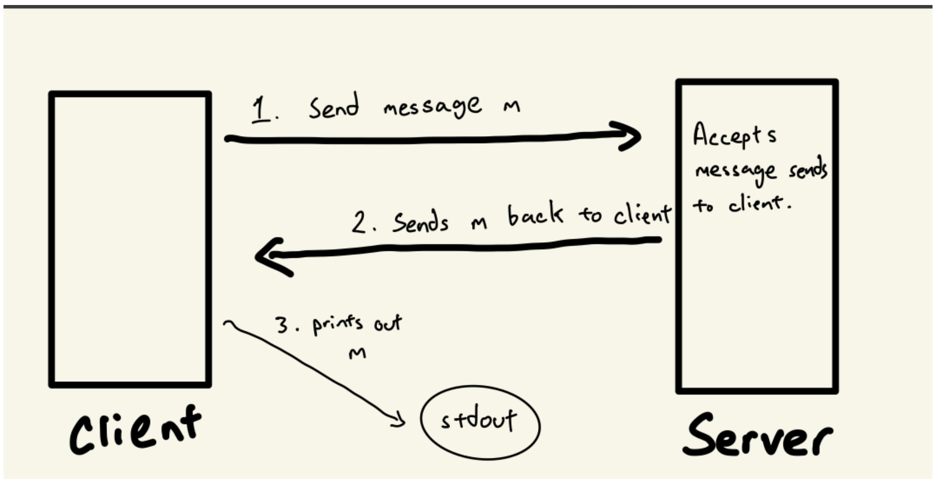
In addition, you have an opportunity to earn extra credit. To do so, we want to see something that adds value to the project. Observing that there is a problem or issue *is not enough*. We want something that is easily actioned. Examples of this include:

- Suggestions for additional tests, along with an explanation of *how* you envision it being tested
- Suggested revisions to the instructions, code, comments, etc. It's not enough to say "I found this confusing" - we want you to tell us what you would have said *instead*.

While we do award extra credit, we do so sparingly.

Echo Client Server

The echo client server is one the most common starter programs involving sockets. The server sends back or echoes back exactly what it receives from the client. The server is IP agnostic, meaning that it works with either IPV4 or IPV6. In order for that to be possible, instead of setting up the socket using specifically AF_INET or AF_INET6. To do that, getaddrinfo was used to get the address information, and that information is used later to create the socket connection. The server creates the socket and waits for it to connect. Once it connects, it is continuously listening and waiting for the client to send something. When the client sends the server a string with a max of 16 bytes, the server accepts it and then sends the same string back to the client. For the client, since we didn't need to worry about it being IP agnostic, I set up the socket using AF_INET which makes it IPV4. The function gethostbyname was used to get the address information based on host name. Once the client connects, it sends the string that was inputted as a command line argument to the server it connected to. Then it receives the same string back from the server and prints it to the standard output. I didn't write much tests for this warmup, I mostly just tested it manually and using binaries other students provided to double check my work.

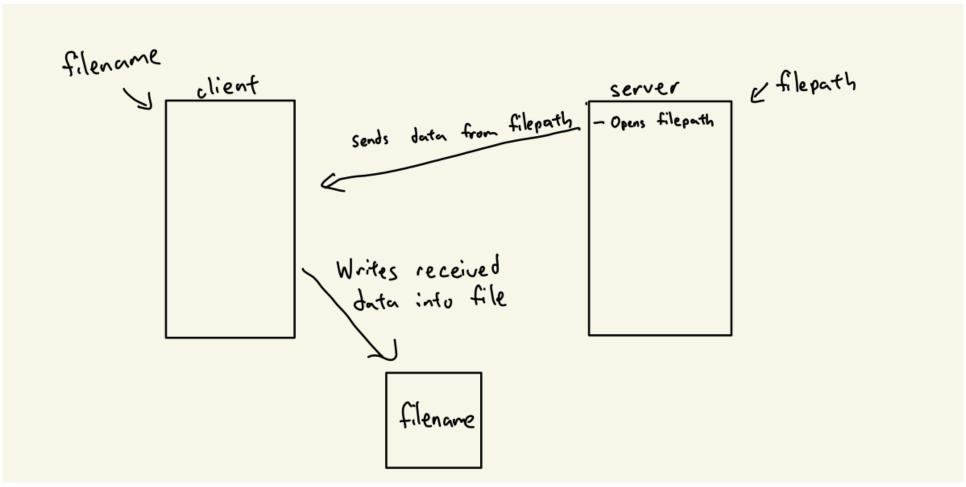


References

For this warmup, I consulted Beej's Guide to Network Programming to better understand how to make my server IP agnostic. To better understand how to use getaddrinfo, I consulted this article on StackOverflow, https://stackoverflow.com/questions/60599932/c-getaddrinfo. Other than those references, I used this github https://github.com/zx1986/xSinppet/tree/master/unix-socket-practice, as a reference to write the basic client and server code.

Transferring a File

In this warmup, the same client and server socket structure was used from the Echo Server and Client warmup. In this warmup, the server accepts a file path as an argument. Before the server accepts a connection to the client, it opens the file using the open function as read only since nothing had to be written to it. The server then connects to the client. Once connected, the server uses the read function, reads into a buffer of a predefined size and sends that over to the client. Now since the send function doesn't always send the all the bytes, I put the send function in another while loop that makes sure the full message was sent to the client. The same logic was apply to the read function for the file, it reads a predefined size but not the entirety of the file so it needs to keep reading it till it reaches the end of the file. Once the entire file is sent, the server closes the file descriptor and the connection to the client. For the client, a similar socket structure was used as the echo client. The client connects to the user, and then before it receives a message from the server, it opens another file using the filename provided, but this time opens it for writing, if the file doesn't exist it creates it. Then in a while loop, the client keeps receiving from the server till the bytes received is 0. Every time it receives a buffer, it writes the same buffer to the file. Once it finishes reading and writing, it closes the file descriptor it opened. Again, I did not create any tests for it, most of the testing was manual and using other binaries that other students provided. For the manual testing, I tested giving the server a file that didn't have a lot of content and doubling checking the file created by the client is the same as the one provided to the server. I also tested giving it a bigger file and then doubling check the checksum of the two files and making sure they're equal.



References

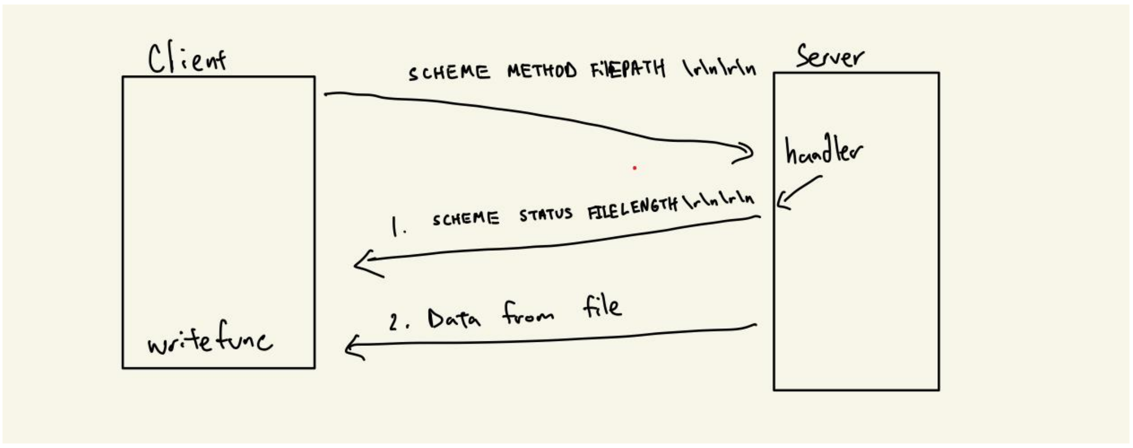
For this, I copied a lot of the same code used in the Echo Client and Server to this warmup. The difference being was nothing was sent to the server this time and the server handle everything outgoing. To better understand file I/O operations, I used the man pages to understand the open function https://man7.org/linux/man-pages/man2/open.2.html and the read and write functions for it.

Part 1 Getfile Protocol

Instead of implementing the sockets directly, I had to implement the Getfile API library. All the descriptions for what each function did in the server and client libraries were in the header files for their respective C files. For the client library, the gfcrequest_t struct needed to be defined with values. Most of these values in the struct would later be used in the gfc_perform function. The get functions return values from the struct such as bytesreceived, filelen, status and so on. The set functions in the client library set those values in the struct such as port number, write function, header function and so on. The major part of the client library was the gfc_perform function. The function sets up the socket connection and sends to the server the header. In this client, getaddrinfo was used to make it IP agnostic and create and connect to the socket. Once its connected, its creates a request header with the scheme and method and file path that is then sent to the server. The client then waits for the server to respond with its own header. The client then parses the response header and makes sure the values are correct along with the file length and status if the status is OK. I used strtok to split the string based on the space between the scheme, status, and file length. Once those are received, it starts receiving the data on the file. There could be the case of where data is sent along with the header, so I had to double check the response header to see if any data was sent after the \r\n\r\n. Again, strtok was used to find that. Going through piazza, I found out that strlen doesn't read NULL characters, so instead of using strlen to find the number of bytes sent along with the header, I took the initial bytes received with the header and just subtracted the number of bytes of the scheme, status, file length, and header end marker. Then it starts to accept the rest of the data sent by the server. I had to use this, because during submission, one test kept failing and it turned out the issue was I was not accounting for the bytes that were sent after the end marker. For each buffer it receives, it uses the write func that the gfclient_download uses. It keeps receiving till it receives the entirety of the file length in the header. Once it receives the length, it closes the socket and exits.

For the server library, similar to the get and set functions of the client library, two structs were used. One was the server itself and the second is the context struct, which contains the socket id, the file length, and the status. The three main functions of the library are gfs_send, gfs_sendheader, and gfserver_serve. The gfserver_server function takes in the gfserver_t struct and spins up a socket connection to the client. Once it connects, it receives the request header from the client. After receiving the request header, the server parses it using strtok to get the information needed. It double checks the scheme and method and makes sure that they're GETFILE and GET respectively. Once those are checked, it checks the filepath and makes sure it begins with a /. If all those are correct, it calls the handler with the handler arguments and the path token. The handler in turn calls the sendheader and send functions. In the sendheader function, it gets the status and file length from the context struct. It enters a switch statement depending on the header status, and then sends the header to the client. Returns 0 if the send was successful. In the send function, it takes the data and the file length, and keeps sending the data till the entire file length is sent and the file length is returned as the value. Once the entire file is sent, it closes the connection and frees up the struct.

No tests were written. Did more manual testing, editing the context files and other files as needed. Also tested with other binaries that student provided. I also used the feedback from gradescope to help debug and find out what was not working correctly in my code.


Client
SCHEME METHOD FILEPATH \r\n\r\n
Server
handler
1. SCHEME STATUS FILELENGTH \r\n\r\n
2. Data from file
writefunc

References

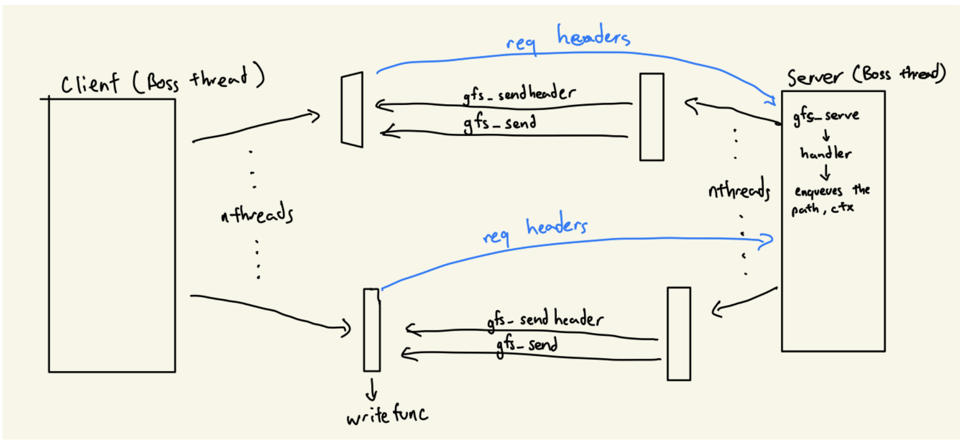
Most of the socket connection for the server and client side, were copies of what I did in the earlier warmups. In addition to those, I used the man pages to look up string functions in c. Such as strncmp, strstr, and strtok. Since it has been a while I coded in C, I used tutorialpoints to better understand structs tutorialspoint.com/cprogramming/c_structures.htm. I also used Stack Overflow to better understand double pointers https://stackoverflow.com/questions/5580761/why-use-double-indirection-or-why-use-pointers-to-pointers.

Part 2 Multithreaded Client server

In this part instead of implementing gfclient or gfserver, I had to implement gfclient_download and gfserver_main and the handler function. For gfclient_download, I created a struct for arguments needed by the worker threads, such as number of requests, the server name, port number. The client also initiates the steque where the requests will be enqueue by the boss thread. The boss thread first creates worker threads. The client locks the boss thread using one of the mutex and then using steque functions to enqueue the request path and then signals the worker threads that the queue has a path ready to be consumed. Using another mutex, the client locks the boss thread, this time waiting for the number of requests to be left is 0. Once it reaches 0, it unlocks the mutex and then broadcasts the signal so all the helper functions return 0 so the client can exit. The helper function uses alot of the code that was provided already to handle the downloading. It locks the mutex and waits for the queue to not be empty and then pops the first path on the queue. Once it finishes downloading, it locks the mutex again, decrements the number of requests, signal to the boss thread and then unlocks it again.

In the server, we had to implement gfserver_main and the handler function. Before the handler function would send the header and the data. In this case, I had to make it multithreaded, meaning that the handler now takes in the arguments, and enqueues those information for the server to handle. It locks it before enqueue, once enqueued it broadcast the signal and unlocks the mutex and sets the context to NULL, and changing the ownership of that information. In the server, similarly to the client, the worker threads were created in the boss thread. The worker threads use the request function where, it locks the mutex, pops off the first available request, and then unlocks it. Then it uses the context library to get the file information and then using gfs_send and gfs_sendheader to communicate with the client. Once it finishes, it frees the request item, and once the server finishes, it destroys the queue and frees the rest of the information used, including freeing the content.

No tests were written, just manual testing including using other students' binaries.



References

I used the pthread example from the lecture videos as a source of reference for creating the threads, mutexes, and conditions. Also used this site to better understand pthreads and multithreading, https://computing.llnl.gov/tutorials/pthreads/. Similarly to part 1, I also used gradescope feedback to figure out what was mistaken in my code.

Improvements

Some feeler tests could be given out as part of the assignment. Tests that are not as comprehensive as the ones in gradescope, but ones that simple enough that can help guide us in the correct direction on how to write tests. For me, coming in I had slight knowledge in C and just never done any unit testing for it. Generally, unit tests are written a little differently from the actual code in the language, so it takes more time to understand. Just having an example available can benefit us greatly. I also found parts of the instruction to be confusing. For example, in part 2, I was confused on what we supposed to do for the handler.c file. At first, I thought we were supposed to implement the actual handler in part 1. It wasn't till I peruse piazza that I found out that the handler.c for part 2 had a different purpose. The list of files available and which ones we should edit and submit and so forth is helpful for submitting assignments, but doesn't do much otherwise. More information on how the files work with each other can prove to helpful too.