Project 1 README

Spring 2023
Shide Qiu

Warmup (Echo) Understanding

In the first warmup assignment, the main challenge is to set up a reliable connection between sockets on server and client side with C's standard socket API. The procedures are described below:

1. The server keeps listening on a specific socket.
2. the client connects to server's socket and sends a message to server.
3. The server receives the message and sends it back to the client.
4. The client receives the message and output the message.

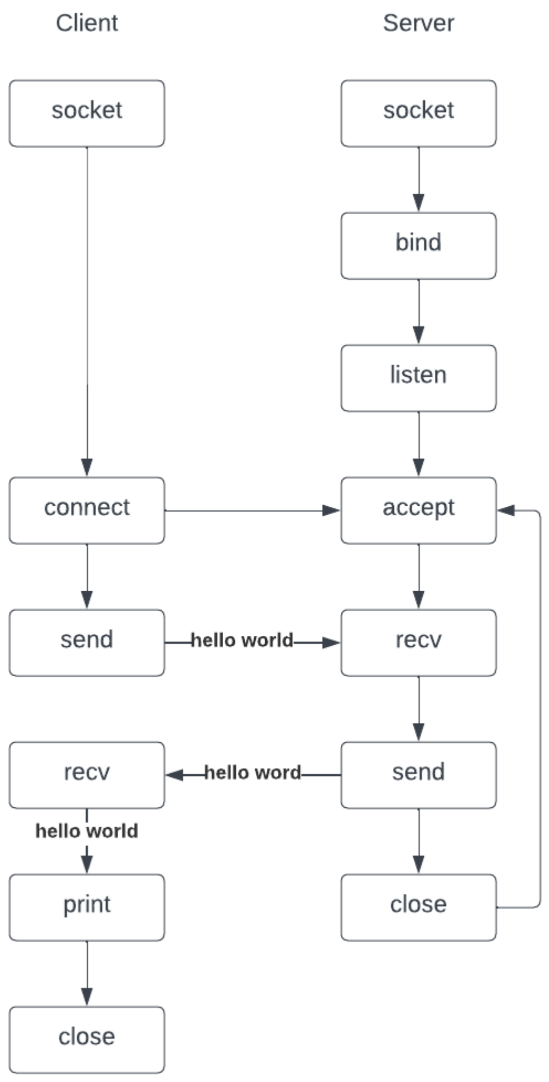
There are some assumptions in this project need to be mentioned. Firstly, the message is no longer than 15 bytes. Since the message is short, it will be sent or received in every single send or recv call. Secondly, the server should not terminate after sending its response. On the other hand, it should keep listening and preparing to serve another request. Thirdly, the program needs to support both IPv4 and IPv6.

Warmup (Echo) Implementation and Testing

The workflow is described in the picture below. According to Beej's Guide to Network Programming, the client and server are built for this project. Since we want to accept both IPv4 and IPv6, we set AP_UNSPCEC for ai_family of addrinfo struct. Also, the SOCK_STREAM is set for ai_socktype to achieve TCP transport protocol. With TCP, the data transfer between client and server is guaranteed. After the preparation work, the sockets on server and client side are both created. Then the server's socket needs to associate (bind) with a port which is used by the kernel to match an incoming packet to a certain process's socket descriptor. It's worth to notice that sometimes it failed when I try to call the bind function. The reason is that the connected socket is still hanging around in the kernel and it's hogging the port. A piece of code need to be added to allow it to reuse the port:

```
int yes = 1;
if (setsockopt(listener, SOL_SOCKET< SOREUSEADDR< &yes, sizeof yes) == -1) {
        perror("setsockopt");
        exit(1);
}
```

The server will keep listening for the incoming connections with listen function. Once the client tries to connect to the server, the server will accept it and get the client's socket information. Until now, the connection between client and server is established. There are different ways to establish the connection, but they all achieve the same result. Beej's method is used because it's easy to set up and the explanation is clear.



Now the client will send a message to the connected server's socket. It only needs to call the send function once because message length is guaranteed to be within 15 bytes. The buffer size is set to 16 and initialize to all 0 since it could tell when the buffer string ends. The server receives the message and send it back to the client. The client receives the same message and prints it out in the end. After this, the client will terminate automatically. However, the server will be ready to accept new connections.

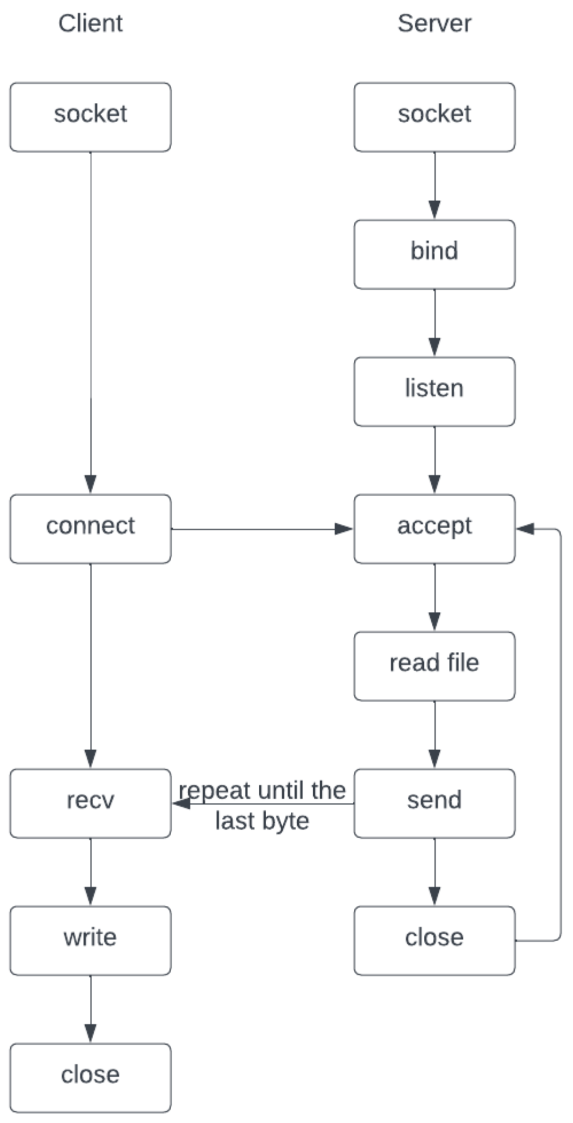
The test is done in the VM. A print function is added in the server code to print out the received message. The client is launched with customized message while the server is running. Both the client and server print the same customized message in the terminal. Leave the server running and launch a new client with another customized message. The new message is print out on both sides. Therefore, I could confirm both server and client work as expected.

Warmup (Transferfile) Understanding

In the second warmup assignment, the first step is the same with the first warmup assignment. We need to set up a reliable connection between sockets on server and client side with C’s standard socket API. However, the server will transfer a file to the client once the connection is established. The client needs to write the received file to a new file. The workflow is shown in the figure below.

Warmup (Transferfile) Implementation and Testing

An explanation of how you implemented and tested your code.



The main challenge is that the file size is unknown, and it is probably larger than the buffer size. Thus, the server needs to keep sending buffer size data until all bytes in the file are sent. After each buffer is sent, the buffer needs to be reset to all 0. Since the last buffer may contain fewer bytes and the 0 will work as a termination signal.

The test is done in the VM. A print function is added in the server code to print out each sent buffer. The client is launched and print the buffer received. Also, file created by the client is compared to the original file.

Part 1 (gfclient & gfserver) Understanding

Clearly demonstrates your understanding of what you did and why - we want to see your design and your explanation of the choices that you made and why you made those choices.

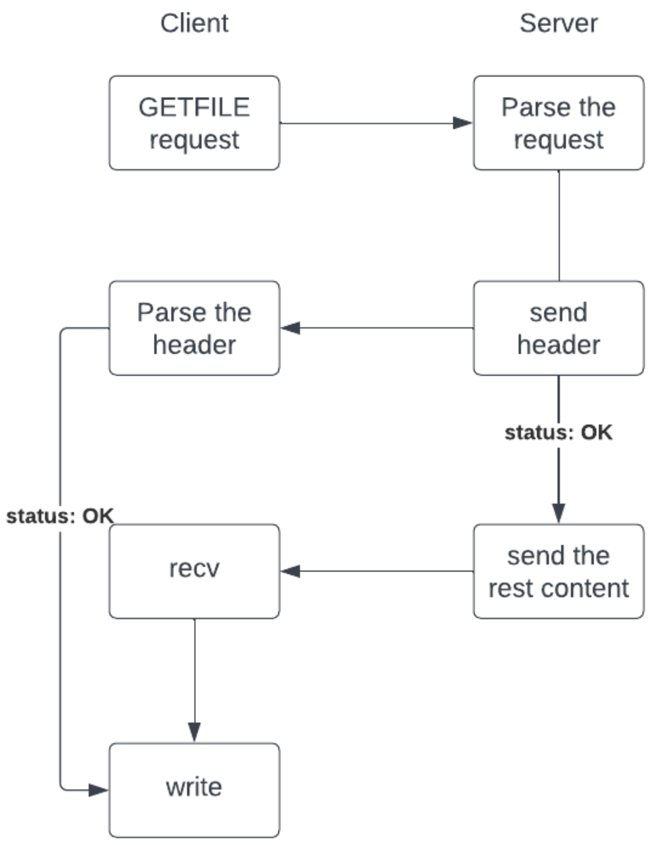
The part 1 is to implement a getfile protocol in client and server sides. The connection between server and client will be skipped here since it is discussed in detail in the warmup assignment. After the connection is established, the following procedures are described below:

1. Client sends a GETFILE request to the server. The format is: <scheme> <method> <path>\r\n\r\n
2. The server receives the request and parses it. The scheme must be GETFILE and the method must be GET. Otherwise, the status will set as INVALID.
3. The server tries to open the file from the path received. It the file does not exist, a FILE_NOT_FOUND status is set. If the file is found, set the status to OK.
4. The server sends the header to the client. The format is: <scheme> <status> <length>\r\n\r\n<content>.
5. The client receives the header and parses it. If the status is OK, write the content to the file.
6. After sending the header, the server keeps sending all remaining content to the client.
7. The client keeps receiving and writing the new content to the file.

The main challenge is that the protocol sending between server and client is not null terminated. We need to add a \0 in the end of each buffer. Otherwise, the buffer will not terminate as expected.

A simplified workflow is plotted in the figure below. The client will initialize the process by sending a GETFILE request to the server. The server will parse the request and define the header according to the parsed result. If the scheme and method do not match with default value (GETFILE, GET), the status will set as INVALID. If the file is not found, the status will be FILE_NOTFOUND. The ERROR status is reserved for when the server is responsible for something bad happening. In conclusion, if the status is not OK, the header format is: <scheme> <status>\r\n\r\n. If the status is OK, the handler will handle the header and the remaining content sending process. It’s important to know that the received GETFILE request is not null terminated. Thus, we must match the \r\n\r\n and add \0 in the end manually.

Part 1 (gfclient & gfserver) Implementation and Testing



On the client side, once it receives the header from the server, it will do the same parsing to understand the server status. If the status is INVALID, it will return -1. If the status is FILE_NOT_FOUND or ERROR, it will return 0. If the status is OK, it will process the contents. There is one detail need to notice is that we should initialize the status as INVALID. The reason is that the header from server may include null as status. In this case, the initial INVALID value will let us exit the processing instead of waiting for nothing.

Once we confirm that the header status is OK. We can start to write the content data in the header. However, we do not know if there is content in the header. We need to track the received bytes of header and minus the bytes of scheme, length, status, space, and \r\n\r\n. If the result is 0, it means there is no content in the header and we could skip to the next step. If the result is not 0, we need to write those content down.

After the work with header, the client will receive and write the remaining data. We keep record the number of received bytes and once it reaches the file size, we terminate the receiving process.

The test is done in the VM. A couple print functions are added in the server and client code to it check if the code works as expected. Firstly, the request protocol is printed on both client and server sides to make sure what we sent is what we received. Secondly, we print out the header sent from server to make sure the parsing process works as expected. Lastly, we print out the number of bytes received for each receiving call and compares it to the file size to make sure we collect the right number of bytes.

With those print functions, we launch the client while the server is running to check if the terminal messages valid and if the new file is created by the client. Also, we edited the workload.txt to add an invalid path. With the invalid path, we test if the header status will be FILE_NOT_FOUND. Moreover, we adjusted the GETFILE request in the client to make it invalid. Then launch the program to check if the header status from server will be INVALID.

Part 2 (multi-threaded gfserver & gfclient) Understanding

In part 2, the focus is the multithreaded programming. In the server side, the boss thread creates multiple worker threads. Also, the boss thread uses the handler function to accept new connection requests and save all those requests into a queue. The worker threads will take the request from the queue and send the file to the client.

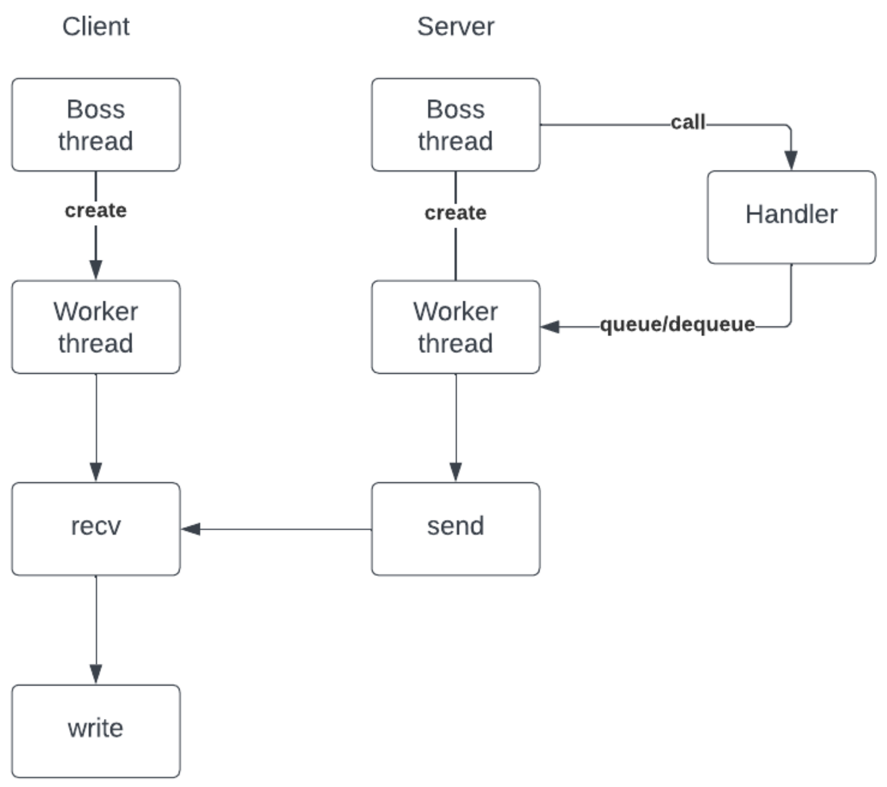
In the client side, the boss thread creates multiple worker threads. Also, the boss thread adds multiple requests to the queue. The worker threads take the request from the queue and sent it to the server. Then the worker threads receive the data sent from server and write it into new files.

The main challenge of this assignment is the race condition. The worker threads may try to access the same element in the queue which could cause error. Thus, more than one mutexes need to be used to solve this issue. The detailed implementation is discussed below.

Part 2 (multi-threaded gfserver & gfclient) Implementation and Testing

The workflow is described in the figure below. In the client side, the boss thread needs 1 m_client and one condition variable cond_client. The m_client mutex is for adding requests into the queue. Every time the boss thread tries to add a new request into the queue, it needs to lock the queue to make sure no other threads can access it. Once it finishes adding the request, it will send a cond_client signal to let other threads know that the queue is unlocked. After adding all requests into the queue, the boss thread needs m_client mutex to add a termination signal into the queue. I use the string “Done” as termination signal. It is important to add this signal because the work threads need to know when to terminate. If we don’t add this signal, the work threads will keep waiting for new requests adding into the queue and result in memory leak. After adding the termination signal, the boss thread will issue a cond_client signal to let other threads know they can access to the queue.

The worker threads are created by the boss thread. They will dequeue request from the queue created by the boss thread. According to the request path provided in the queue, the worker threads start to send request to server and prepare to receive the data from the server. After receiving the data, the work threads will write the data into a new file. Whenever a worker thread tries to dequeue a request from the queue, the queue needs to be locked by m_client mutex to make sure no other worker threads can dequeue the queue at the same time. If there is no request in the queue, the worker thread will wait for the cond_client. Once the worker thread grabs the request from the queue, it will unlock the queue and send cond_client signal to other thread and let them access the queue. The worker thread will then check if the grabbed request is the termination signal “Done”. If it is “Done”, the worker thread will exit since there is no more thing in the queue. Before the exit, the worker will add the “Done” back to queue so that other worker threads can use it as the termination signal and exit. This step is very important since there are multiple other worker threads waiting for new request adding to the queue. If the “Done” is not added back to the queue, all such worker threads will be stuck there forever. If the grabbed element is not “Done”, the worker thread will continue to finish the receiving and writing process.



In the server side, there is one m_server mutex and one cond_server mutex implemented. The boss thread will use the gfs_handler function to keep listening to the incoming requests from the client. Once it receives a request, it will add it into the queue. Similar with the client side, when the worker thread tries to add new request into the queue, it needs to use m_server mutex to lock the queue so that it can make sure no other threads can access it at the same time.

The worker threads are created by the boss thread. Each worker side will try to get a request from the queue created by the worker thread and send the data to client. When the worker thread tries to dequeue the queue, it needs to lock the queue like the client worker thread does. If there are no requests in the queue, the worker thread will simply be waiting there until new requests are added in.

The test is done in the local VM. A couple print functions are used for debugging purpose. In the client side, I print a message with thread id to confirm all worker threads are created as expected. Another message with thread id is also printed when the worker thread is joined to the boss thread. The numbers are matched in boss message, and it makes sure that the boss/client model work as expected.

In the server side, I record the number of times for gfs_handler function is called to check if all requests are received in the server boss thread.

In the end, I adjusted the number of request in the client side and check the number of new files created by the client to confirm the communication between server and client works as expected.

Reference

1. Beej’s Guide to Network Programming. https://beej.us/guide/bgnet/html/
2. Socket Tutorials https://www.linuxhowtos.org/C_C++/socket.htm
3. POSIX Threads Programming https://hpc-tutorials.llnl.gov/posix/
4. High-level code design of multi-threaded GetFile server and client https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit
5. Learn C in X minutes https://learnxinyminutes.com/docs/c/
6. Short introduction to header files in C https://www.youtube.com/watch?v=u1e0gLoz1SU
7. What is an opaque pointer in C https://stackoverflow.com/questions/7553750/what-is-an-opaque-pointer-in-c
8. Operating System – Three Easy Pieces