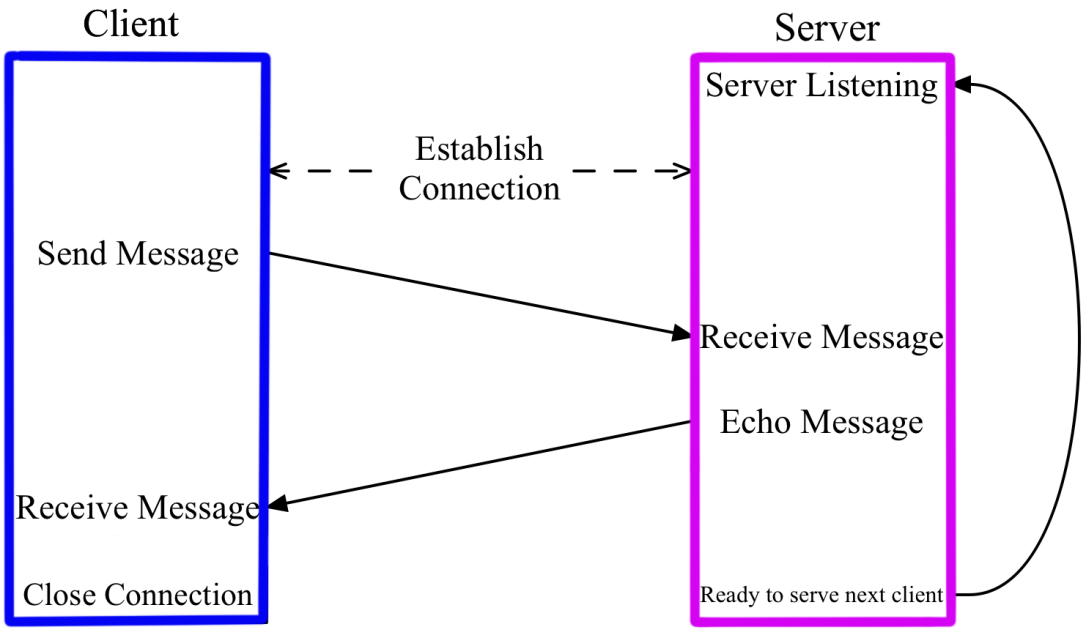
Table of Contents

Project Design:

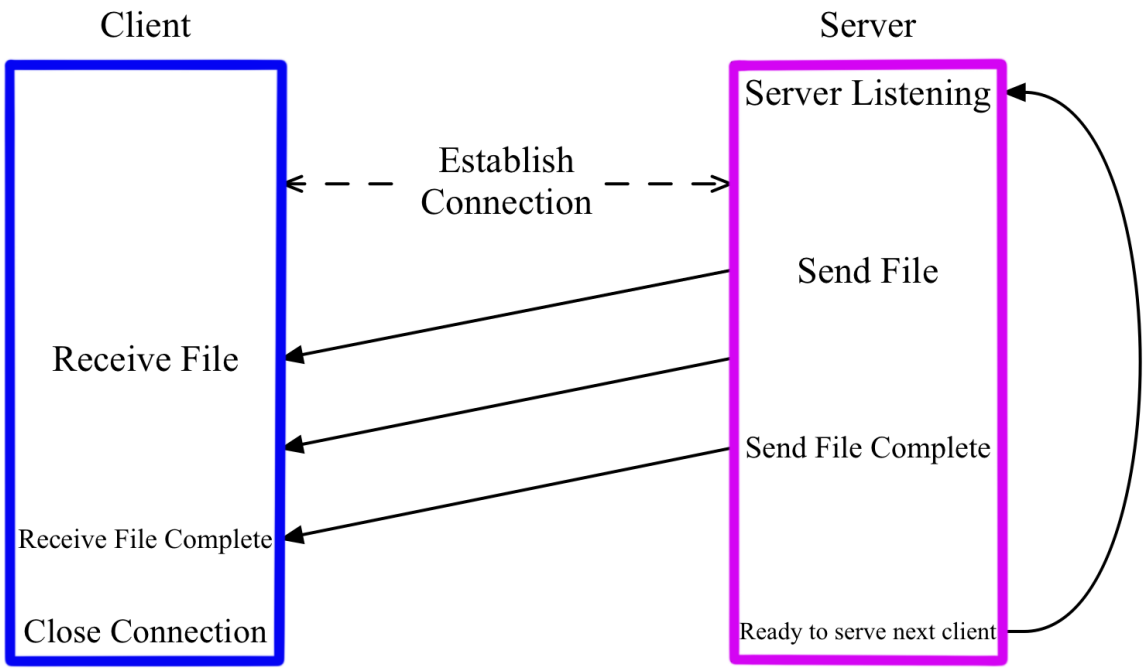
Echo Client-Server

1. A client-server pair using TCP sockets.
2. Client sends a message to the server.
3. Server receives the message and sends the same message back.
4. Client and server would be IP agnostic.
5. The message will not be longer than 15 bytes.



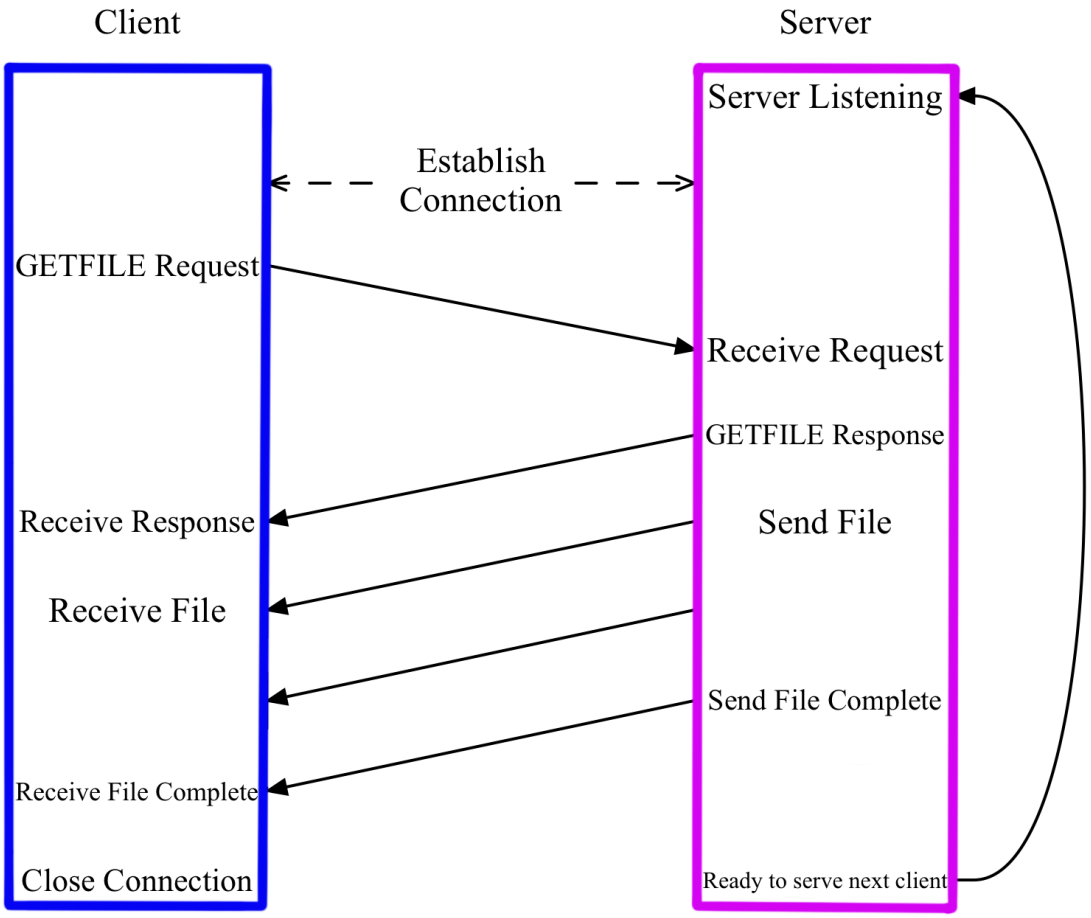
File Transfer Client-Server

1. Server listens on a socket.
2. As soon as a client connects, server sends the client a file.
3. The data is sent in chunks.
4. The server waits for the entire data to be received.
5. Once the transfer is complete, the server goes back to listening on the socket, ready to serve another client.



GETFILE Client-Server

1. Transfer client-server using the GETFILE protocol.
2. GETFILE protocol request (client sends to the server) –
   - GETFILE GET <path>\r\n\r\n
   - E.g. GETFILE GET /foo.jpg\r\n\r\n
3. GETFILE protocol response (server sends to the client) –
   - If no error in the request header and the file exists: GETFILE <status> <length>\r\n\r\n<data> E.g. GETFILE GET /foo.jpg\r\n\r\n
   - If data cannot be sent: GETFILE <status>r\n\r\n E.g. GETFILE INVALID\r\n\r\n
4. "Status" in the server response must be one of – "OK," FILE_NOT_FOUND," "ERROR," or "INVALID."
5. "\r\n\r\n" marks the end of a header.
6. "Length" in the server response is the size of the file.



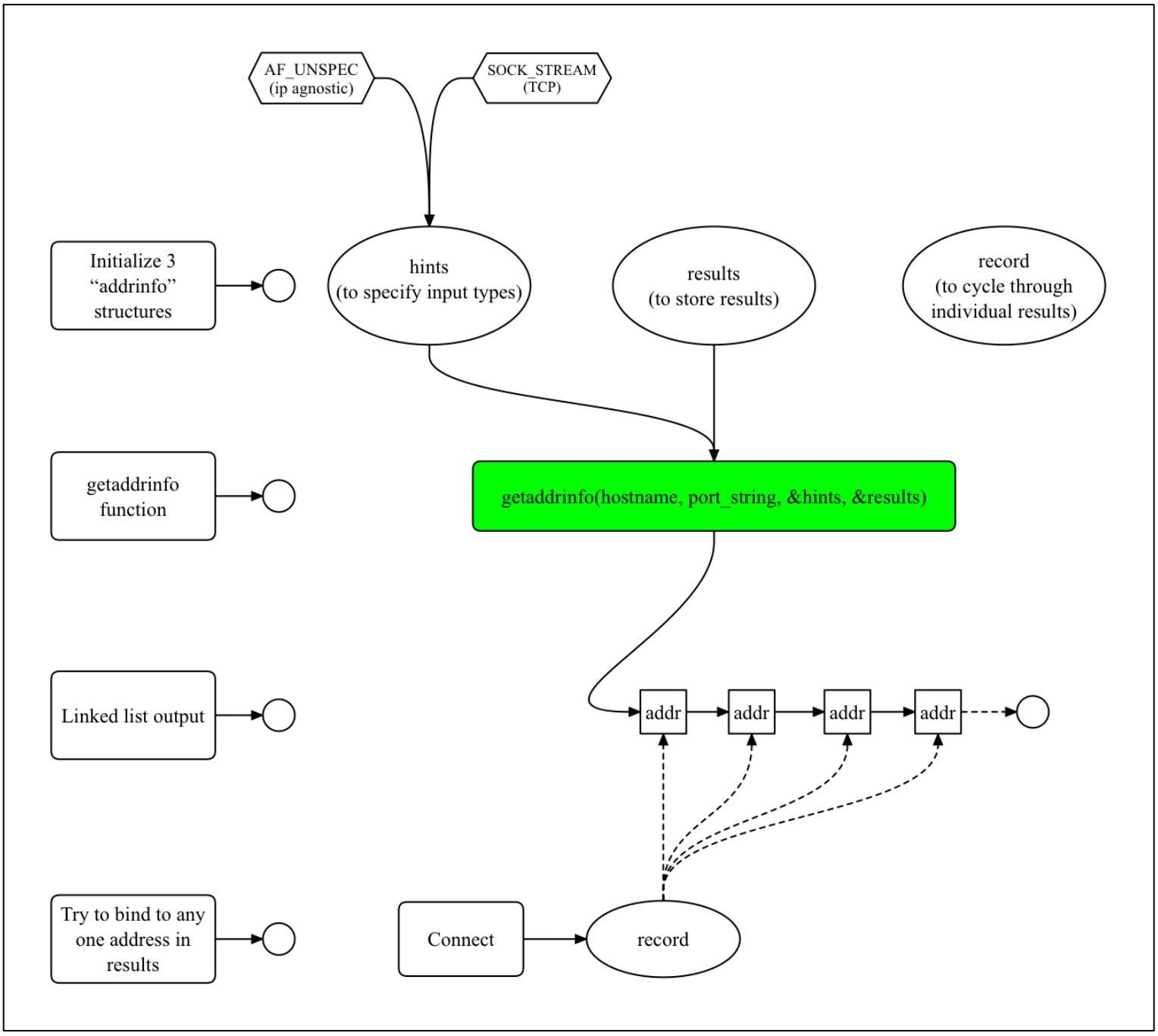
Multithreaded GETFILE Client-Server

1. Client and server implementing the GETFILE protocol just like before.
2. Additionally, both the client and the server would use the boss-worker multithreading pattern to request and receive, and respond and send multiple files at once, respectively.
3. Both the client and server would use shared work queues.
4. Both the client and server would use mutexes to access the shared queue.
5. Both the client and server would use conditional variables to coordinate access to the shared queue.

Critical choices and Trade-offs.

getaddrinfo() [1]

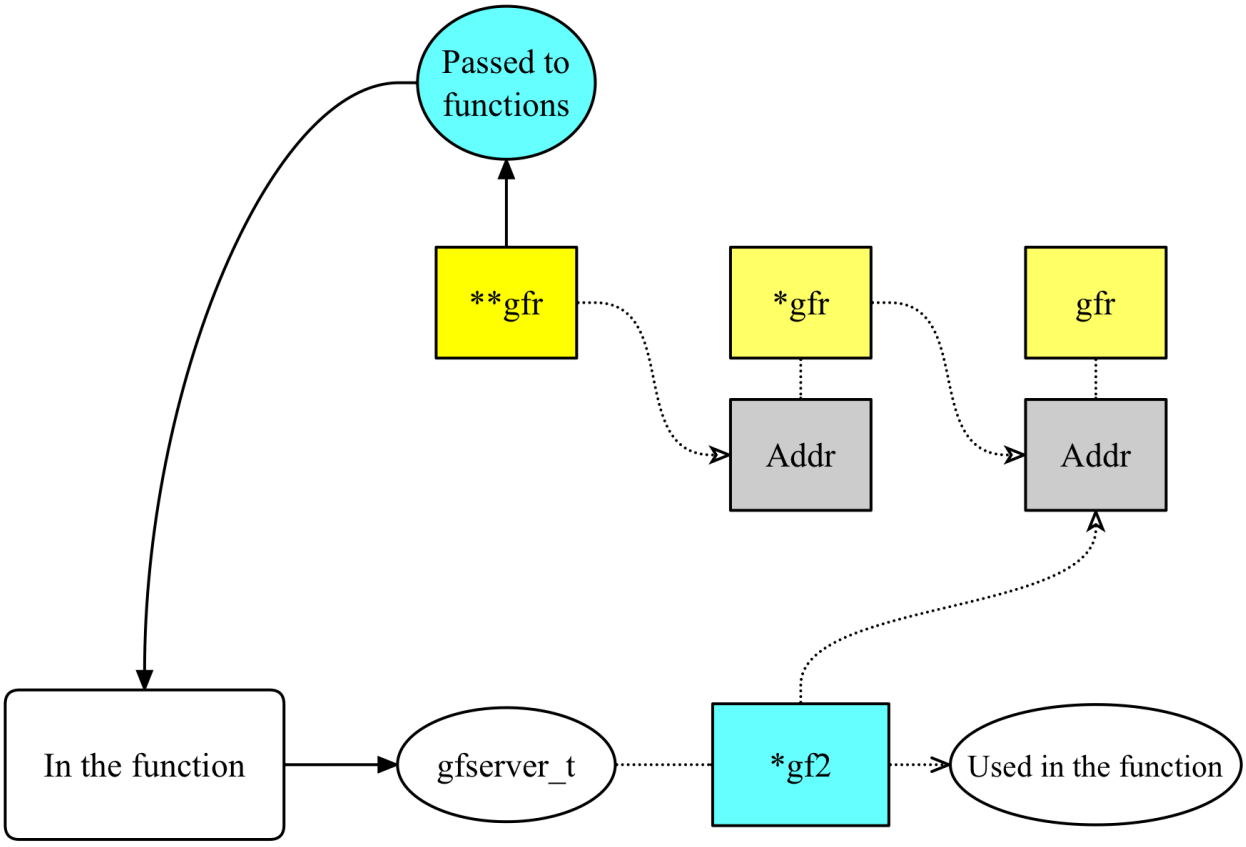
As the project required, the implementations had to be IP agnostic. After much research, using the getaddrinfo function seemed to be the most straightforward way to make the program work with IPv4 and IPv6. It works as below.



Opaque pointer [2]

As the project required, the server context, "gfr" is passed as an opaque pointer to all the other functions that help serve a request. Correspondingly, the client context, "ctx" is passed as an opaque pointer to all the server functions. The reason being, we should hide the implementation details from the users of the library.

To use this opaque pointer efficiently in each of the functions, we then define a new pointer *gf2 that points to gfr. As expected, there are other ways to deal with this, but this seemed to be the most straightforward. The approach can be seen more clearly below.

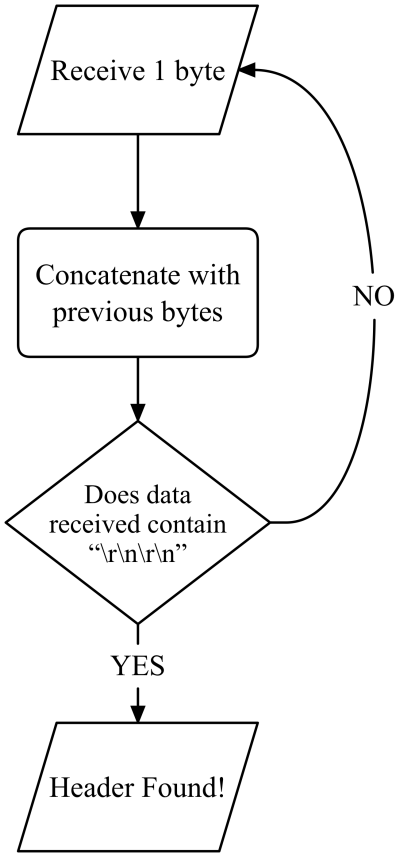


Parsing the header [1] [3]

A well-formed header starts with the scheme, "GETFILE" and ends with the end string, "\r\n\r\n" – the header can be received in one data segment or can be partially received in one and the rest in the next. In the header sent by the server to the client, the segment containing the entire header can also contain following data.

One way to deduce the header would be to keep checking the presence of the end string in each segment, and if found, separate the header and the data following the header (if data is present).

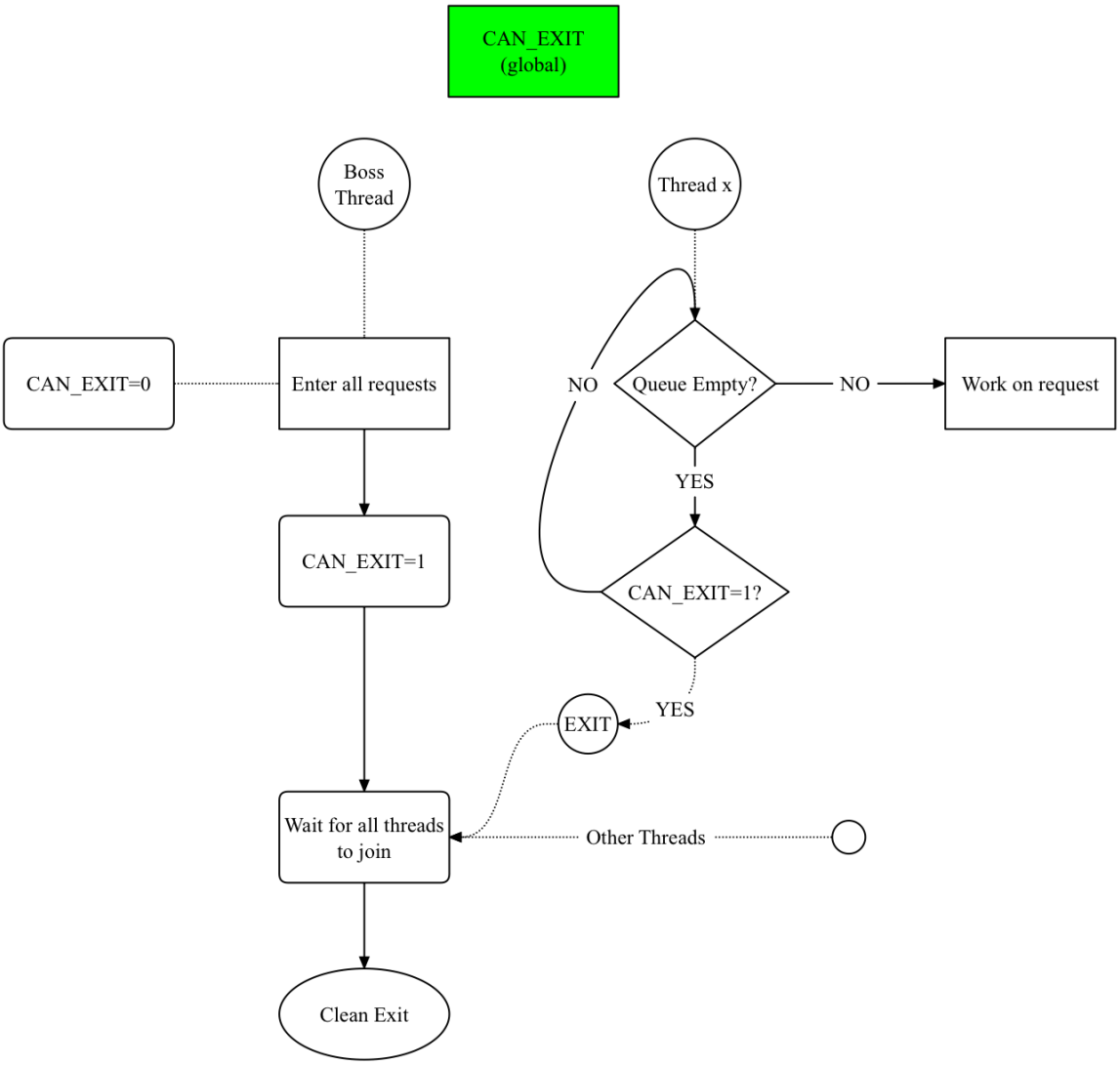
Though, the approach that consistently worked in all scenarios, reliably, was to receive one byte at a time until we see the end string, concatenate all the bytes to form our header, and start receiving bigger chunks for the data. The approach is given below.



When should the threads exit? – in multithreaded gfclient [4] [5] [6]

Unlike the multithreaded gfserver that runs in a never-ending loop, the multithreaded client has to make a clean exit as soon as all the requests in the queue have been worked on. Can the threads check if the queue is empty and exit if it is? Well, the threads are created before anything is put into the queue, so, on the initial check, the queue will be empty. Hence, we run a risk of the threads exiting without doing any work.

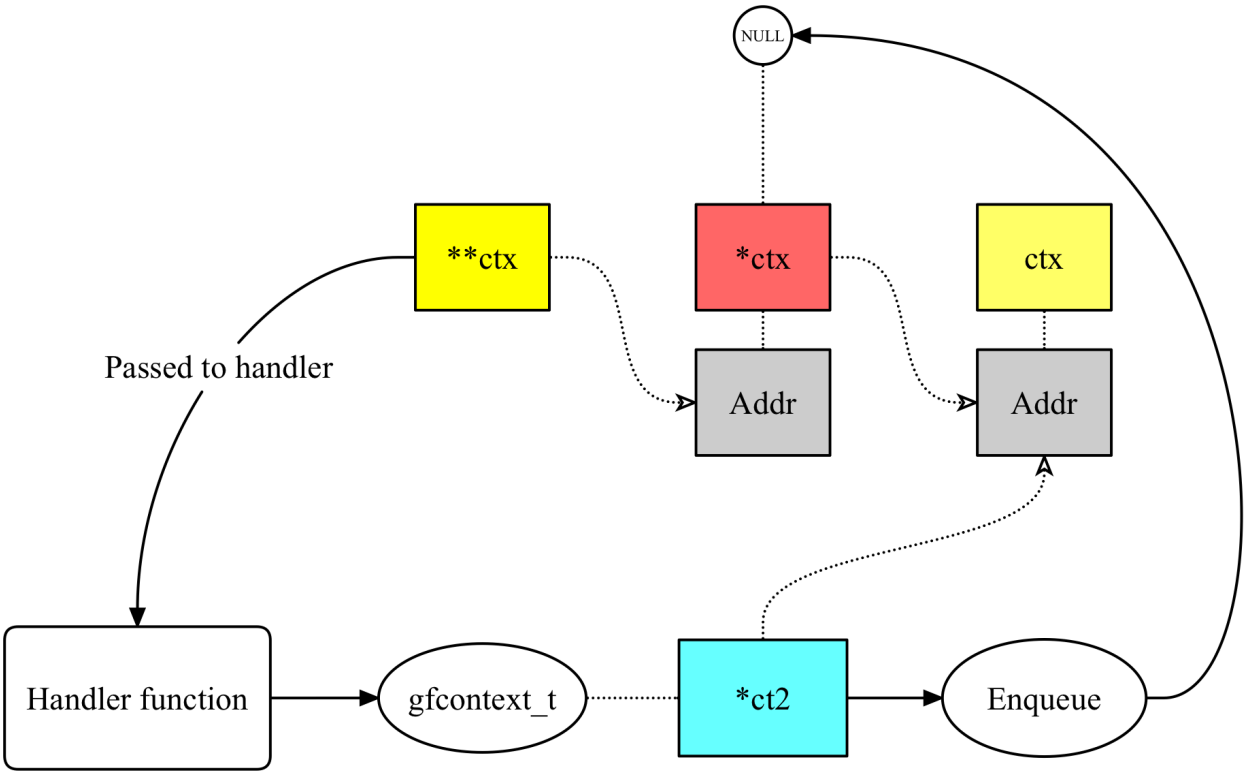
Again, there are several ways we could deal with this. The approach we went for is - using a global variable (CAN_EXIT) that is initialized to 0, and the boss threads sets it to 1 *after* all the requests are entered into the queue. Each thread, if finds the queue empty, checks the variable, and if it is 1, exits. *Please note that all the queue and CAN_EXIT operations are done using a mutex.*



Making client context, "ctx" equal to NULL – in multithreaded gfserver [2]

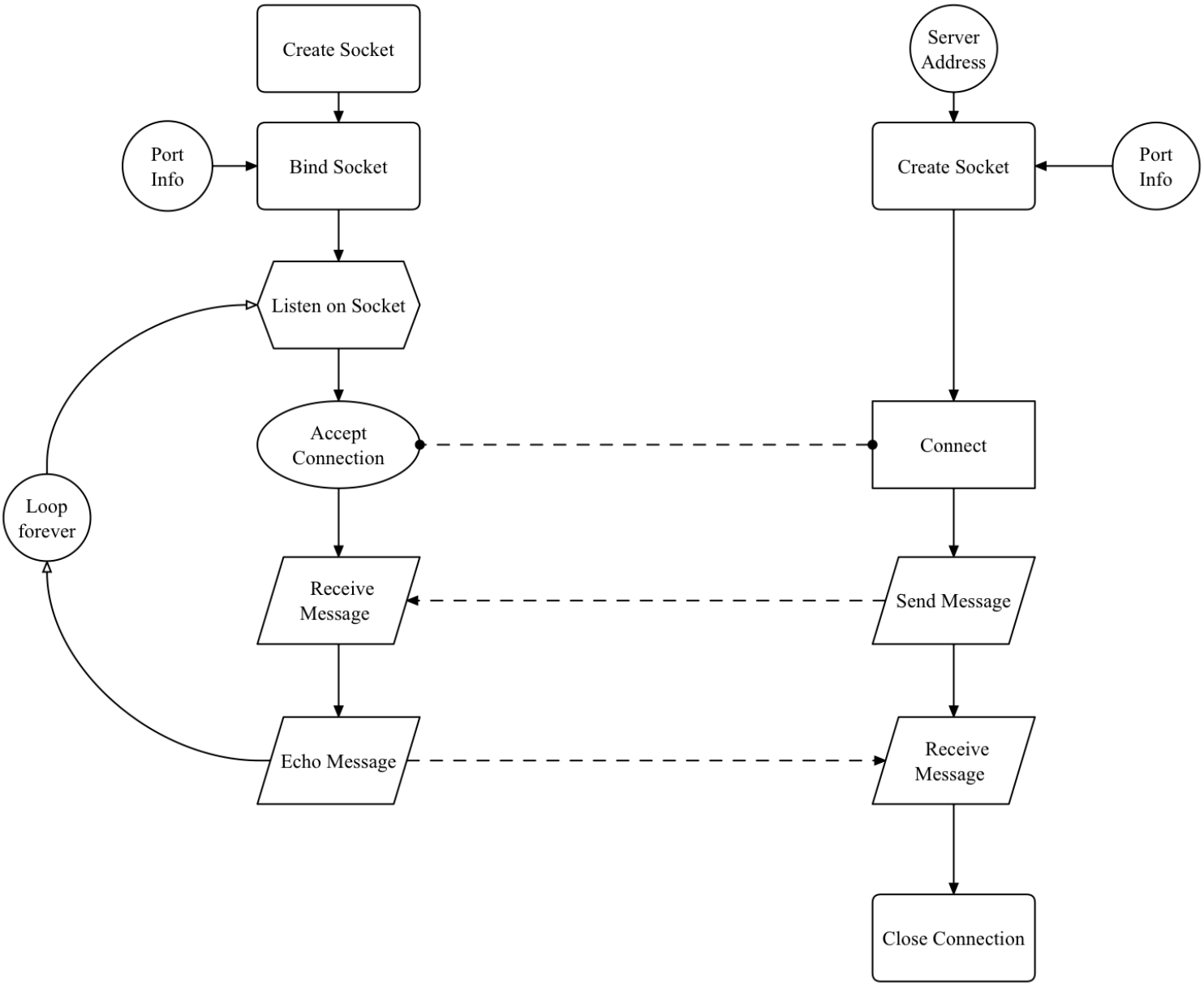
The opaque pointer, **ctx is passed to the handler function, which needs to be pushed into the queue for the threads to work on, but the handler has to make ctx NULL *before* exiting the handler function (which could be before the request is claimed by any thread).

One way around this, is the approach we discussed in the opaque pointer step above. We create a new pointer *ct2 of *struct gfcontext_t* and make it point to ctx. ct2 is then enqueued in the queue, and *ctx can be made NULL without losing the context.

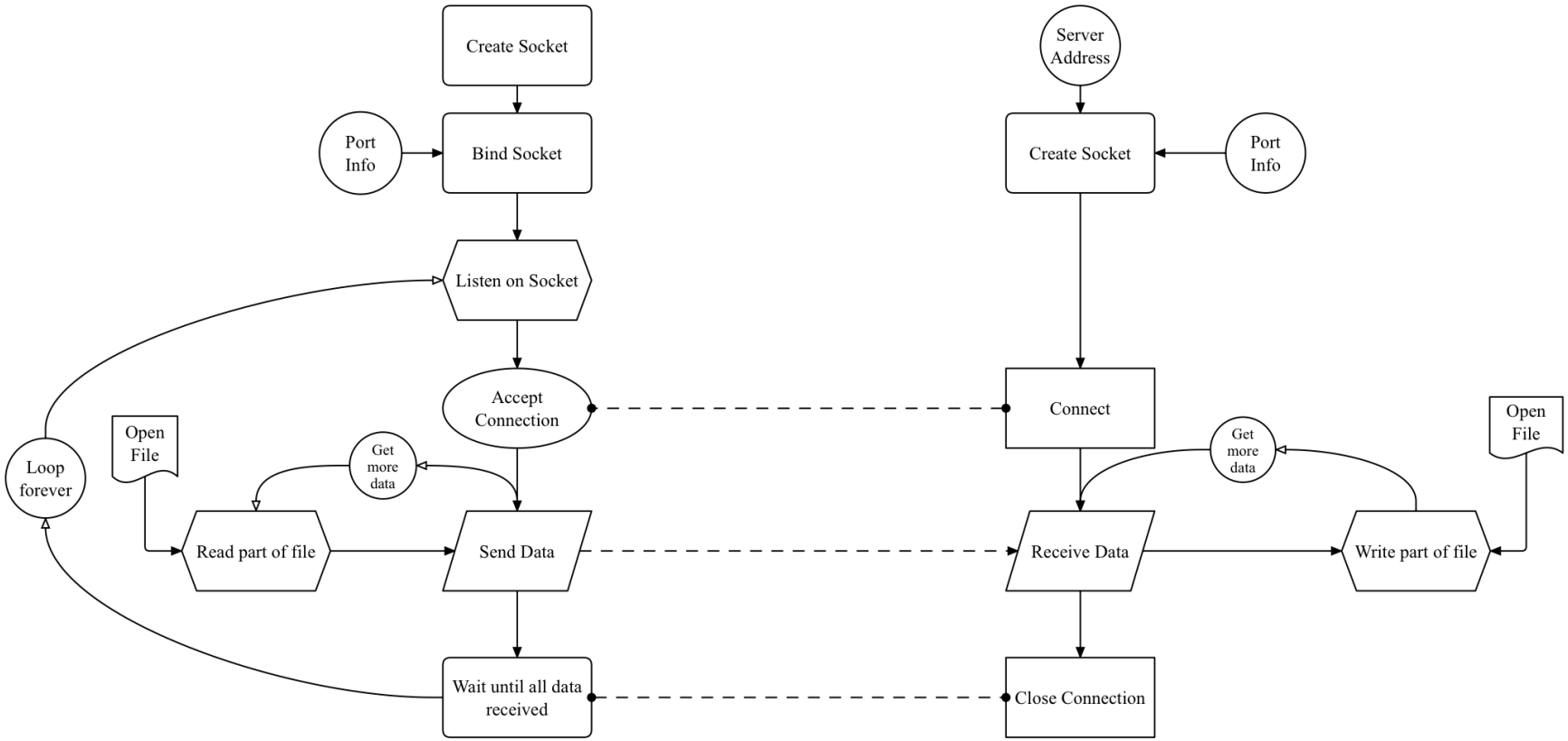


Flow of control

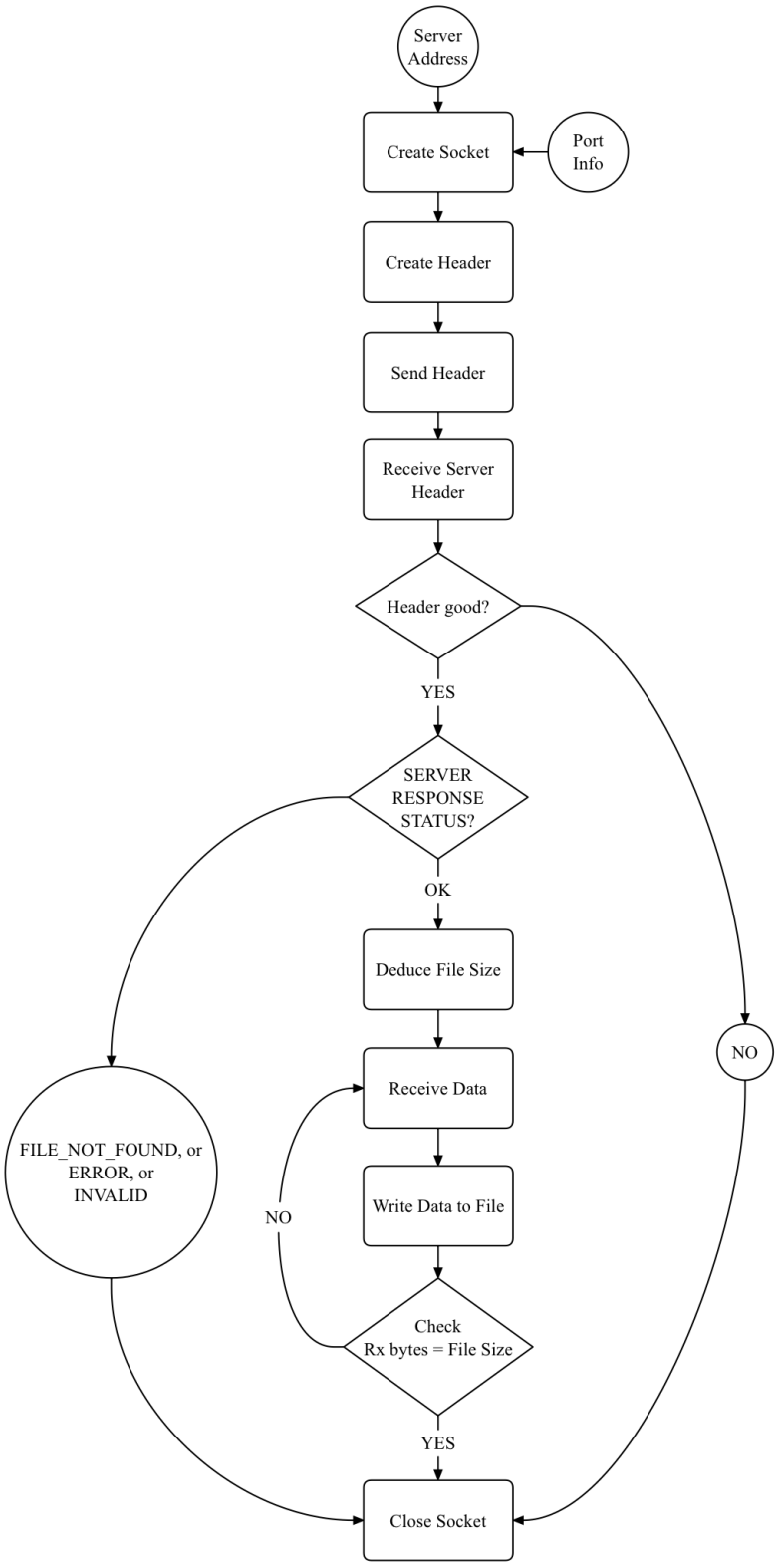
Echo Client-Server (Client on the right; server on the left)



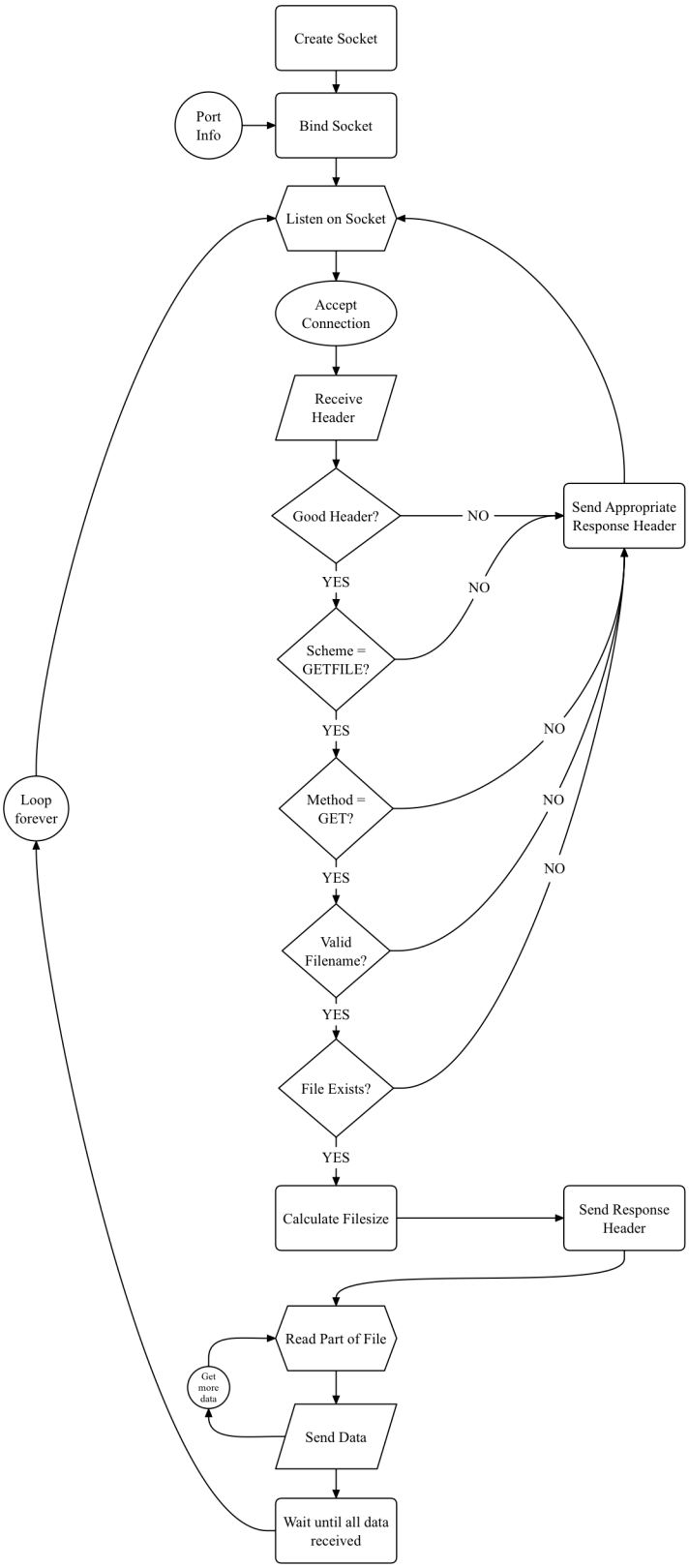
Transfer File Client-Server (Client on the right; server on the left)



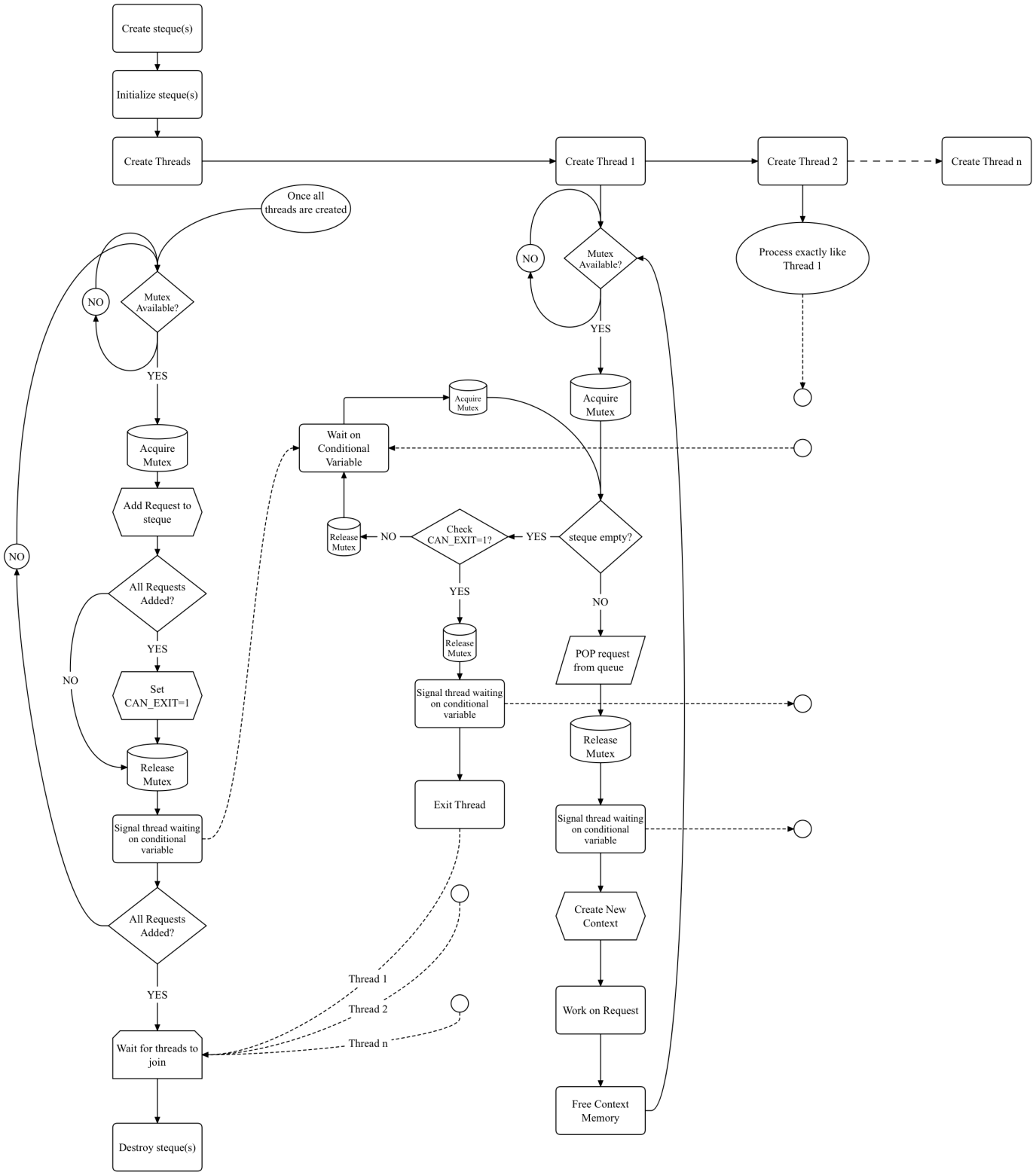
GETFILE Client



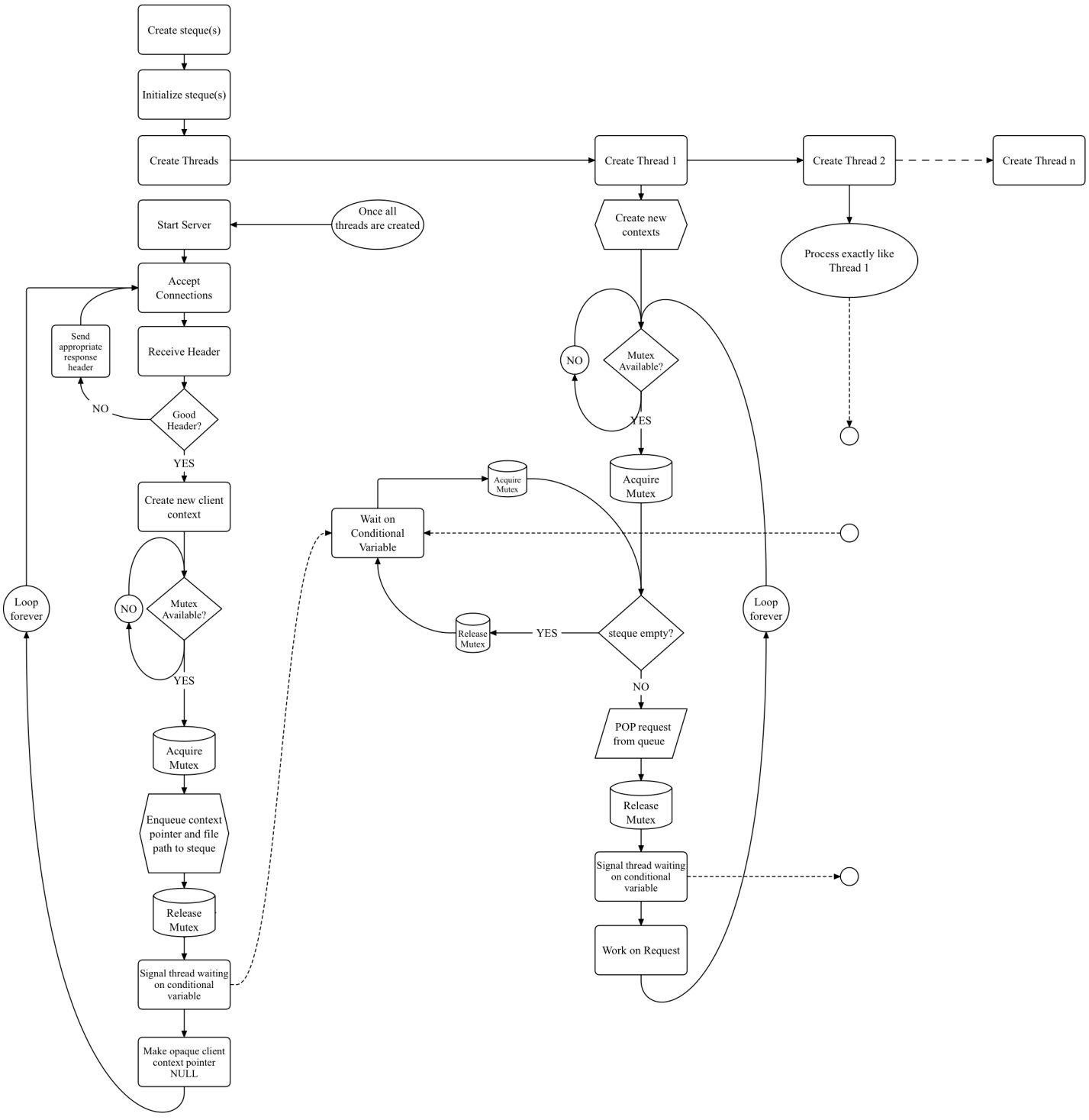
GETFILE Server



Multithreaded GETFILE Client



Multithreaded GETFILE Server



Code Implementation

Echo Client: [1] [3]

1. Build a socket and connect (using the *getaddrinfo* function).
2. Send a message and wait to get the same message back from the server.
3. Print the message and close the socket.

Echo Server: [1] [3] [7]

1. Build a socket and bind *(getaddrinfo).*
2. Listen on the socket and accept a client connection.
3. Receive a message from the client.
4. Print the message and send the same message back to the client.
5. Go back to listening.

Transfer Client: [1] [3] [7]

1. Build a socket and connect *(getaddrinfo).*
2. Receive the file in parts sent by the server and write it to a file.
3. Once all the data is received, close the socket.

Transfer Server: [1] [3] [8]

1. Build a socket and bind *(getaddrinfo).*
2. Listen on the socket and accept a client connection.
3. Open a file, read the file in parts and send the file in parts to the client.
4. Once all the data is sent, wait for all the data to be received by the client.
5. Once all the data is received, go back to listening.

GETFILE Client: [1] [3] [9] [2] [10]

1. Build a socket and connect *(getaddrinfo).*
2. Create a GETFILE header with the path to the file required and send the header to the server *(gfc_perform).*
3. Receive the header and review the status sent by the server *(gfc_perform).*
4. If the status is anything but *OK*, close the connection *(gfc_perform).*
5. If the status is *OK*, start receiving the file in parts sent by the server *(gfcperform)* and write *(writecb)* it to a file.
6. Once all the data is received, close the socket *(gfcperform),* and return success *(0)* to the main function.
7. If the server disconnects before all the data is received, return an error *(-1)* to the main function.

GETFILE Server: [1] [3] [2] [10] [11]

1. Build a socket and bind *(gfserver_serve).*
2. Listen on the socket and accept a client connection *(gfserver_serve).*
3. Receive the header from the client and make sure the format is correct. If not, send an *INVALID* status header to client and go back to listening *(gfs_sendheader).*
4. Check if the file requested exists. If not, send a *FILE_NOT_FOUND* status header to the client and go back to listening *(gfs_sendheader).*

5. The handler could also want to send an *ERROR* status header *(gfs_sendheader).*
6. If the request header is correct and the file exists, send an *OK* status header *(gfs_sendheader)*, open the file, read the file in parts and send the file in parts to the client *(gfs_send).*
7. Once all the data is sent, wait for all the data to be received by the client.
8. Once all the data is received by the client, go back to listening.

Multithreaded GETFILE Client: [1] [3] [4] [10] [5] [6] [2] [12] [13]

1. The boss thread will create (*pthread_create)* the specified number of worker threads *(nthreads).*
2. The boss thread will enqueue the requests *(nrequests)* (using *mutex*) in the queue (and make global variable, *CAN_EXIT* = 1) (signal on *conditional variable* to let waiting thread know).
3. For each worker thread (thread handler = **worker_handler*),
    - Wait till there is something in the queue (check queue using mutex, wait on *conditional variable*).
    - Get a request off the queue (using *mutex*).
    - Build a socket and connect *(gfserver_serve)*.
    - Create a *GETFILE* header with the path to the file required and send the header to the server *(gfserver_serve)*.
    - Receive the header and review the status sent by the server *(gfserver_serve)*.
    - If the status is anything but *OK*, close the connection *(gfc_get_status)*.
    - If the status is *OK*, start receiving the file in parts sent by the server and write it to a file *(writecb)*.
    - Once all the data is received, close the socket.
    - Get another request off the queue (using *mutex*).
    - If the queue is empty, exit, and join the boss thread (if global variable, *CAN_EXIT* is 1).
4. The boss thread will wait for all worker threads to exit and join *(pthread_join)*.
5. Once all worker threads join the boss thread, destroy the queue *(steque_destroy)*.

Multithreaded GETFILE Server: [1] [3] [4] [10] [5] [6] [2] [14]

1. The boss thread will create the specified number of threads (*nthreads*).
2. The boss thread will build a socket and bind.
3. The boss thread will listen on the socket and accept client connections.
4. Every time a client sends a request, the boss thread will receive the header from the client and make sure the format is correct. If not, send an *INVALID* status header to client and go back to listening.
5. If the request header is correct, the boss thread will enqueue it in the queue, and go back to listening (*gfs_handler* in *handler.c*, using *mutex*) (and *signal* on the *conditional variable* to let waiting thread know).
6. For each worker thread,
    - Wait till there is something in the queue (check queue using *mutex*).
    - Get a requested path off the queue (using mutex).

- Check if the file requested exists. If not, send a *FILE_NOT_FOUND* status header to the client (*gfs_actual_handler* in *handler.c* using *gfs_sendheader*) and get another request off the queue (using *mutex*).
- If the request header is correct and the file exists, send an *OK* status header, open the file, read the file in parts and send the file in parts to the client (*gfs_actual_handler* in *handler.c* using *gfs_sendheader*).
- Once all the data is sent, wait for all the data to be received by the client.
- Once all the data is received, get another request off the queue (using *mutex*).
- If the queue is empty, wait (on the *conditional variable*) till the boss thread adds a request to the queue.

Testing

- The testing for this project was done using Python.
- All the clients were tested using appropriate Python servers, and all the servers were tested using the appropriate Python clients.
- The *Pytest* module was used for the unit tests. [15]
- Various file types with sizes ranging from 1KB to 100MB were used for the testing.
- A huge part of the testing focused on header logic, which was tested using a multitude of header possibilities – incorrect headers, partial headers, split headers, headers with data in the same segment, and many more.
- Although Gradescope was not intended to be used as a test harness, a lot of times, Gradescope revealed further avenues for testing.
- Compared to implementing the clients and servers written in C, the Python clients and servers took a fraction of the time.
- Since the transfers happen instantly when run locally, delays were added to the Python test clients and servers to make sure they were handled as expected.
- There were multiple tests using control data and also a mixture of control and normal data.
- In addition, valgrind was used to make sure there are no memory leaks. [16]
- For the multithreaded client and server, the tests were timed and compared to the exact requests ran on the single threaded counterparts, to confirm multithreaded gains.
- ***All the Gradescope tests passed for each of the project sections.***

Works Cited

[1] B. Hall, "Beej's Guide to Network Programming," [Online]. Available: https://beej.us/guide/bgnet/html/.

[2] "Opaque pointer," [Online]. Available: https://en.wikipedia.org/wiki/Opaque_pointer.

[3] M. Kerrisk, "The Linux man-pages project," [Online]. Available: https://www.kernel.org/doc/man-pages/.

[4] A. Gavrilovska, *Georgia Institute of Technology, CS6200 Lectures.*

[5] A. D. Birrell, "An Introduction to Programming with Threads," 1989.

[6] B. Barney, "POSIX Threads Programming," Lawrence Livermore National Laboratory, [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/.

[7] A. Raman, *Echo Client Code.*

[8] A. Raman, *Echo Server Code.*

[9] A. Raman, *Transfer Client Code.*

[10] "C library function - free()," [Online]. Available: https://www.tutorialspoint.com/c_standard_library/c_function_free.htm.

[11] A. Raman, *Transfer Server Code.*

[12] "Understanding “extern” keyword in C," [Online]. Available: https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/.

[13] A. Raman, *GETFILE Client Code.*

[14] A. Raman, *GETFILE Server Code.*

[15] "Pytest," [Online]. Available: https://docs.pytest.org/en/stable/.

[16] "The Valgrind Quick Start Guide," [Online]. Available: https://www.valgrind.org/docs/manual/quick-start.html.

[17] A. Raman, *Multithreaded GETFILE Client Code.*

[18] A. Raman, *Multithreaded GETFILE Server Code.*