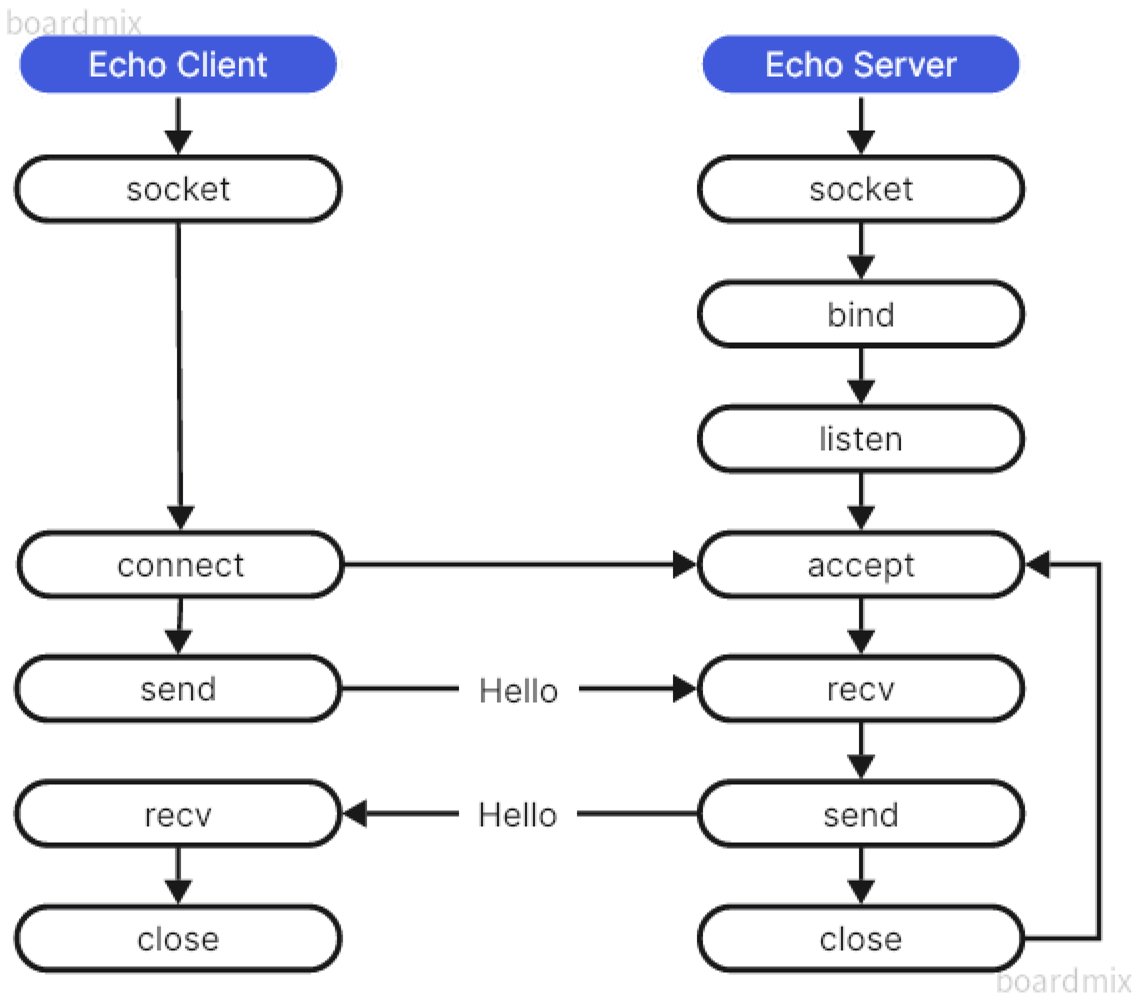
Project README file

Introduction

This is GIOS Project 1 Spring 2024 Readme file from course CS6200. It has four sections of project I.

1. Echo Client-Server
2. Transferring a File
3. Getfile Protocol
4. Multithreaded Getfile

Echo Client-Server



EchoClient and the EchoServer as shown above. The server echoes (repeats back) the message sent by the client.

In order to support both IPv4 and IPv6, used hints.ai_family = AF_UNSPEC and ai_socktype to SOCK_STREAM for TCP protocol; Furthermore the program checks if the provided server address is IPv4 or IPv6 using inet_pton and adjusts the address family accordingly.

For the echoclient, once the socket file descriptor is created, the connect method is called upon to connect with the server.

For the echoserver, following the creation of the socket, the server socket is configured with the SO_REUSEADDR option to enable reusing the address for subsequent. The socket file descriptor is then bound to an available address using the bind() method. To manage incoming client connections, the listen() function is called on the echoserver socket, specifying the maximum number of client connections it can handle.

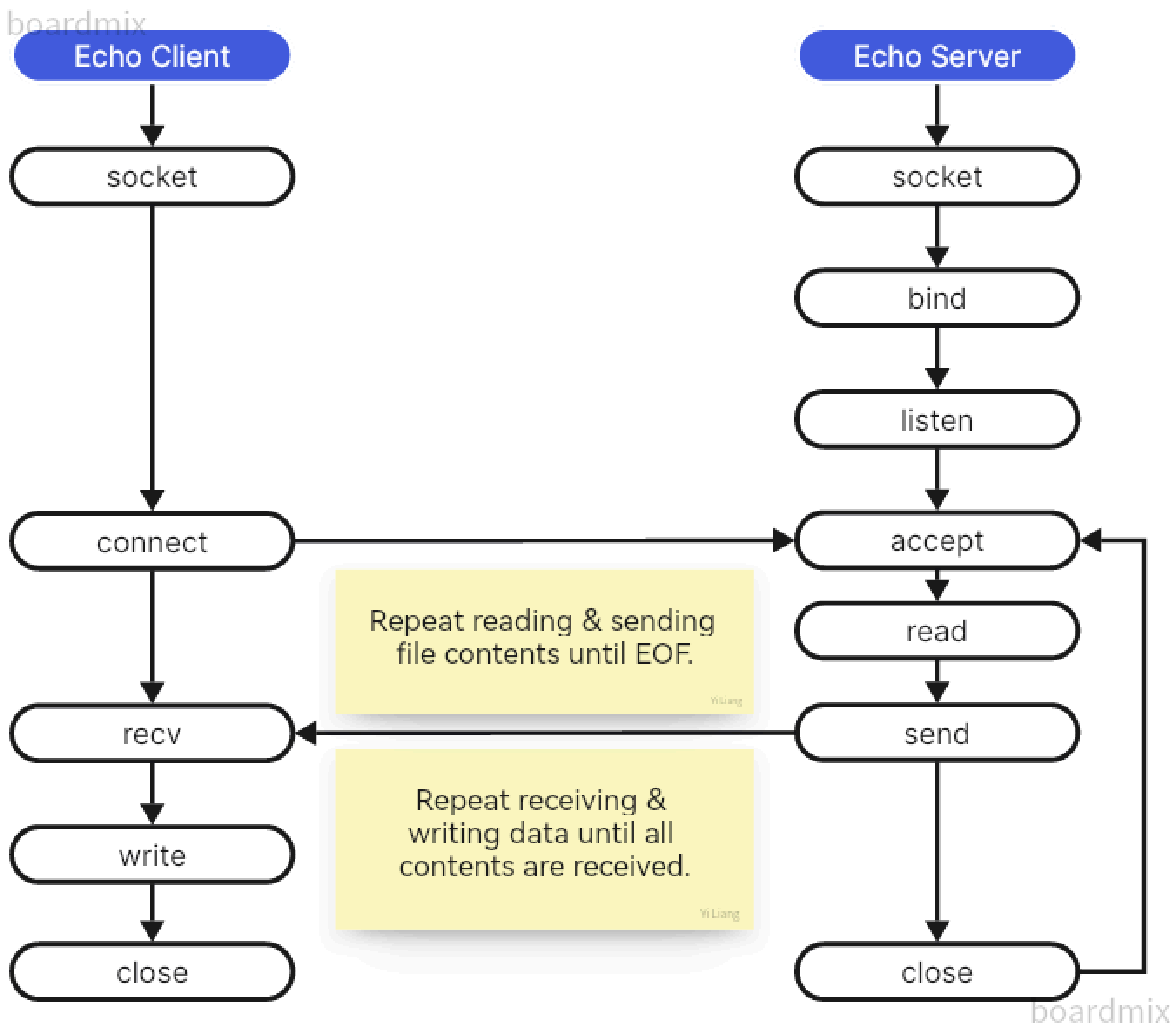
Used getpeername() to determine who sent the echo so that we can respond.

Testing:

Testing done by running the executable with different combinations of command line options like various hostnames, port numbers, and messages.
Successful printing of message passed from client to server and vice versa.

Transferring a File

The main idea of this part of the project, is to implement file operation to send() and recv(). A deeper implementation of Echo Client-Server.


Echo Client
socket
connect
recv
write
close
Echo Server
socket
bind
listen
accept
read
send
close
Repeat reading & sending file contents until EOF.
Repeat receiving & writing data until all contents are received.
boardmix

Data retrieved from the file is stored in a buffer along with the number of bytes read. Due to the nature of transferring data between sockets over a network, it cannot be guaranteed that the entire content will be transmitted or received in a single operation.

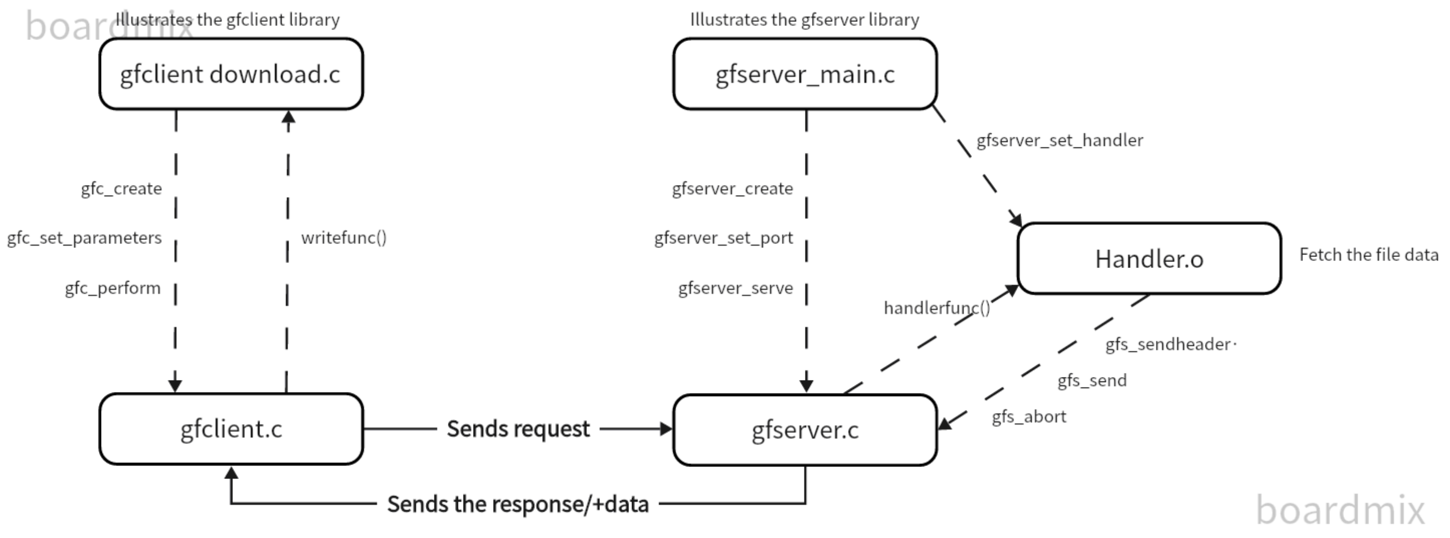
As a result, the read/write and send/receive processes must be iterated to ensure the successful transfer of the file. Also, each iteration of the receive operation provides the number of bytes received and complete the write operation. Since the client is unaware of the file's size, the receive operation is repeated indefinitely to ensure the complete transfer of all data. Once the data transfer is complete, the server closes the socket, signaling to the client that the file transfer has concluded.

Testing:

Using name of the file as a command line argument, the transfer server effectively transmitted the file content. This was confirmed by printing the total number of bytes sent with the number of bytes received on the transfer client's end. The transferred file content was successfully viewed and double checked.

Part 1. Getfile Protocol

In this part, we push even further the ideal from Transfer client / server. We use library to standardize the program, which has a pair of client / server interact with each other using the GETFILE protocol. The gfserver responds to the client's parsed request with file status, size, and data obtained from the registered callback function. The client, upon receiving the server's response, proceeds to save the file content using the registered write callback.



gfclient:

The client_download.c leverages methods like gfc_create and gfc_perform in gfclient.c by passing the opaque pointer gfcrequest_t *gfr. Struct pointer **gfr, comprising elements such as server, port, writefunc callback, status, and headerfunc callback, is utilized across methods for connection establishment and file transfer.

In gfc_perform, first part of code is inherited from previous Echo client/server and Transfer part to solve host name and establish connection with server. Then client sends the request header string, following the GETFILE protocol with the workload path. When client receives a response header, it decodes file status, size, an end marker, and subsequent file content in bytes.

Continuous reception of server responses checks for the header marker '\r\n\r\n'. Each recv call provides the number of received bytes.

Using pointer arithmetic, the buffer pointer moves past the marker, and the file data is streamed to the registered write function callback. This receive operation is repeated until the file size is reached.

To deal with the possible situation where the file data come together with the HEADER, we use HeaderRevcValid to control the flow. If HeaderRevcValid is 0, meaning the response is first time received and needs verification of server's response status.

The server's response status, denoted by GF_OK, GF_ERROR, GF_FILE_NOT_FOUND, or GF_INVALID, is stored in the status element of gfcrequest_t *gfr. To convert the server status to a string (OK, ERROR, FILE_NOT_FOUND, or INVALID), the gfr_getstrstatus method is employed, passing (*gfr)->status. Once the status is OK, set HeaderRevcValid to 1 and jump the code to receive and write file data directly.

gfserver:

In the server side (gfserver_main.c), methods like gfserver_create and gfserver_serve are used, passing the opaque pointers gfserver_t and gfcontext_t *ctx. The former handles callback registration, handler functions, port, and connections, while the latter stores ongoing request details.

Invalid requests trigger a response with the header GF_INVALID, and the server proceeds to the next request. Valid requests pass the file path and gfcontext_t pointer to the handlerfunc(), where file data is read, and gfs_sendheader and gfs_send convey request status, size, and content.

In case of errors or file absence, gfs_abort closes the connection via the gfcontext_t pointer.

Socket communication involves reading server response and client request byte-by-byte, with string functions like strstr and strtok used to examine and potentially split the header.

To be more specific:
after accept() successful, there are two scenarios to deal with

1. The HEADER request is sent in full:
   Use strstr() to find end marker "\r\n\r\n" and so on. Send INVALID if anything is not correct. Key algorism would be using strcmp():

```
// if three are equal, should return 0
if (0 == strcmp(scheme, "GETFILE") + strcmp(method, "GET") + strncmp(filepath, "/", 1))
```

2. The HEADER request is not sent in full:
   they come in pieces via recv() like G E T F I L E G E T
   We cannot use strstr() now to find end marker "\r\n\r\n" to determine now because it still could be a valid HEADER. We need to find a way to append the buffer to determine the end marker "\r\n\r\n". Naturally I want to loop another recv() to do so and it works if the data is indeed coming

in.
Problem is that if we are actually in first scenario and there's no more data coming in recv() pipe. The recv() then hangs because client doesn't close and waiting for reply... Server and client waiting for each other. I decide to use select() to determine the pipe status and timeout when client fails to send data or pipe ends prematurely.

Testing:

Tested the server response to invalid request scheme , method and file path. Checked all test cases in the https://github.gatech.edu/cparaz3/6200-tools/. Added some more tests like premature closes, file path limit, anything that that I can think of. Confirmed transmission by printing the total number of bytes sent, which coincided with the number of bytes received on the client's end. Double checked the transferred file content can be successfully viewed and hash checked.

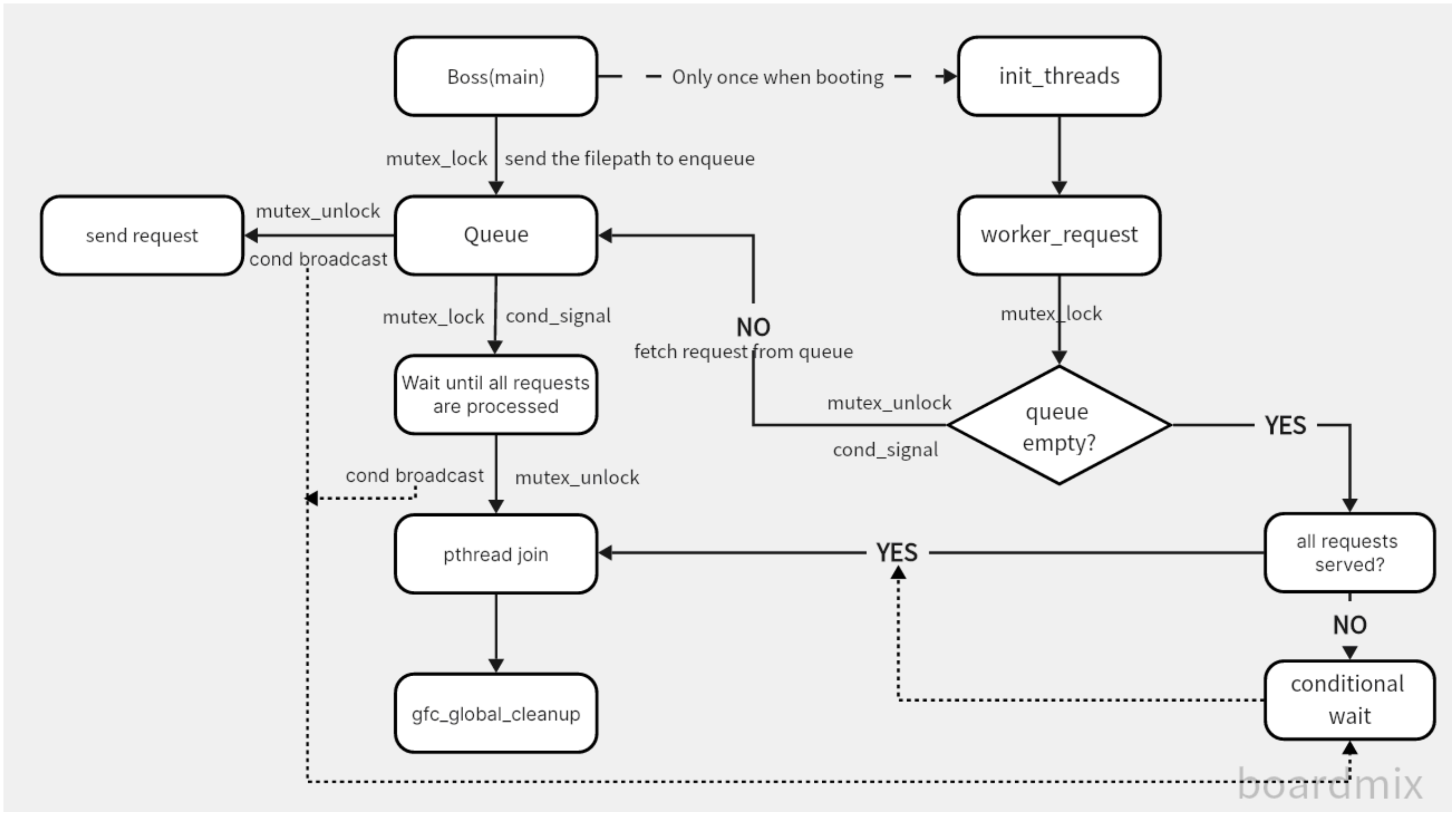
Checked if the client closes the socket after successfully receiving the header and file data, depending on the server-specified file size.

Also, confirmed that the gfc_perform method returns the correct code corresponding to each server status.

Part 2. Multithreaded Getfile

In Part 1, the Getfile server can only handle a single request at a time. To overcome this limitation, we made the Getfile server multi-threaded by implementing our own version of the handler in handler.c and updated gfserver_main.c as needed. This is the actual part where we finally apply and use the lectures content...

Multithreaded gfserver_main.c

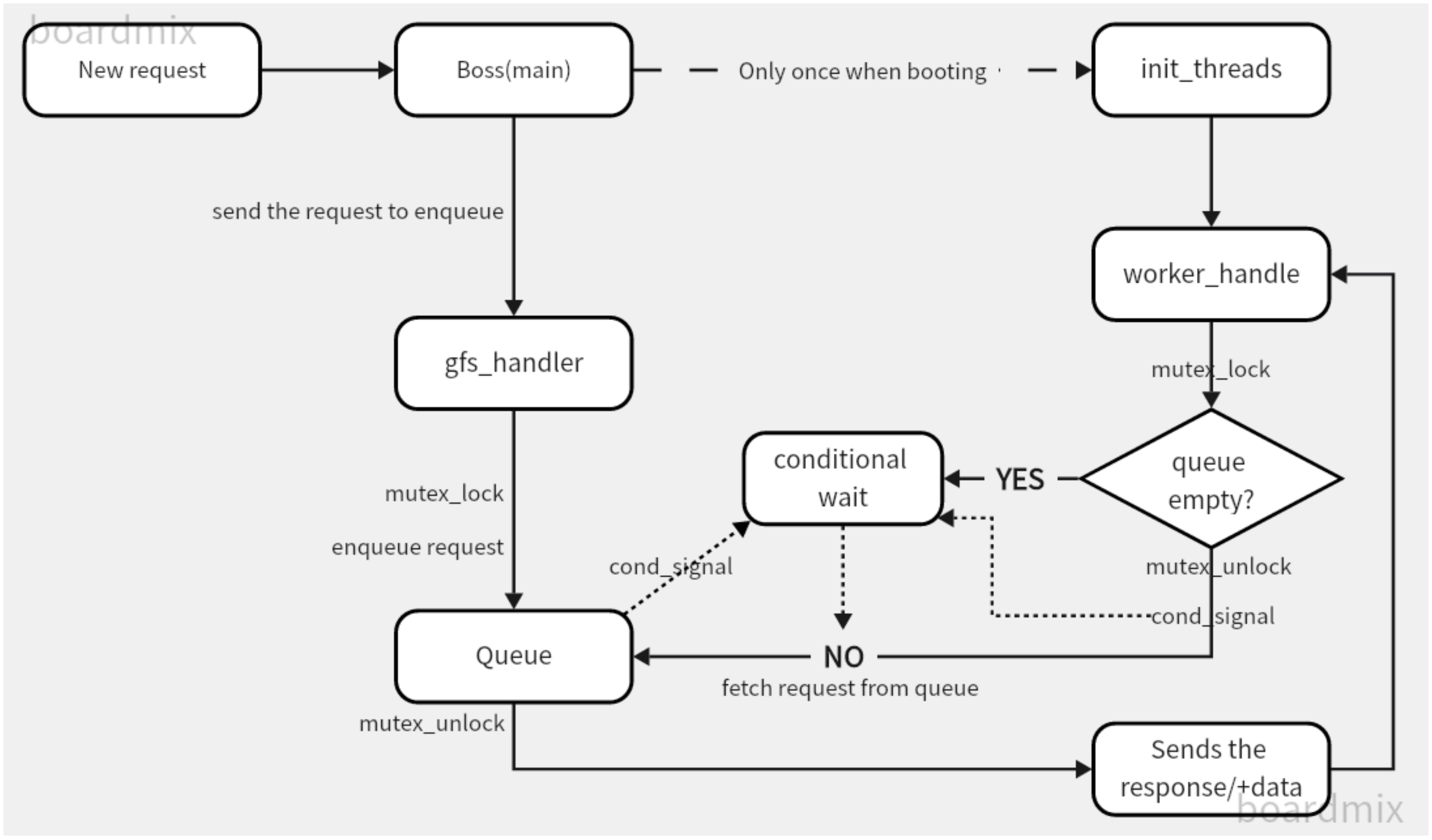


In the Multithreaded Getfile server, the Boss thread initializes a pool of worker threads using the init_threads method, specifying the thread count from the command line. Worker threads wait for requests in a queue using pthread_cond_wait within a pthread_mutex_lock. The Boss thread forwards gfserver requests to the handler callback, which enqueues them in the queue under pthread_mutex_lock. After enqueuing, the mutex is unlocked, and worker threads are signaled with pthread_cond_signal.

The conditional wait breaks when the queue is not empty, the mutex is unlocked, and worker threads process requests, handling file data read and transmission. Upon successfully sending the response header and file contents, worker threads return to serve the next request.

More specifically:
When the wait condition ends (queue not empty), a request is popped. The file path is passed to content_get, returning a file descriptor if found; else, FILE_NOT_FOUND status is sent to the client. Using fstat, the file size is determined and sent to gfs_sendheader with the context. File contents are sent to the client via gfs_send until the file length is reached, followed by freeing the request and context.

Multithreaded gfclient_download:



In the multithreaded client for the Getfile protocol, the boss thread creates a worker thread pool. File paths from the workload are enqueued in the request queue based on the number of requests. Worker threads, upon launch, wait conditionally until the queue is empty. When a request is enqueued under mutex lock, threads are signaled with pthread_cond_signal, and the mutex is unlocked.

After the conditional wait, worker threads dequeue a request, process it, and send it to the gfc_perform method of gfclient. This repeats until all requests are processed and files are received. The boss thread awaits request completion, unlocks the mutex for worker threads to join and terminate. Finally, gfc_global_cleanup() is called to free allocated memory.
The challenge part is to coordinate with the boss thread to effect a clean shutdown.
The boss thread has to wait in a conditional wait until the total requests served and processed are less than the actual number of requests. When both counts match the actual number, the conditional wait mutex is unlocked, and control exits the loop, returning to the main method. Here, using pthread_cond_signal can cause deadlock because it is uncertain to wake all threads. So instead, we have to use pthread_cond_broadcast to make sure the pthread_join function is iteratively called for all threads to join the boss thread and terminate. After worker threads are terminated, gfc_global_cleanup is invoked to free memory used for global resources.

Testing:

Confirmed transmission by printing the total number of bytes sent, which coincided with the number of bytes received on the client's end. Double checked the transferred file content can be successfully viewed and hash checked. Both under single thread and multithreaded scenario.

For the client, when the number of threads equals the number of requests, worker threads processed requests successfully, terminating upon receiving files.

For the server, tested if worker threads synchronously serve requests fetched from the queue, returning to a wait condition to handle upcoming requests.

References

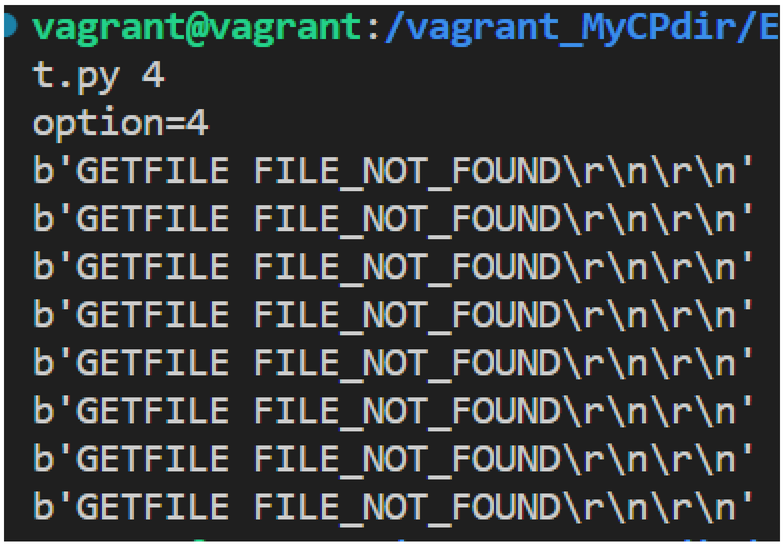
Reused the socket level code part from Echo client / server

- "Beej's Guide to Network Programming." https://beej.us/guide/bgnet
- Example: Accepting connections from both IPv6 and IPv4 clients https://www.ibm.com/docs/en/i/7.5?topic=sscaaiic-example-accepting-connections-from-both-ipv6-ipv4-clients
- zx1986. 2013. "Zx1986/XSinppet." https://github.com/zx1986/xSinppet/tree/master/unix-socket-practice.
- "C Library Function - Memset() - Tutorialspoint." https://www.tutorialspoint.com/c_standard_library/c_function_memset.htm.
- "How to Convert Unsigned Short to String in C?" https://stackoverflow.com/questions/63666912/how-to-convert-unsigned-short-to-string-in-c.
- "C - Socket Programming -- Recv() Is Not Receiving Data Correctly." https://stackoverflow.com/questions/30655002/socket-programming-recv-is-not-receiving-data-correctly.
- "C++ - Sending Image (JPEG) through Socket in C Linux." https://stackoverflow.com/questions/15445207/sending-image-jpeg-through-socket-in-c-linux.
- "Strtok - How to Split HTTP Header in C?" https://stackoverflow.com/questions/22732119/how-to-split-http-header-in-c.
- "C - Differ between Header and Content of Http Server Response (Sockets)." https://stackoverflow.com/questions/16243118/differ-between-header-and-content-of-http-server-response-sockets.
- Referred the below Github file to understand the process control flow https://github.com/xericyang97/GIOS/tree/main/Project1

- “C Cut Char Arry and Save Binary Data from Socket.” https://stackoverflow.com/questions/32908299/c-cut-char-arry-and-save-binary-data-from-socket.
- "GIOS Pr1 high-level code design” https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/
- “Writing to/Reading from File Using Pointers, C.” https://stackoverflow.com/questions/31388934/writing-to-reading-from-file-using-pointers-c.
- “POSIX Threads Programming.” https://hpc-tutorials.llnl.gov/posix/#PassingArguments
- “Multi-Threaded Programming With POSIX Threads.” http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html
- Producer consumer example code shared in the lecture.
- “How Do I Share Variables between Different .c Files?” https://stackoverflow.com/questions/1045501/how-do-i-share-variables-between-different-c-files.
- “Pread(2) - Linux Manual Page.” https://man7.org/linux/man-pages/man2/pread.2.html.

In addition

If I use the object files gfclient.o and gfserver.o from part 2 to test for part 1 with malformed requests. The server returns FILE_NOT_FOUND instead of INVALID for the malformed requests.



```
vagrant@vagrant:/vagrant_MyCPdir/E
t.py 4
option=4
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
b'GETFILE FILE_NOT_FOUND\r\n\r\n'
```

Same logic, If I used my own code from part 1 gfserver to implement part 2 code by generating new gfserver.o file, the server will fail to handle request. It will always return FILE_NOT_FOUND. So I guess this inconsistency is inevitable to make part 2 work. This also means we cannot test part 2 code for the malformed requests. Unfortunately, how to figure out the reason behind and how to fix it is way beyond my ability of C program skill...