Implementation of a GetFile Server

Author: Yuan Yang (yyang998)

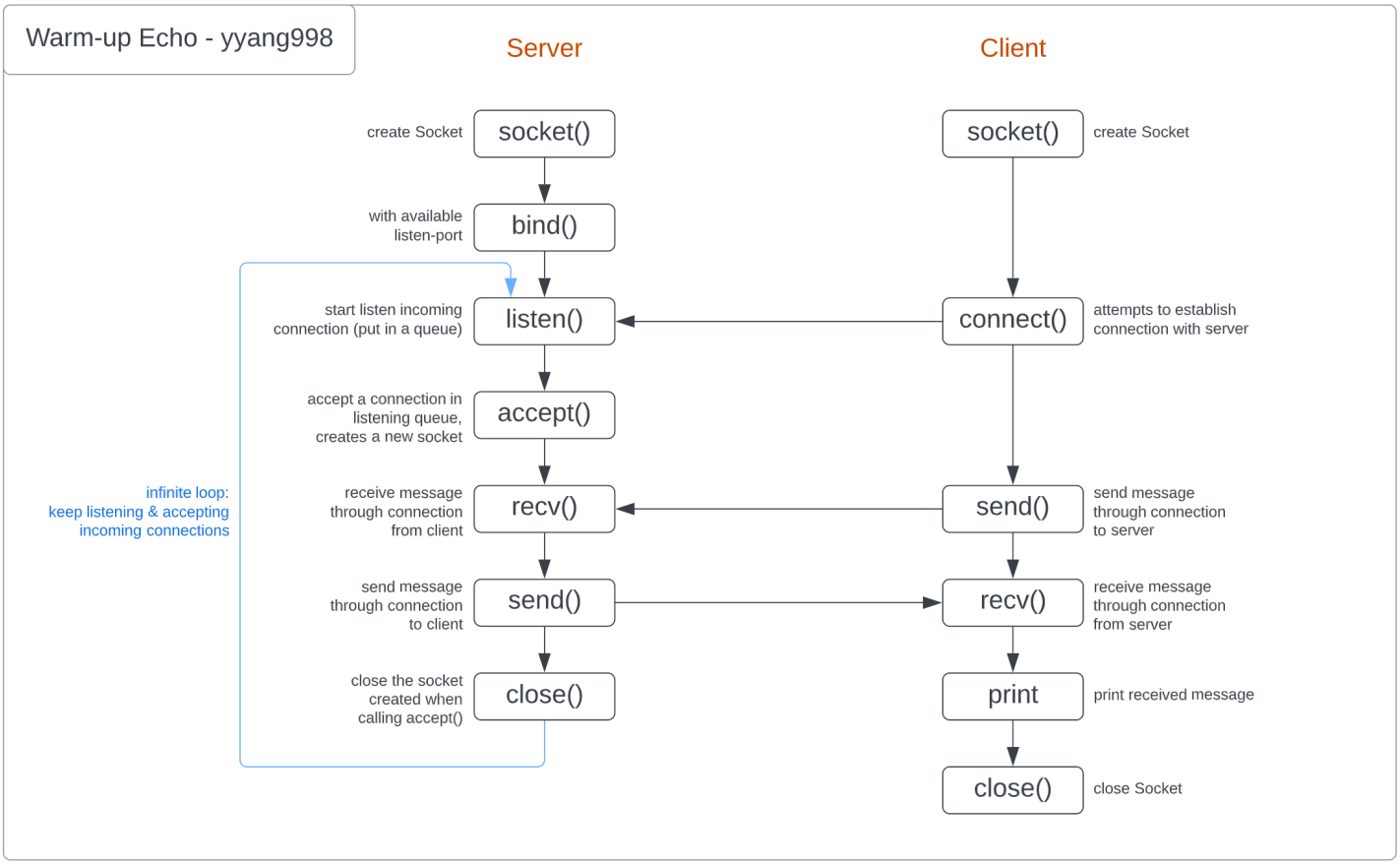
This project is developed under Ubuntu Linux 20.04 environment with Vagrant and VirtualBox.

1. Warm-up: Echo Client-Server

Design

In this part of the project, a simple echo client-server system is implemented. The basic mechanism is that Client side sends a message to Server side, then Server side sends the same received message back to Client. The expected outcome is that the Client side prints the exact same message it sent to Server.

The figure below shows the flow of control for implementing this mechanism using Socket. This diagram is the fundamental for this whole project, and the flow of control in the next three parts are all based on this one.



The detailed flow of control is detailed explained here below. The rest of this report will refer to this section for socket related mechanisms.

Server side

- Flow of Control for the server side:
  i. Creates a node for communication;

ii. Binds the node with a specified listen-port;
iii. Starts listening to incoming connections and puts them in a queue with limited capacity;
iv. Accepts a connection from the queue and creates a new Socket node to handle this connection specifically;
v. After connection established, server can receive message from client;
vi. Server sends the message back to client;
vii. Closes the new socket created in step 4 cause this round of connection is done.

- Key points for server side:
  - Server will never stop unless it is forced to. After step 7, it will go back to step 3 and repeat the whole process.
  - Several sockets(nodes) will be created, but only the socket created in step 1 will never be closed as long as the server is working. Other nodes are all used for the communication with different clients and are closed after each communication is done.
  - For the queue established for storing incoming connections in step 3, if it is full, the server will reject the incoming connections and the client side will get error.

Client side

- Flow of Control for the client side:
  i. Creates a node for communication;
  ii. Connects with the server that is bound on the port specified;
  iii. Sends message to server;
  iv. Receives the message back from server;
  v. Prints the message;
  vi. Closes the socket created in step 1.

Choices and Code Implementations

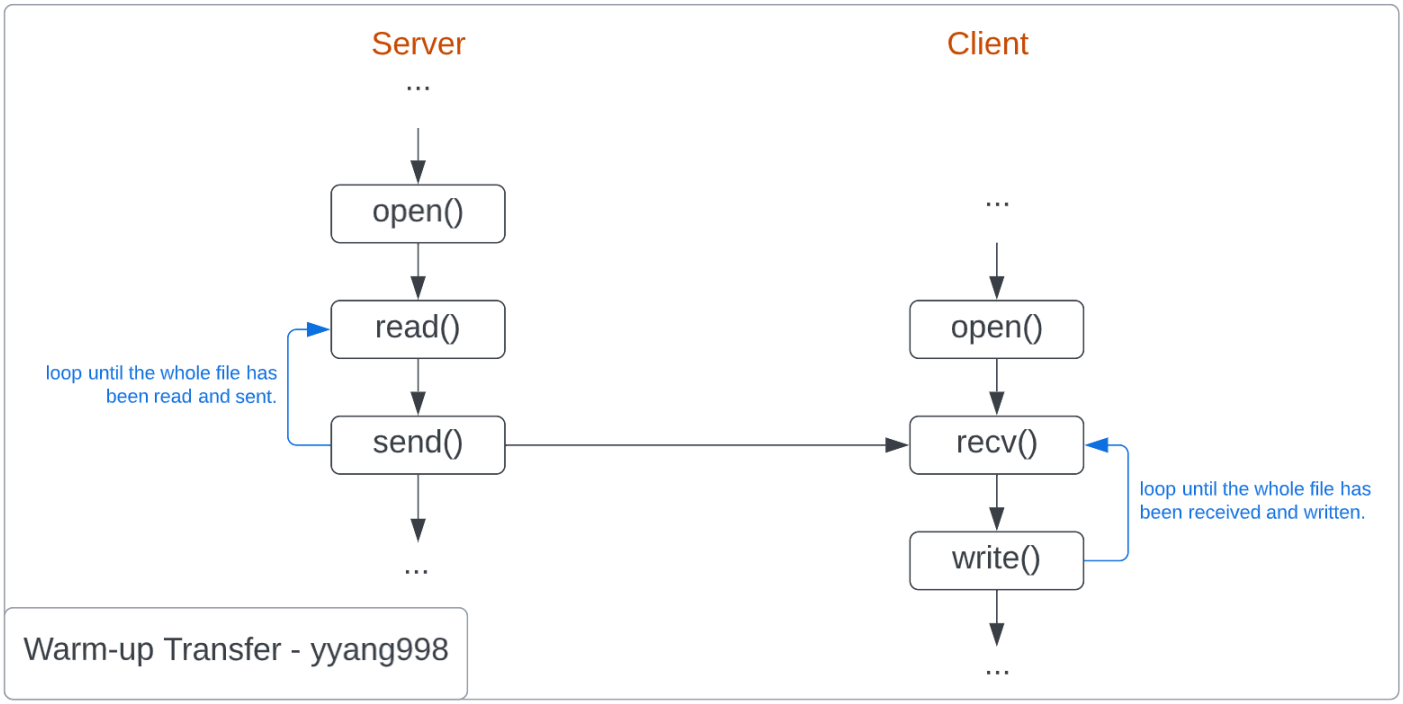
- Why choose system calls recv()/send() over read()/write(): recv() and send() are designed for socket related operations, and accepts different flags. Moreover, the errors these two operations can return are more specific for sockets and are helpful for solving any bugs caused be socket file descriptor.
- How to let server supports both IPv4 and IPv6: when selecting the socket addresses using `struct addrinfo hints`, the `ai_family` options should be set to `AF_UNSPEC` to support both IPv4 and IPv6.
- How to increase the successful rate for creating and binding socket: after calling `getaddrinfo()`, the returned result is a linked list of addresses. And we loop the whole linked list until we can find an address that can be used to create socket and bind to the port successfully.

- How to resolve "Address already in use" error when calling bind(): set the socket option(`setsockopt()`) of created socket to be allowing port re-using.

2. Warm-up: Transferring a File

Design

This part is very similar to the echo part. All the basic socket related operations are the same as the echo part. Instead of echoing a message back and forth, server reads a file and sends the content of the file to client, while client receives the file content from server and writes to a file on client side.



Server side

1. Once the connection with client established, open() the file specified in the inputs;
2. read() file content to a buffer;
3. send() the content in buffer to client.
   The file might be large and could not be read into the buffer as a whole, so we need to keep reading and sending using a loop until all the content are sent.

Client side

1. open() file specified in the inputs. Creates a new file if the file path not exists;
2. recv() content from server into a buffer;
3. write() the received content to the file.
   Similar to the server, we need to keep receiving and writing using a loop until all the content are written.

Choices and Code Implementations

- When opening a file from the server side, it is opened only with read permission. Because we only want to read this file instead of modifying it.

  ```
  open(filename, S_IRUSR);
  ```

- When opening a file from the client side, it is opened with write permission. Also, the option needs to be specified to allow creating a file when not exists.

  ```
  open(filename, O_CREAT | O_WRONLY, S_IWUSR);
  ```

- For a large file, it is likely that read() and send() will only send a chunk of file at once. So it is required to loop read() and send() until the whole file has been read. The criteria for judging if the whole file has been read and sent: read() returns 0, means read/write pointer is pointing at the end of the file.

Tests

In order to ensure my code can handle the transfer of a very large file, I created a file that contains 100,000 lines of pseudo-content using command:

```
tr -dc "A-Za-z 0-9" < /dev/urandom | fold -w100|head -n 100000 > bigfile.txt".
```

Then use command below to test:

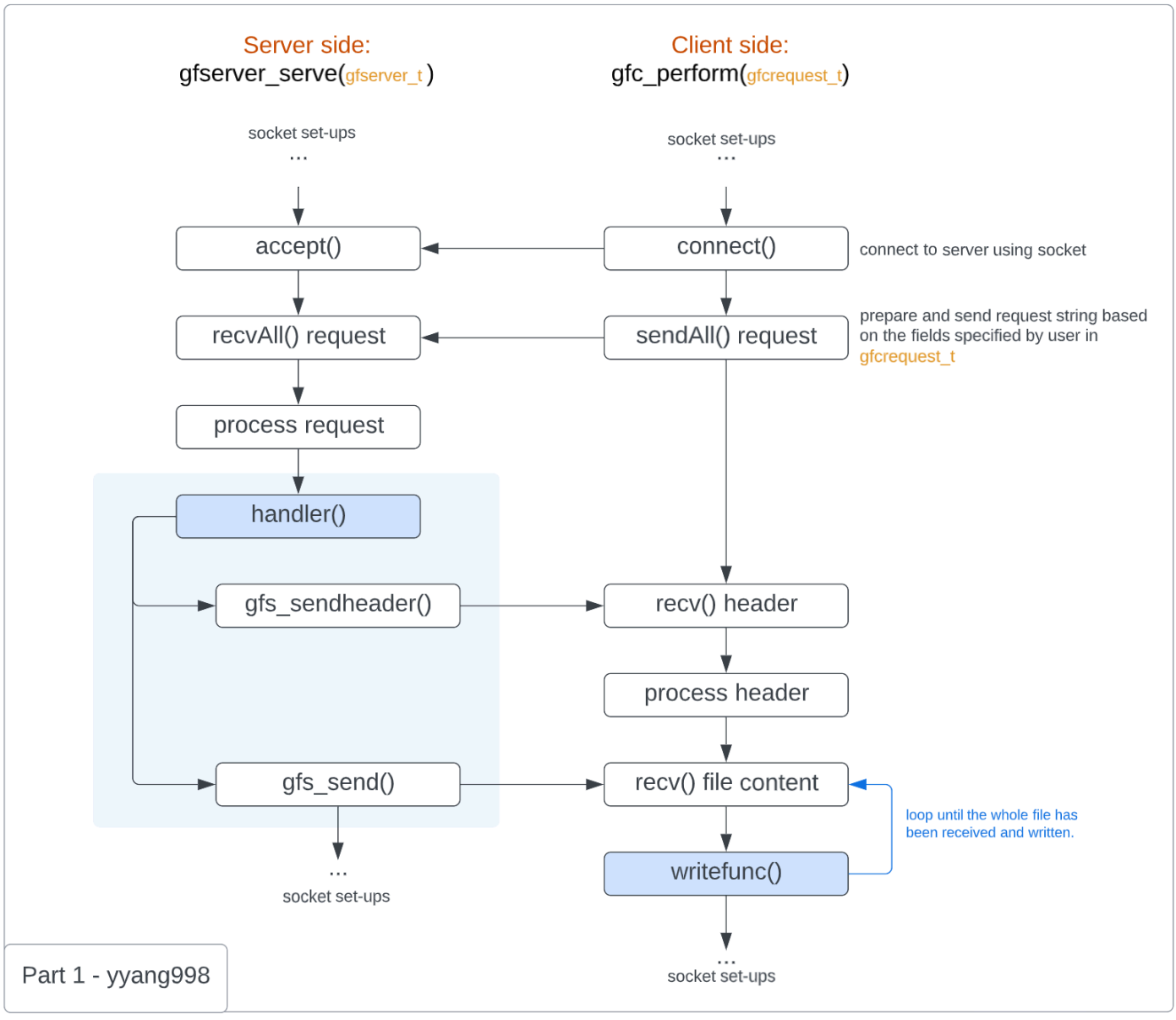
```
./transferserver -f bigfile.txt
```

After the transfer, the output file contains all the 100,000 lines. The code tested to be reliable.

3. Part 1 - Implementing the GetFile Protocol

Design

This part of the project implements a library to support requesting and sending file between two nodes. The basic connection mechanism is still implemented using socket, and the communication between server and client is still sending files. The difference is that server will find the file based on the request sent by client, and then send the file to the client. Moreover, several APIs are implemented on both server and clients side for usage.



Server side : gfserver.c; Server will never stop unless it is forced to.

1. Receives the requests sent by client: make sure the whole request is received;

2. Parses the request using `sscanf()` : gets and validates Schema, Method, and File Path specified in request;
3. Calls callback function `handler()` to handle the request; handler() further processes request and then call `gfs_sendheader()` and `gfs_send()` functions to send header and file content to client.

Key points for Server side: gfclient.c

- Callback function handler() opens and reads file requested n the server side, gets the file contents, and then calls gfs_sendheader() and gfs_send() implemented in gfserver.c with the known file contents. We do not need to get the status, file_len, and content of file by ourselves, the handler will provide the data.
- When the request received in step 2 is not valid, instead of calling handler(), server needs to send the response header by itself to let client know that status is invalid.

Client side

1. Prepares and sends request message to server: make sure the whole request is sent;
2. Receives response header sent from server;
3. Parses the request using `sscanf()` : gets and validates Schema, Status, File Length and possibly some file content attached at the tail.
4. Receives file content from server and call `writefunc()` to write the content into a file location specified by user. Repeat this step until the whole file has been received (criteria: received bytes == File Length got in step 3).

Key points for Client side:

- In step 3, the received response header might contain some file content. In this process, in addition to getting Status and File Length, we need to watch out for the file content attached. If some file content is detected, `writefunc()` needs to be called to write the content to the file.

Choices and Code Implementations

- One of the most challenging things in this part of the project is to prepare and parse the messages/info sent and received between server and client. Generally, `sscanf()` is used to process all the messages with some format specified in project spec.
- Both of server and client need to receive header from each other. The headers are terminated with "\r\n\r\n". When receiving the header, we want to make sure we can receive and copy the whole header, and then we can use `sscanf()` to parse it. I implemented a function called `recvallHeader()` in `gf-student.h` . This function keeps receiving info from the socket

connection until encounters "\r\n\r\n". Before encountering this marker, after receiving a chunk/part of the header into a buffer, it uses `memcpy()` to copy the info from buffer to the final string that created to store the whole header.

- On the client side, in order to check if response header contains file content, the strategy I used is:
  i. get the position `pos` of "\r\n\r\n" in the received header;

```
char *pointer = strstr(responseHeader, "\r\n\r\n");
int pos = pointer - responseHeader;
```

  ii. check if `pos + 4` is the end of the header.

```
pos + 4 <= recv_size //indicates "\r\n\r\n" is not the end of the header
```

  iii. call writefunc() to write the file content appended at the end to the file. Also need to record the number of bytes that has been written into file.
- On the client side, when receiving and writing the file content, we need to keep track of how many bytes have already been received and written. Then compare to the actual file length to provide an indication on whether the operation successes or fails.
- On the client side, increases the buffer size for storing the response file content to 64000, otherwise takes a long time for receiving large file.

Tests

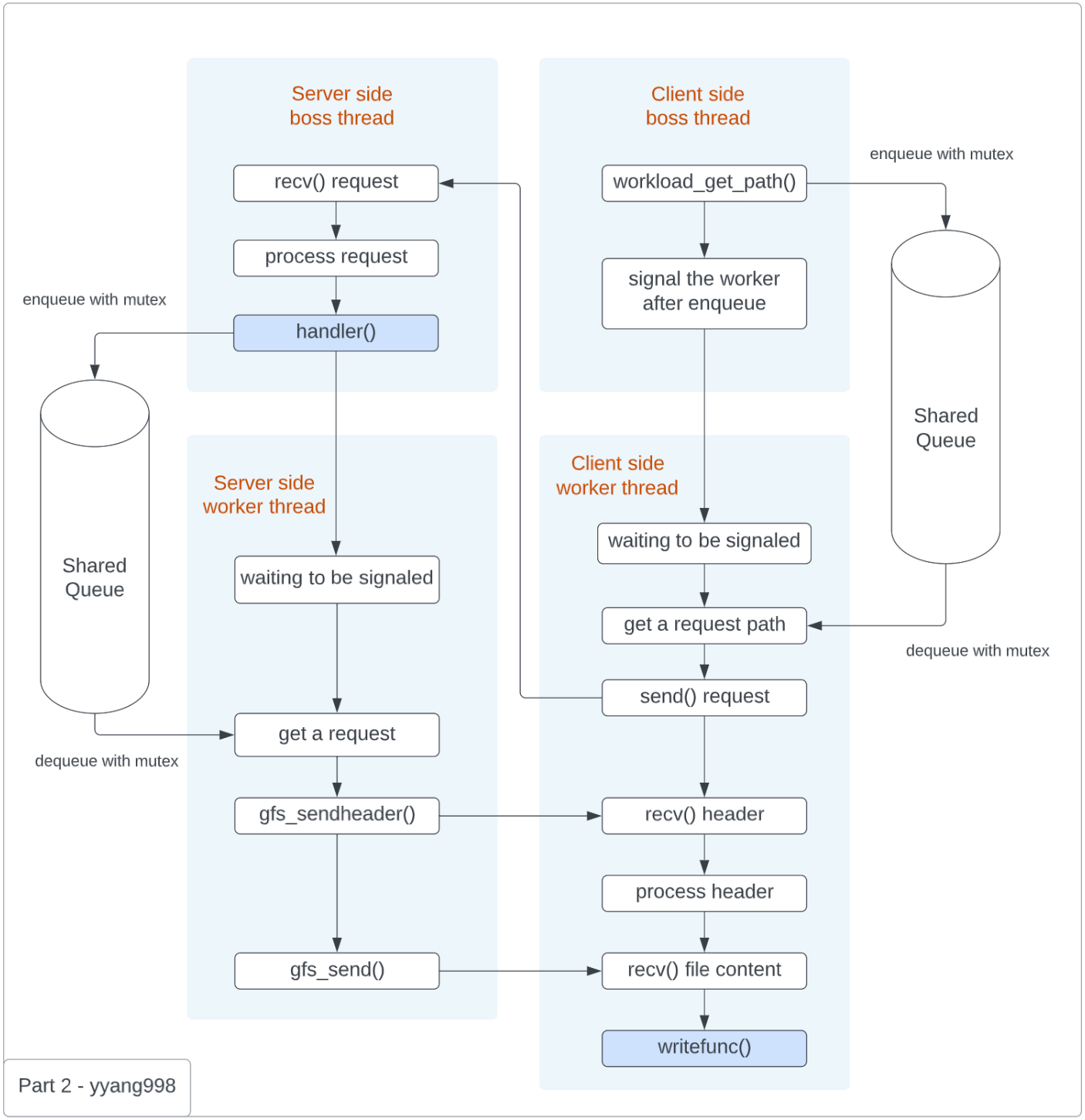
- Used binaries posted by instructors and classmates in Piazza (used `chmod` command to change permission of the binaries downloaded online).
- Modified workload.txt and content.txt to test server and client for other scenarios other than an OK status: provide a path that not exists which can lead to a FILE_NOT_FOUND status.
- For client:
  - Modified gfclient_download.c: set path to be a path that not exists `gfc_set_path(&gfr, path_not_exists);` , leading to FILE_NOT_FOUND status.
  - Interrupt the client while transferring a request, print out the status provided by `handler()` in `gfs_sendheader` : the status should be INVALID. Also print out the header that will be sent back to client: the file length should be omitted.
- For server:
  - force stop the server while transmitting file content: error message should be printed.
- Major bug found during debugging phase:
  - client hung error: didn't stop receiving when everything's actually received on the client side. The server will not close the socket online, but not sending anything in. While client is still

waiting for the content to come in.

4. Part 2 - Implementing a Multithreaded Getfile Server

Design

In this part of the project, for both server and client sides, multithreaded processing functionalities needs to be implemented using PThread Programming. The user can specify how many threads they want on the client and server sides respectively, and server and client follow boss-worker thread pattern to handle the requests.
The basic code structure of this work referenced **High-level code design of multithreaded GetFile server and client** provided in the project spec.



Server side : gfserver_main.c

- Boss thread: keep accepting connection, receiving requests, enqueue requests to shared queue
  i. Initialize a queue using `steque_init()` to store the requests accepted by current boss thread; This queue is protected by a `mutex`, and will be shared with all the worker threads created in next step.
  ii. Creates and starts worker threads using `pthread_create()`;

iii. Call `gfserver_serve()` and this function will call `gfs_handler()`. In `gfserver_serve()`, the process here will be looping forever until force quit:
    - accept a connection with client, receive and process a request;
    - lock the mutex coupled with the shared request queue;
    - enqueue the received request into the queue;
    - broadcast to awake worker threads to work on the requests in the queue;
    - unlock the mutex coupled with the shared request queue;

- Worker threads: stand by in a thread array and process the requests in shared queue when signaled
    i. lock the mutex coupled with the shared request queue;
    ii. after being broadcasted, get a request from shared queue;
    iii. process request and prepare header: get status, file length;
    iv. send header via `gfs_sendheader()`;
    v. read the file content and send via `gfs_send()`.

Client side : gfclient_download.c

- Boss thread: in main() function
    i. Initialize a queue using `steque_init()` to store the paths that client wants to request; This queue is protected by a `mutex`, and will be shared with all the worker threads created in next step.
    ii. Creates and starts worker threads using `pthread_create()`;
    iii. Load request paths from workload file and enqueue all the paths to shared queue (until the number of requests enqueued reached the number of requests specified by user)
        - load a request path using `workload_get_path()`;
        - lock the mutex coupled with the shared request queue;
        - enqueue the request path into the queue;
        - broadcast to awake worker threads to work on the requests in the queue;
        - unlock the mutex coupled with the shared request queue;
    iv. When all the requests are processed and all the worker threads are done, join worker threads. (Detailed mechanism will be explained in Choices and Code Implementations)
- Worker threads: stand by in a thread array
    i. lock the mutex coupled with the shared request queue;
    ii. after being broadcasted, get a request from shared queue;
    iii. if the request is a poison pill, put the poison pill back to the shared queue, broadcast other stand-by worker, unlock mutex, then terminate this thread; if not, continue to next step.
    iv. prepare request and send to server;
    v. receive header and file content from server;
    vi. write content to file.

Choices and Code Implementations

- **Client needs to know when to exit**: On the client side, after all the requests are processed, all the worker threads need to be exited and joined. And the client side will need to be terminated automatically. The strategy I used here for letting workers know when the requests are all fully processed is to push a poison pill into the shared request queue. The process are listed:
  i. Boss thread enqueues a poison pill into the queue at the end after all the valid requests are enqueued. So the poison pill will the last node being popped from the queue, and encountering poison pill indicates all the requests are completed or being completed.
  ii. When a worker thread dequeued a poison pill, this worker knows there is nothing left for it to process. So in this worker thread, the poison pill need to be pushed back (other active workers might still need this), other workers need to be signaled, the shared queue need to be unlocked. After all these works are done, the current thread can exit.

```
if (strcmp(req_path, "poison") == 0) {
  printf("popped poison from queue, exiting current thread. \n");
  steque_enqueue(request_queue, "poison"); //put poison back (we only have one pois
  pthread_cond_broadcast(&worker_cons);    //wake other idle worker
  pthread_mutex_unlock(&m);                //release the lock for others
  return 0;
}
```

- **Adapted Memory Ownership model** introduced in piazza @78 for server side.
- The basic model for using mutex in multithreaded programming when enqueue:

```
pthread_mutex_lock(&m);
steque_enqueue(request_queue, cur_node);
pthread_cond_broadcast(&worker_cons);
pthread_mutex_unlock(&m);
```

- The basic model for using mutex in multithreaded programming when dequeue:

```
pthread_mutex_lock(&m);
while (steque_isempty(request_queue)) {
  pthread_cond_wait(&worker_cons, &m);
}
cur_request = steque_pop(request_queue);
pthread_mutex_unlock(&m);
```

- On the server side, worker thread needs to open and read requested files. The operations for opening and reading must be thread-safe.

- `content_get()` is used to get the file descriptor.
- `fstat()` is used to get the length of the file:

```
struct stat stats;
fstat(file_fd, &stats);
size_t file_len = stats.st_size;
```

- `pread()` is used to read file;

Tests

- Used binaries posted by instructors and classmates in Piazza (used `chmod` command to change permission of the binaries downloaded online).

References

- Class Lectures and Recommanded Papers
- Piazza discussions
- Beej's Guide to Network Programming
- Practical TCP/IP Sockets in C: http://cs.baylor.edu/~donahoo/practical/CSockets/
- https://www.linuxhowtos.org/C_C++/socket.htm
- https://man7.org/index.html
- https://stackoverflow.com/questions/70853190/recv-reading-several-times-when-send-sends-only-once-in-tcp-c-sockets-can-i
- https://stackoverflow.com/questions/11952898/c-send-and-receive-file
- Test method used in Warm-up Transfer part: https://stackoverflow.com/a/64585623
- High-level code design of multi-threaded GetFile server and client: https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit
- https://hpc-tutorials.llnl.gov/posix/