Part 1

Design

For Part 1, we mainly had to implement `handle_with_curl.c`, a function that takes a file path and then use the `curl` library to attempt to retrieve the file from the web. The server in `webproxy.c` must then pass `handle_with_curl` as a callback function to its workers.

The basic design of `handle_with_curl` is based off the recommendations in the `curl` library documentation; I invoke `curl_easy_init` to get an "easy handle", and then manipulate the handle before performing the request. From there, I register the request url and a write callback, and call `curl_easy_perform`.

Within the `write_callback`, the program first checks if the header is sent, and sends the header if not. It uses `curl_easy_getinfo` to get the content length so it can send the header before actually processing the file data. Once the header is marked as sent, `write_callback` then process the file and sends data to the client using `gfs_send`.

Decisions and Tradeoffs

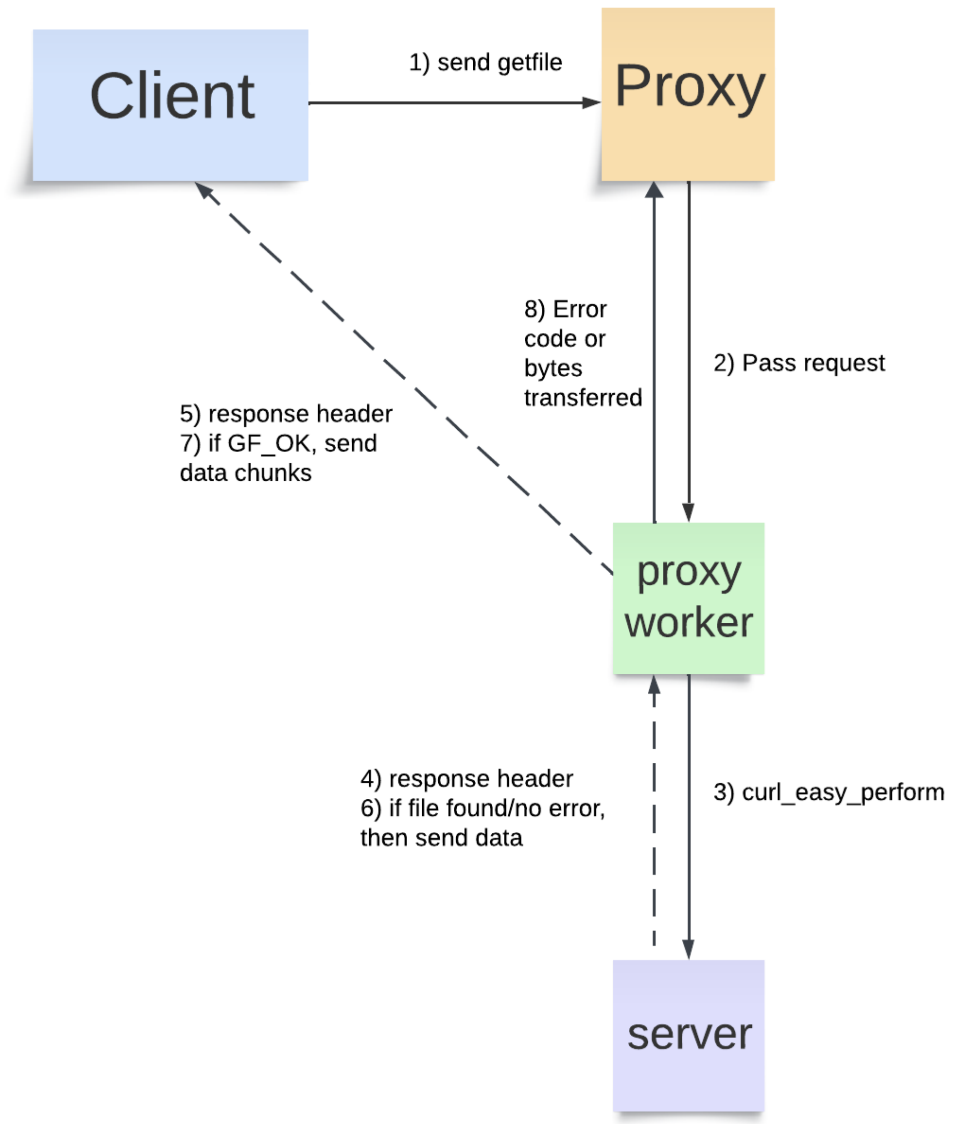
The biggest design decision I had to make was deciding how to ensure that the header is sent first before any data is sent. I had a few options.

One possibility is to perform two separate `easy_perform` calls: one just to get the header and send it to the client, and then a second one for the data. The pros of this solution is that you don't have to worry about accidentally sending data chunks to the client before the header is sent. The cons is that API calls are computationally expensive, and doubling our calls is very inefficient, especially if we may need to handle a large number of requests. So, while this would technically work, I wanted to avoid it.

The second possibility was to create a dynamically-sized buffer that I would store all the file data in. Only after the buffer is completely full would I send the header, and then send the data afterwards. The benefit of this is it doesn't require a second `easy_perform` call just for the header, and also, it keeps the data transfer simple since it's all sent at once. However, the problem with this is with very large files, the buffer can become extremely difficult to manage. Frequent reallocs are actually computationally expensive, and may actually fail if a large enough block of contiguous memory canont be found.

Thus, I landed on my current implementation. I realized that `write_callback` is called repeatedly until the transfer is complete, so I actually decided to make it in charge of calling my `send_header` function. I added a `header_sent` flag in the `header` data, so that this method would send the header if not yet sent, and only after the flag is true will it begin transferring the data. This not only solves the problem of making sure data is not sent until the header is sent, but it also avoids making multiple `easy_perform` calls and having to use a dynamically-sized buffer.

Flow of Control Graph



The client sends the getfile request to the webproxy, which manages its workers. The workers pick up the request and invoke `handle_with_curl` on the request, which involves sending requests with the `curl` library to the specified path. The server responds to the worker either with a response header; if the file is not found or there is an error, the worker sends an appropriate header to the client and then exits, returning an error code to the proxy.

Otherwise, if the file is found, the proxy worker sends a GF_OK header ot the client. The server then sends the worker chunks of data, which the worker passes to the client one chunk at a time. After all the data is done sending, the worker returns the nubmer of bytes transferred to the proxy server.

Testing

To test my Part 1 implementation, I ran the `webproxy` and sent requests using the provided implementation of `gfclient_download`. I used print statements throughout to see the number of bytes being sent, the header, and the response codes.

Once the provided implementation was worknig as expected, I created a manual modification of `gfclient_download` from PR1 that would be compatible with this project. I added a number of edge cases, such as non-existent files to test `FILE_NOT_FOUND` handling, and also tested sending multiple requests at once to make sure `handle_with_curl` would work in a multi-threaded program.

Part 2

Design Designs and Tradeoffs

The overall design of part two involves two servers: the proxy server and the cache server, both of which are multi-threaded.

Proxy Server (Data Channel)

For my design, I made the proxy server in charge of the "data channel"; it would initialize a `stequeu` called `segment_queue`, then intialize shared memory segments and enqueue them into the `segment_queue` (as well as a second `stequeue` called `all_segments`, which is used for cleanup).

I decided that a separate queue would be the best way to manage the memory segments. An alternative would be to map a segment to a worker; as each worker only works on one request at a time, it would theoretically work. However, this has the potential to be very inefficient if the number of segments and number of workers do not match. If there are fewer segments than workers, we'll have workers that can't work at all. If there are fewer workers than segments, we'll have unused segments intialized and just sitting around.

I also used a `stequeue` instead of an IPC like `mq` here, since this queue is only used within the proxy process (shared between the proxy boss and the proxy worker threads). Using IPC here would be overkill and introduce unnecessary overhead. On the downside, since `stequeue` is not threadsafe, I had to use a `mutex` and a `condition variable` to control popping from the queue.

Cache Server (Command Channel)

I had the cache be in charge of the command channel, meaning it would initiate an `mq`, which proxy workers can add requests to. The cache would also initiate a pool of worker threads, which can pick up requests from the `mq`.

I chose to use a POSIX message queue here because an `mq` is inherently threadsafe! No mutexes or condition variables necessary. Moreover, it is also suitable for IPC, so the proxy workers can also queue requests into it for the cache_workers to pick up.

I chose to keep the cache server relatively simple. It doesn't do much other than initializing a mesasge queue and the worker threads. We could have the cache server help assign memory segments to workers, but it would be inefficent and would open up the possibility of many workers being blocked by one boss thread. Instead, I decided to have as much autonomy to the proxy and cache workers as possible for more concurrency and lower chance of blockage.

Proxy Worker (handle_with_cache) and Cache Worker

Another important decision was what to actually include in the `shared_memory_segment` struct versus the message requests. In my design, I put almost all the information pertient to the request within the shared memory segment: the segment name, segment size, a read and write semaphore, bytes written to the segment, file size of the requested file, the requested file path, the transfer status, and a buffer for the data.

```
typedef struct {
    char segment_name[256];
    size_t segsize;
    size_t bytes_written;
    size_t file_size;
    char request_file[256];
    int transfer_status;
    sem_t read_sem;
    sem_t write_sem;
    char data[];
} shared_memory_segment;
```

The `cache_request_msg` on the other hand is very barebones and simple: it just includes the segment name and the full segment size (I say "full" here because this is the size of the entire `shared_memory_segment` plus the `segsize` which is the size of the buffer). This is all it needs to be able to access the segment, and thus, all the other information!

```
typedef struct {
    char segment_name[256];
    size_t full_segsize;
} cache_request_msg;
```

The reason I chose to balance it this way is because it's a very time-efficient implementation. The important data is stored directly in the memory segment which is already shared between the two processes, so no copying + transferring is necessary.

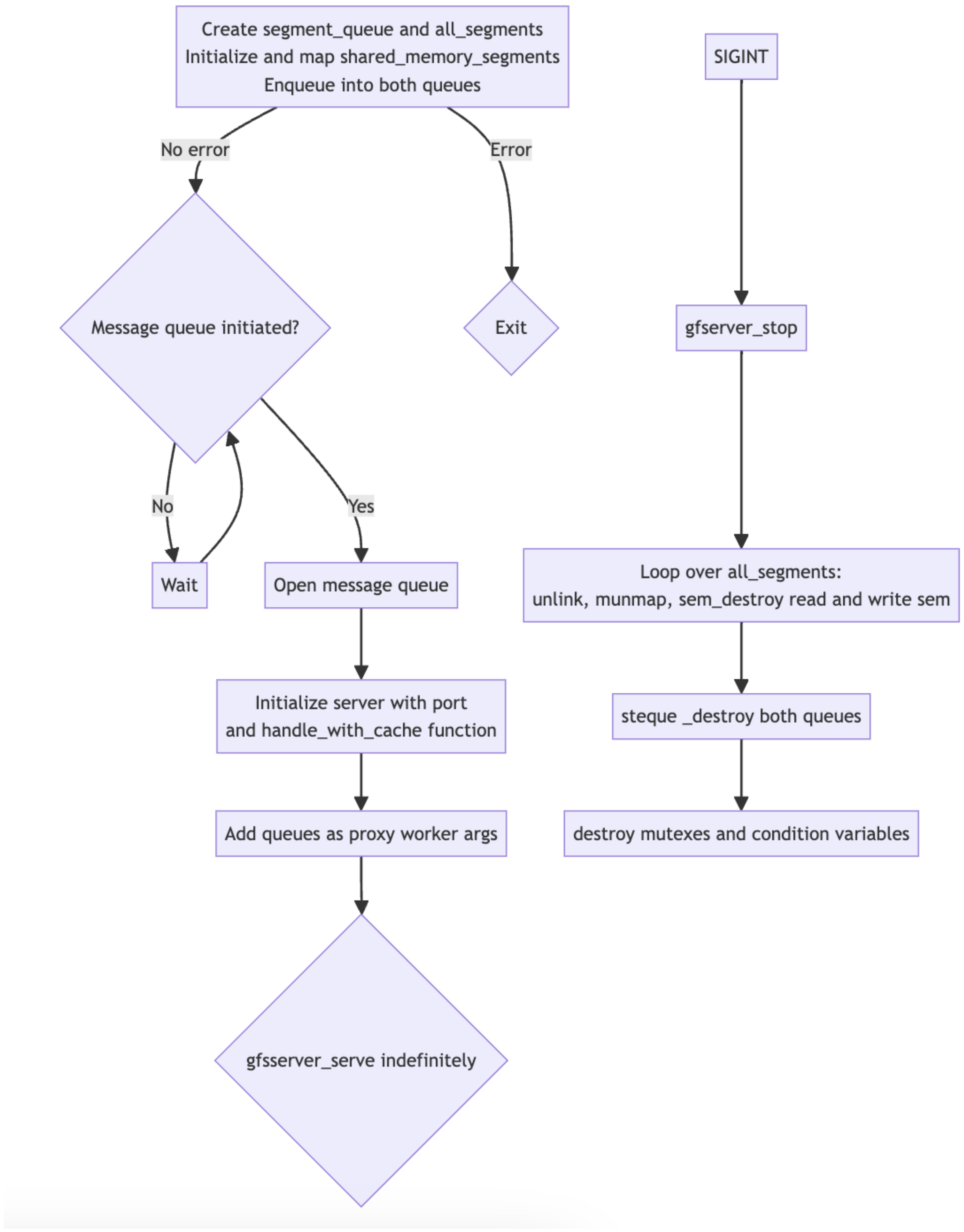
As the request is being processed, the proxy worker and the cache worker have to continuously pass information to one another (such as the status of the transfer, the number of bytes written so far, etc.), so passing messages back and forth would be inefficient, not to mention very complicated. It would likely have to involve two queues to transmit messages in two directions. By using IPC, we keep the implementation very simple, and the two processes can quickly access the important data about the request at any point.

On the downside, using shared memory requires more synchronization and cleanup afterwards; thus, we manage this with two semaphores & have to keep a `all_segments` queue inside the proxy boss so the boss can clean up all the segments.

Finally, I chose to use semaphores over mutexes for synchronizing over the shared memory segment. While semaphores can be complex to manage and prone to deadlocks, the POSIX api provides many useful functions for managing them such as `sem_open`, `sem_wait`, and `sem_post`. Moreover, mutexes are generally only used within a multi-threaded process, and not used for IPC between multiple processes.

Flow of Control Graphs

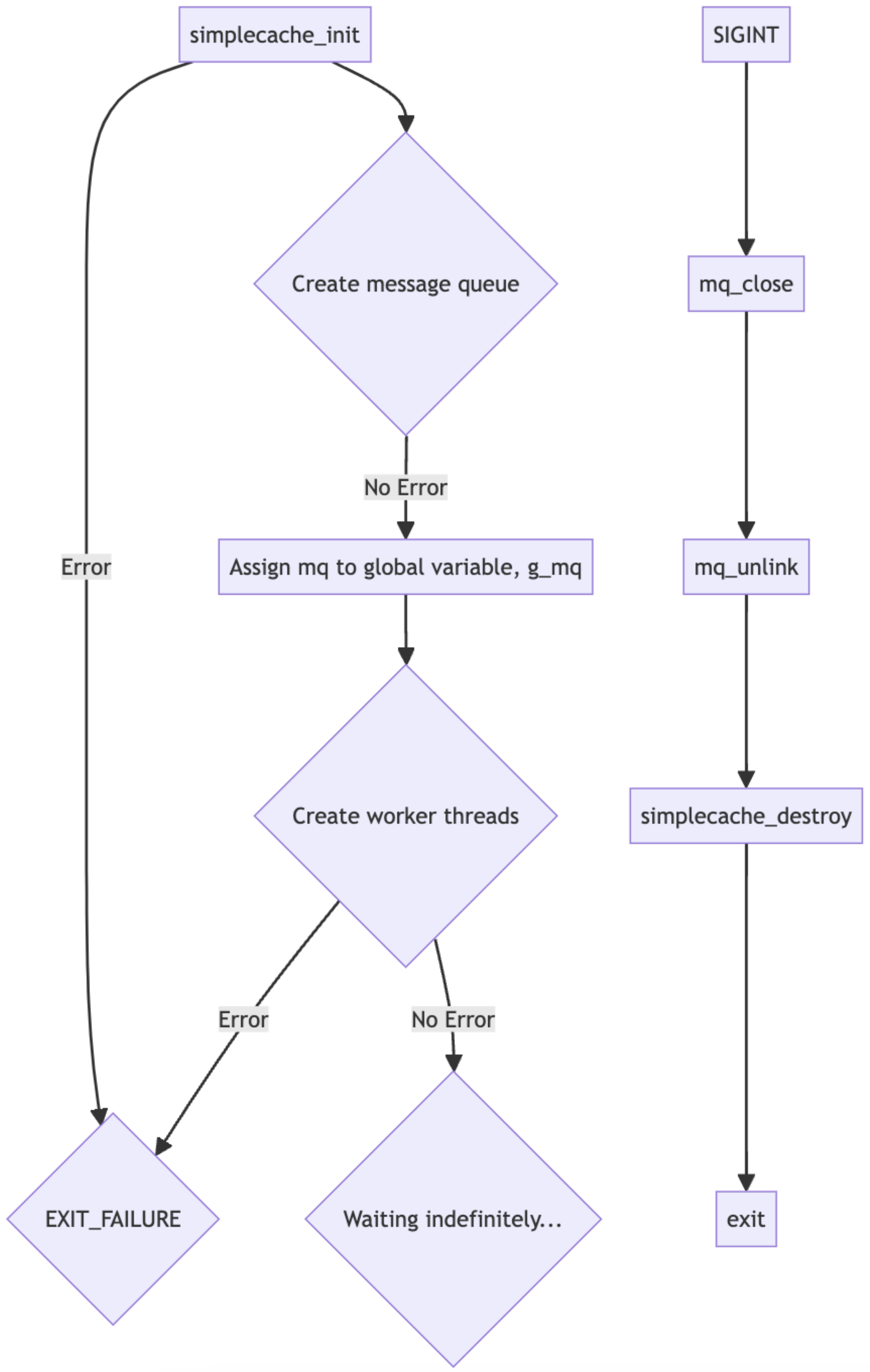
Proxy Server


Create segment_queue and all_segments
Initialize and map shared_memory_segments
Enqueue into both queues
No error
Error
Message queue initiated?
Exit
No
Yes
Wait
Open message queue
Initialize server with port and handle_with_cache function
Add queues as proxy worker args
gfsserver_serve indefinitely
SIGINT
gfserver_stop
Loop over all_segments:
unlink, munmap, sem_destroy read and write sem
steque _destroy both queues
destroy mutexes and condition variables

The flow of control for the proxy is quite self-explanatory from the diagram. It creates the two segment queues, initializes and enqueues each segment, then waits for the message queue to be initiated. Once initiated, it opens the message queue, initializes the server (and registers the `handle_with_cache` function with its workers), and passes the segment queue and message queue to its workers. It then serves indefinitely, waiting for requests.

The SIGINT flow involves stopping the server, looping over the `all_segments` queue to clean up every segment, destroying queues, and destroying mutexes and condition variables.

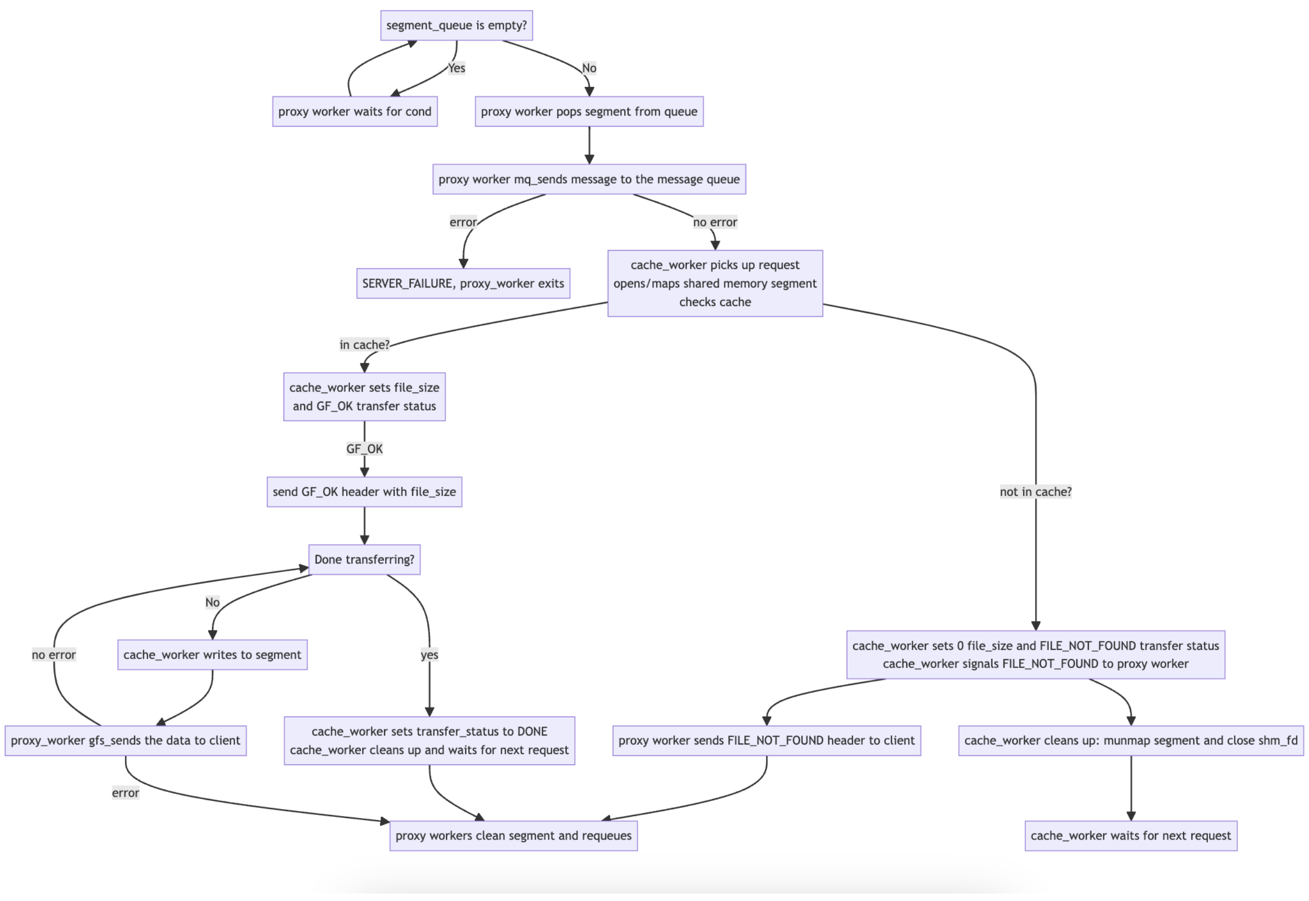
Cache Server



The flow for the cache boss is very simple. It initiatse a cache with `simplecache_init`, creates a message queue, and assigns it to a global variable (`g_mq`) for cleanup purposes. It creates a pool of worker threads then waits indefinitely.

Upon `SIGINT`, it closes the message queue, unlinks it, destroys the cache, and exits.

Request Flow


segment_queue is empty?
Yes
No
proxy worker waits for cond
proxy worker pops segment from queue
proxy worker mq_sends message to the message queue
error
no error
SERVER_FAILURE, proxy_worker exits
cache_worker picks up request
opens/maps shared memory segment
checks cache
in cache?
not in cache?
cache_worker sets file_size
and GF_OK transfer status
GF_OK
send GF_OK header with file_size
Done transferring?
No
yes
no error
cache_worker writes to segment
proxy_worker gfs_sends the data to client
cache_worker sets transfer_status to DONE
cache_worker cleans up and waits for next request
error
proxy workers clean segment and requeues
cache_worker sets 0 file_size and FILE_NOT_FOUND transfer status
cache_worker signals FILE_NOT_FOUND to proxy worker
proxy worker sends FILE_NOT_FOUND header to client
cache_worker cleans up: munmap segment and close shm_fd
cache_worker waits for next request

The most complicated of the flows is the flow between the proxy worker who picks up a segment, enqueues a request in the message queue, and its interactions with the cache worker who picks up the message.

The diagram shows the details the best, but at a high level, when the proxy worker picks up a request from the proxy boss, it pops a memory segment from the queue, then sends a message to the message queue. It waits for a cache worker to pick up the request.

A cache worker pops the request from the message queue, which includes the segment name and size, and opens/maps the segment. It then checks the segment for the request information.

The cache worker checks the cache. If there is no cache hit, it signals `FILE_NOT_FOUND` to the proxy worker and exits. The proxy worker sends a `FILE_NOT_FOUND` header to the client, then cleans the segment and requeues it.

If there is a cache hit, the cache worker will set the file size in the segment, and also set the transfer status to `GF_OK`.The proxy worker will send a corresponding ehader ot the client.

Then, a loop begins. The cache worker writes to the memory segment (specifically the data buffer), the proxy worker sends the data to the client, and so on and so forth, until the file is done transferring.

Once done, the cache worker cleans up and waits for the next request, and the proxy worker cleans the segment and requeues it for use in the next request.

Testing

To test, I first initiated my proxy server and cache server in different orders to make sure that they could be initiated out of order (making sure the proxy server will wait for the message queue to be up instead of just exiting).

Next, I used the provided `gfclient_download` file to test basic file transfer. With my proxy server and cache server running, I ran `./gfclient_download` with the corresponding port number, and checked if it ran successfully. If it ran successfully, I then visually inspected the files in the `courses` folder to see if the images inside `cached_files` were accurately copied into the folder.

Next, I tested with `ipcstress.py` from `6200-tools` (downloaded from https://github.gatech.edu/cparaz3/6200-tools). There are three different test suites that I used: `base`, `soak`, and `stress`.

The `base` test helped test whether the basic implementation was working, as well as the handling of file not found errors, and files that were found but of size 0 bytes.

The `soak` test runs for a long time (large number of requests) with a fixed number of threads. Using this test suite, I was able to track down some memory freeing errors that I had within the proxy boss, such as using `gfserver_stop` twice, calling `unlink` and `munmap` out of order, and not destroying the read and write semaphores.

The `stress` test is meant to test the robustness of the implementation under a very heavy, concurrent workload. This test in particular helped me uncover an issue with my segment queue that was not apparent when running my other tests; under high concurrency, it revealed a flaw with my implementation of the mutex/cond around the message queue, occasionally causing a stequeue `underflow`.

For each of these tests, I could also see how many requests were completed sucessfully and how quickly. After passing all of these tests locally with good performance, my implementation was also passing all the checks in Gradescope.

References

- CURL library: https://curl.se/libcurl/c/
- POSIX documentation: https://pubs.opengroup.org/onlinepubs/9699919799/
- Using mq in POSIX: https://www.softprayog.in/programming/interprocess-communication-using-posix-message-queues-in-linux
- 6200-tools tests: https://github.gatech.edu/cparaz3/6200-tools
- How to use semaphores for IPC: https://stackoverflow.com/questions/32205396/share-posix-semaphore-among-multiple-processes
- IPC Overview lecture slides: - https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf
- POSIX message queues: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/MQueues.html
- Referenced PR1 for creating worker pool in cache server
- Flexible array members (for data buffer in shared_memory_segment): https://www.geeksforgeeks.org/flexible-array-members-structure-c/

The course slack! I read almost every message in the #project-3 channel. A few to point out in particular:

- Rain Kim's explanation here: https://omscs6200.slack.com/archives/CJ5G8BN30/p1707889116579699
- What to do with error handling: https://omscs6200.slack.com/archives/CJ5G8BN30/p1707943993456779
- Setting error options for error handling: https://omscs6200.slack.com/archives/CJ5G8BN30/p1707948329117409
- High level part 2design from Sean Scalabrini: https://omscs6200.slack.com/archives/CJ5G8BN30/p1709221948990939?thread_ts=1709153383.529869&cid=CJ5G8BN30
- Do not close file descriptor in the cache worker! https://omscs6200.slack.com/archives/CJ5G8BN30/p1711242387407659?thread_ts=1711240275.890329&cid=CJ5G8BN30
- Useful back and forth about mapping and queueing memory segments: https://omscs6200.slack.com/archives/CJ5G8BN30/p1711020940558529?thread_ts=1710950860.018299&cid=CJ5G8BN30
- Helped me debug my cleanup: https://omscs6200.slack.com/archives/CJ5G8BN30/p1711385955013549?thread_ts=1711379196.685599&cid=CJ5G8BN30
- ...and many more!