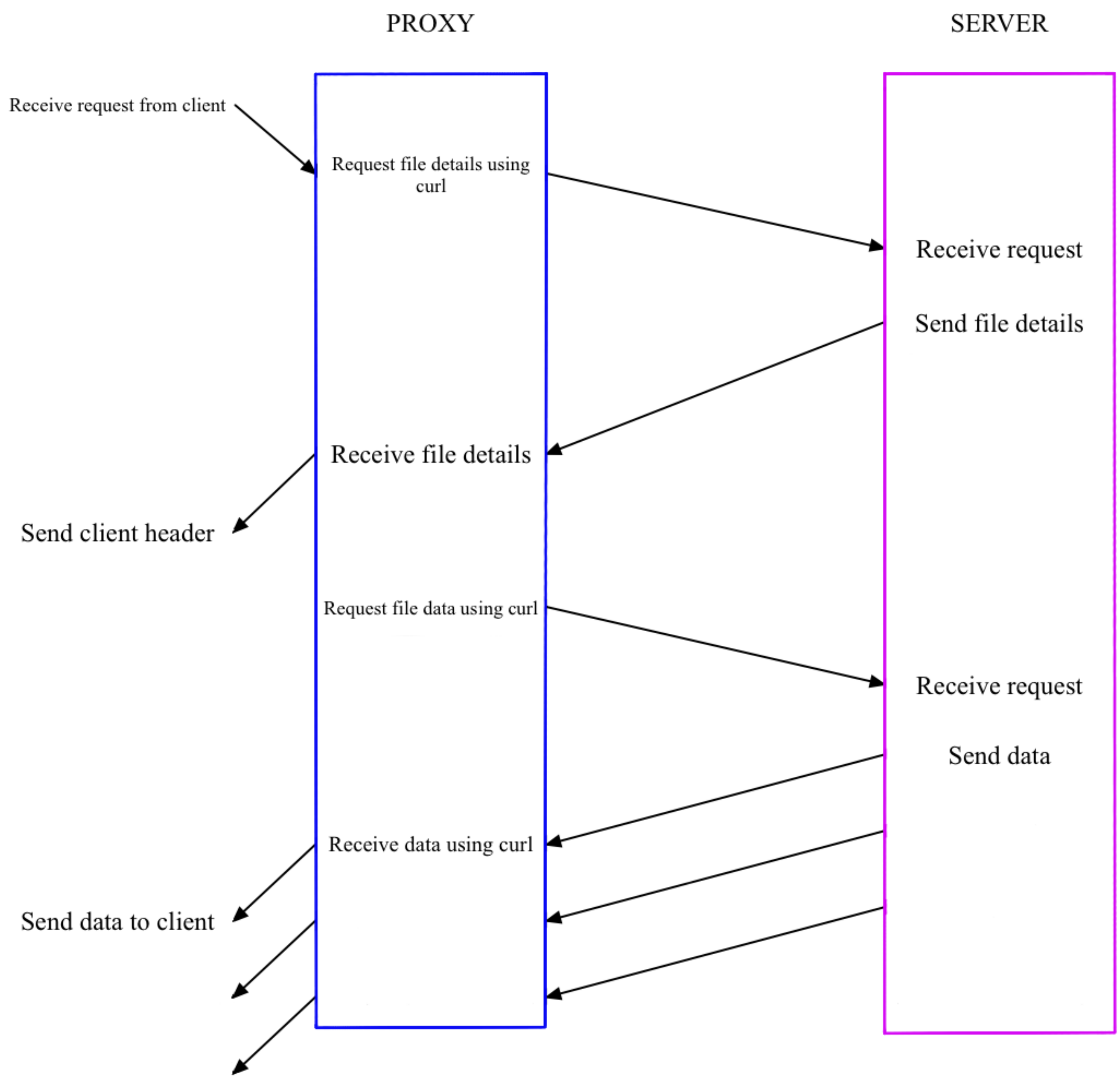
Table of Contents

Project Design:

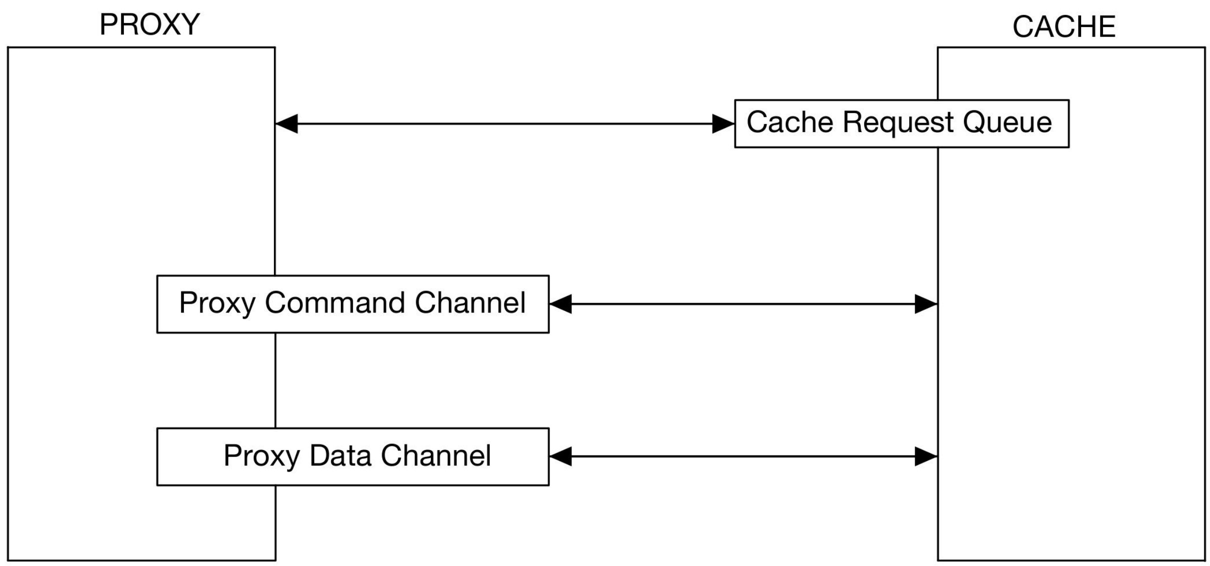
GETFILE Proxy – HTTP Server

1. A Client-Proxy-Server system.
2. Client-Proxy communication using GETFILE. [1] [2]
3. Proxy-Server communication using HTTP (using libcurl). [3]
4. Client sends a GETFILE request to the proxy.
5. Proxy sends the HTTP-equivalent request to the server.
6. If the file exists in the server, proxy sends a GETFILE OK header to the client, if not, proxy sends a GETFILE FILE_NOT_FOUND header.
7. If the file exists in the server, the proxy receives the file in chunks from the server and forwards it to the client.



<u>GETFILE Proxy - Cache</u>

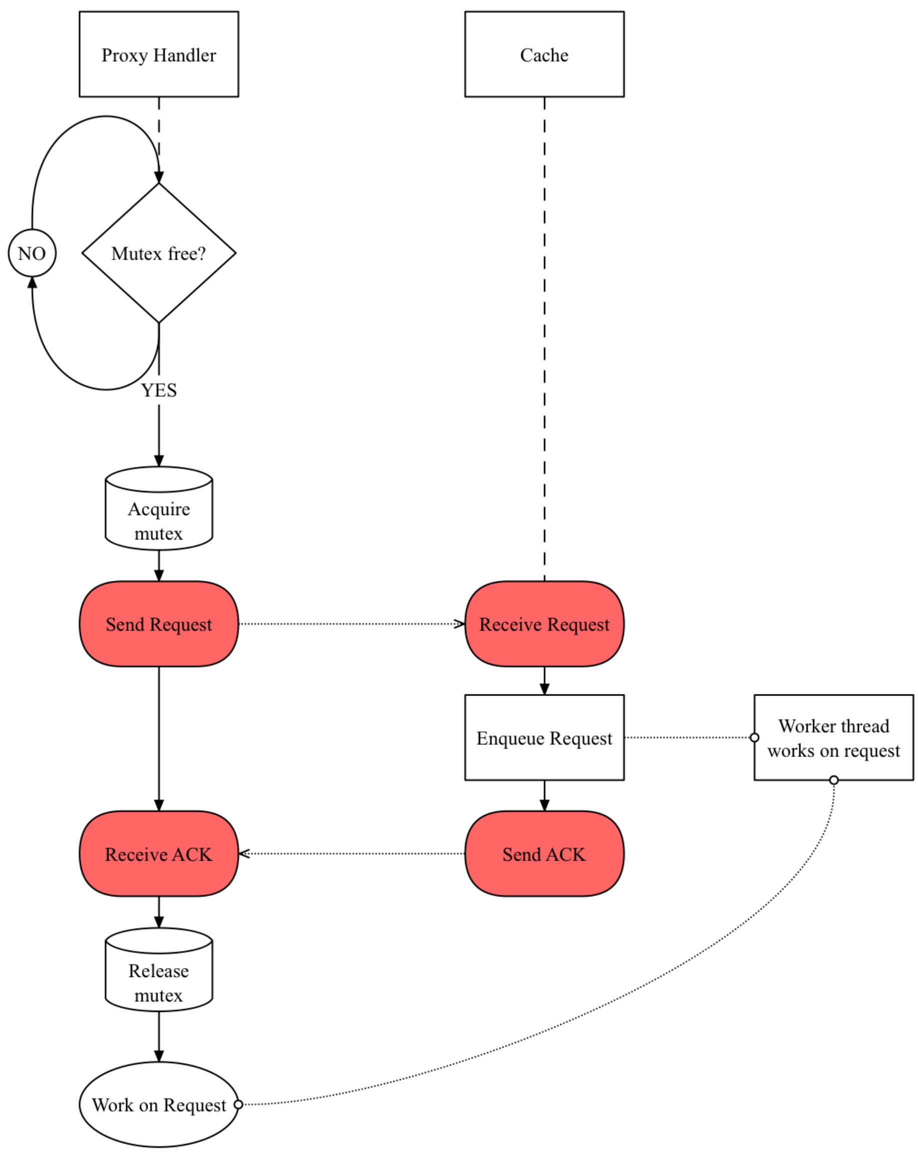
1. A Client-Proxy-Cache system.
2. Similar to the Client-Proxy-Server system, but the server replaced with a cache which is a different process running on the same machine as the proxy. [4]
3. Proxy has a number of shared memory segments.
4. Client sends a request to the proxy.
5. Proxy invokes the proxy handler to take care of the request.
6. Proxy handler acquires an available shared memory segment.
7. Proxy handler sends the request to the cache on cache's dedicated message queue.
8. Cache's boss thread receives the request and enqueues it.
9. A cache worker thread works on the request and communicates with the proxy thread using the proxy handler thread's command and data channels.
10. Proxy handler requeues the shared memory segment once the request is complete.



Critical choices and Trade-offs.

<u>How would the cache receive requests?</u>

- There needs to be some way the proxy sends the initial request to the cache.
- For this, the cache needs to have a "known" communication mechanism.
- In this project, we used a System V Tx and Rx message queue for this purpose. [5]
- The per-request command and data channels are owned by the proxy, and the cache request queue is owned by the cache.
- This separation in ownership allows any number of proxies to connect to a single cache.
- The actual message sent by the proxy to the cache are the details about the command channel for the request (the command channel shared memory key).
- The message on the proxy side is sent using a mutex, so only one request is sent at a time. An acknowledgement from the cache side confirms message reception. [6]



Who owns the per-request command channel? [5]

- In theory, the cache must be able to serve any number of proxies, only limited by the number of simultaneous threads in the cache.
- If the command channel is owned by the cache, the number of simultaneous requests that can be served by the cache can only be equal to the number of command channels.
- Also, an inactive proxy could degrade the cache's performance by hoarding the command channel indefinitely, or a malicious proxy could corrupt the command channel to receive some other request's data.
- Hence, it makes the design more effective for the command and data channels to be owned by the proxy instead of the cache.

What does the command channel look like? [5] [7]

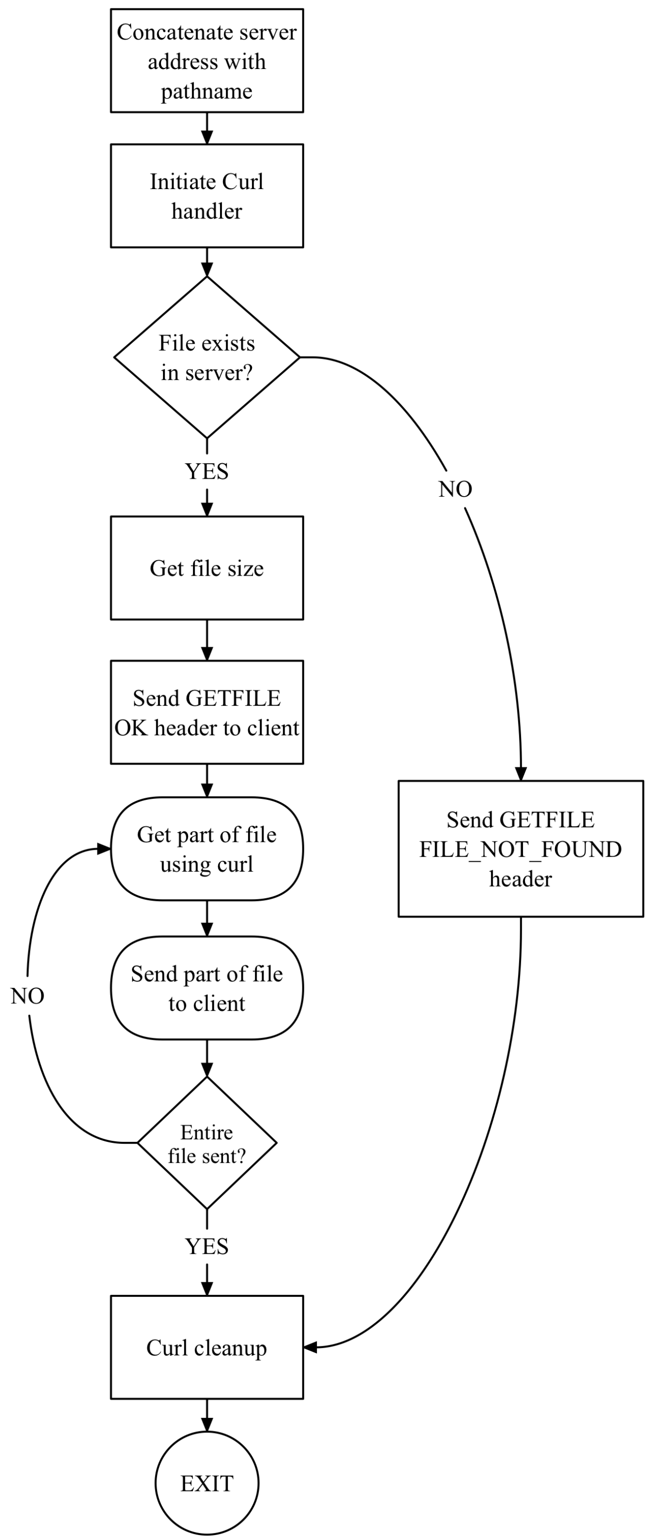
- The per-request command channel, as discussed above, is owned by the proxy.
- The command channel key is the only information the proxy handler sends to the cache-request message queue.
- The command channel is a System V shared memory that has the following variables,
    - The command channel's own key – so the proxy handler can send the key to the cache.
    - The data channel key – the data channel is also a System V shared memory.
    - The size of the data channel segment – so the cache can map the data channel.
    - Requested file path – so the cache knows what file is requested.
    - File size – so the cache can let proxy know the size of the file (if the file exists).
    - Message control – flags to indicate events like file not found, file found, all file sent, etc.
    - Synchronization Tx and Rx – Lightweight System V message queues for synchronization between the proxy thread and the cache thread.

How can multiple proxies send requests to the same cache?

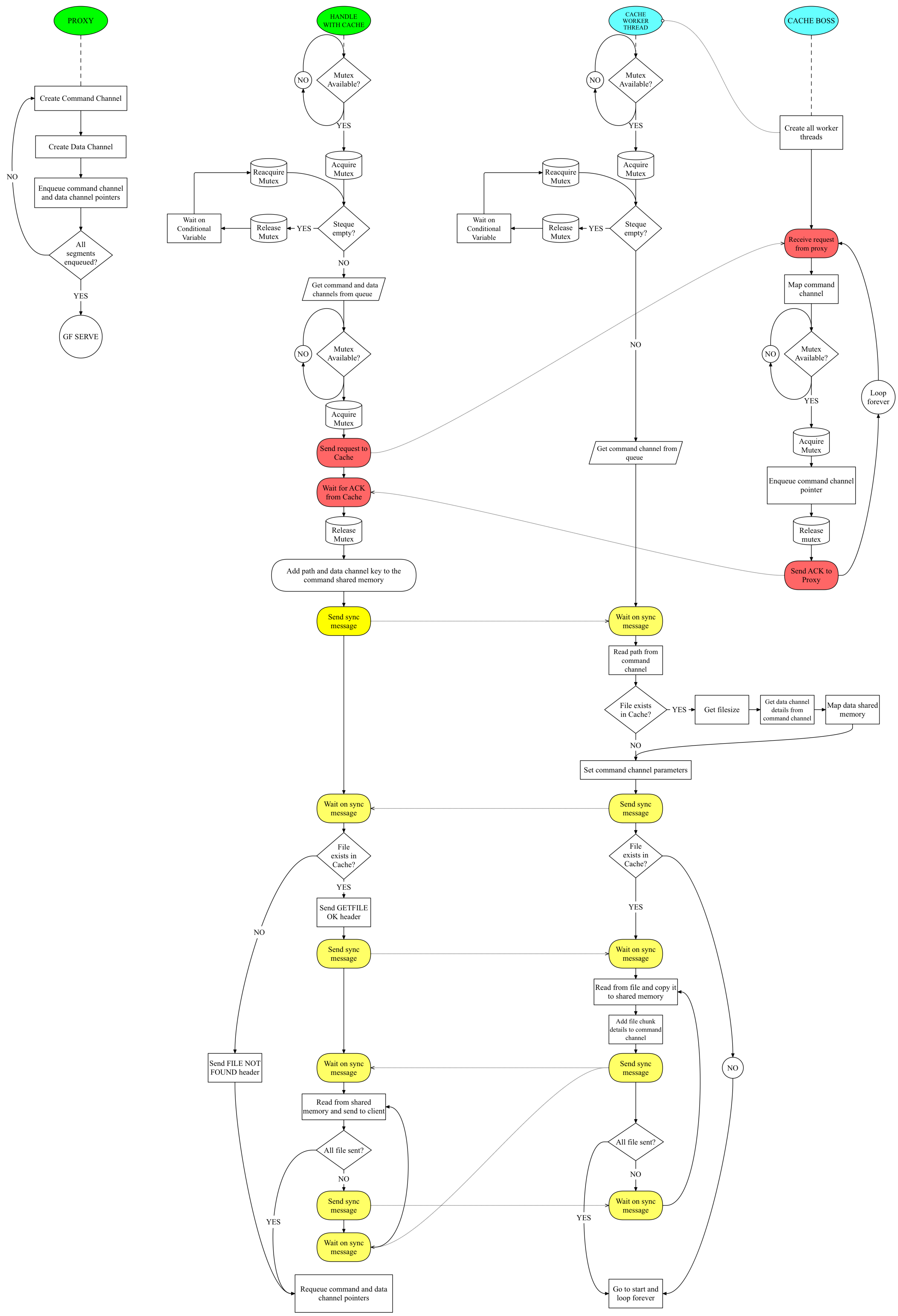
- The cache receives requests in the form of the shared memory command channel key that the proxy sends.
- Multiple proxies can send requests to the same cache given only one thread sends a request at one time and the cache request queue is held by that particular thread until it receives an acknowledge from the cache.
- So, for multiple proxies to use the same cache, the proxy threads in different proxy processes need to use a process-shared mutex (or any other process-shared synchronization construct) to make sure only one request is sent to the cache at one time.

Flow of control

<u>Handle with Curl</u> [3]



Proxy – Cache [3] [6] [7] [8] [5]



Code Implementation

Proxy – HTTP Server: [3] [7]

1. Proxy (*webproxy.c*) initializes the GETFILE server (*gfserver_init*).
2. Proxy sets the thread worker handler (*GFS_WORKER_FUNC*) and the argument for the handler (*GFS_WORKER_ARG)*.
3. Handler (*handle_with_curl.c*) gets invoked when there is a request.
4. Handler concatenates the server address (*arg*) with the requested pathname (*path*) to form a valid web address.
5. Handler initiates curl handle (*curl_easy_init*) and sends request to the http server to get the file details (no body – *CURLOPT_NOBODY*).
6. Handler then sends the appropriate header and the file size if the file exists to the client (*gfs_sendheader*).
7. If the file exists, handler makes another request to the web server to get the file data.
8. The file is directly sent to the client (*gfs_send*) using the curl handle's "write function" (*CURLOPT_WRITEFUNCTION*).
9. If all of the file is sent successfully, we clean up curl handle (*curl_easy_cleanup*) and return.

Proxy – Cache: [8] [6] [7] [5] [1] [2]

- Proxy:
  1. Proxy (*webproxy.c*) creates message queues to communicate with the cache (*msgget*).
  2. Proxy creates the System V shared memory command channels and the data channels (*shmget* and *shmat*), and enqueues them. The data channels will be of the size specified (*segsize*).
  3. Proxy then initializes the GETFILE server (*gfserver_init)*.
  4. Proxy sets the thread worker handler (*GFS_WORKER_FUNC*) and the argument for the handler (*GFS_WORKER_ARG)*.
  5. Proxy handler (*handle_with_cache.c*) gets invoked when there is a request from a client.
  6. Proxy handler uses a mutex to dequeue a pair of command and data channels.
  7. Proxy handler uses another mutex to send the command channel key to the cache.
  8. Proxy handler then copies the requested file pathname (*path*) to the command channel, sends a sync signal (*msgsnd*) using the command channel's sync message queue, and waits for a cache worker thread to respond (*msgrcv*).
  9. Once the cache worker responds, proxy handler sends the appropriate header to the client based on if the file exists and if it does, the file size (*gfs_sendheader*).
  10. If the file exists, proxy handler syncs with the cache worker again to let cache worker know it is ready to receive the file.
  11. Once the proxy handler receives sync ack, it starts reading from the shared data channel and keeps sending it to the client (*gfs_send*) until all file is received, at which point it simply returns.

- Cache:
  1. Cache boss creates the worker threads (*pthread_create*).
  2. Cache boss also creates the dedicated cache message queue on which proxies will send requests (*msgget*).
  3. Cache boss waits for a request to arrive (*msgrcv*). When a request arrives, since the request is a shared memory command channel key, the cache boss creates the command channel pointer (*shmget and shmat*) and enqueues (using a mutex) the command channel pointer. Once the pointer is enqueued, the cache boss sends (*msgsnd*) an acknowledgement to the proxy handler and goes back to listening on the dedicated message queue.
  4. The cache worker (using a mutex) dequeues a command channel pointer, creates the sync message queues specified by the command channel, and waits for proxy handler's response.
  5. Once the cache worker gets the confirmation that the proxy handler has copied the requested pathname into the command channel, the cache worker checks if the file exists (*simplecache_get*), and if it does, calculates the size of the file (*st_size*), and copies all this information to the command channel. The cache then signals the proxy handler and waits for an acknowledgement (if the file exists).
  6. If the file exists, the cache worker maps the shared memory data channel using the data channel key and size in the command channel.
  7. The cache worker then starts copying the file in segment-size chunks to the data channel (*pread*) and sends a signal to the proxy handler to start reading. The cache worker then waits for an acknowledgement from the proxy handler to read the next chunk of the file.
  8. Once all the file is read and copied into the data channel, the cache worker changes the command channel flags, so the proxy knows.
  9. The cache deletes the command and data channel mappings (*shmdt*) and goes back to serving another proxy handler request.

Testing

- The testing for this project was done using Python.
- The proxies in both parts of the project were tested using Python clients.
- The *Pytest* module was used for the unit tests. [9]
- Various file types with sizes ranging from 1KB to 100MB were used for the testing.
- Numerous combinations of number of threads, number of segments, segment size, number of files requested, etc. were used wherever applicable in the client, proxy and cache processes.
- The orders of when each of the process (client, proxy, and cache) would start was randomized.
- The cache was also tested using multiple simultaneous client and proxy processes.
- An additional Python script was run after each test to verify the contents of the sent and received files.
- Although Gradescope was not intended to be used as a test harness, a lot of times, Gradescope revealed further avenues for testing.
- ***All the Gradescope tests passed for each of the project sections.***

Works Cited

[1] A. Raman, *Project1 Multithreaded GETFILE Client Code.*

[2] A. Raman, *Project1 Multithreaded GETFILE Server Code.*

[3] "libcurl," [Online]. Available: https://curl.haxx.se/libcurl/.

[4] A. Raman, *Handle with curl code.*

[5] "Linux manual page - System V," [Online]. Available: https://man7.org/linux/man-pages/man7/sysvipc.7.html.

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

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

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

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

[10] B. Hall, "Beej's Guide to Unix IPC," [Online]. Available: https://beej.us/guide/bgipc/html/multi/mq.html.