CSE130 Assignment 3: HTTP Server Proxy

Carson Petrie CruzID: cepetrie Email: cepetrie@ucsc.edu

1. Program Overview

The purpose of programming assignment three was to create a HTTP proxy, which would invisible redirect the requests from a client to a specified set of servers whilst monitoring their performances.

1.1 Program Requirements

Our program required that we create a *httpproxy.c*, a file capable of redirecting client requests to an optimal server. Our *httpproxy.c* utilizes internal states to estimate server loads, and after an interval specified by the user will send a healthcheck to all the servers to account for non-proxied requests. *Httpproxy.c* must also support a caching implementation such that if specified our program will remember a specified number of files up to a specified size to be immediately returned to the user.

1.2 Program Restrictions

Like most previous assignments, there are minimal restrictions to this program. We are not allowed usage of the FILE or HTTP libraries for C, and our program must compile without warnings or errors.

1.3 Program Structures

The following are structure definitions. They will be explained in later sections, but put here as a reference for the reader.

```
typedef struct
Server {
    int ID;
    int port;
    int requests;
    int failures;
    int status;
} Server;
```

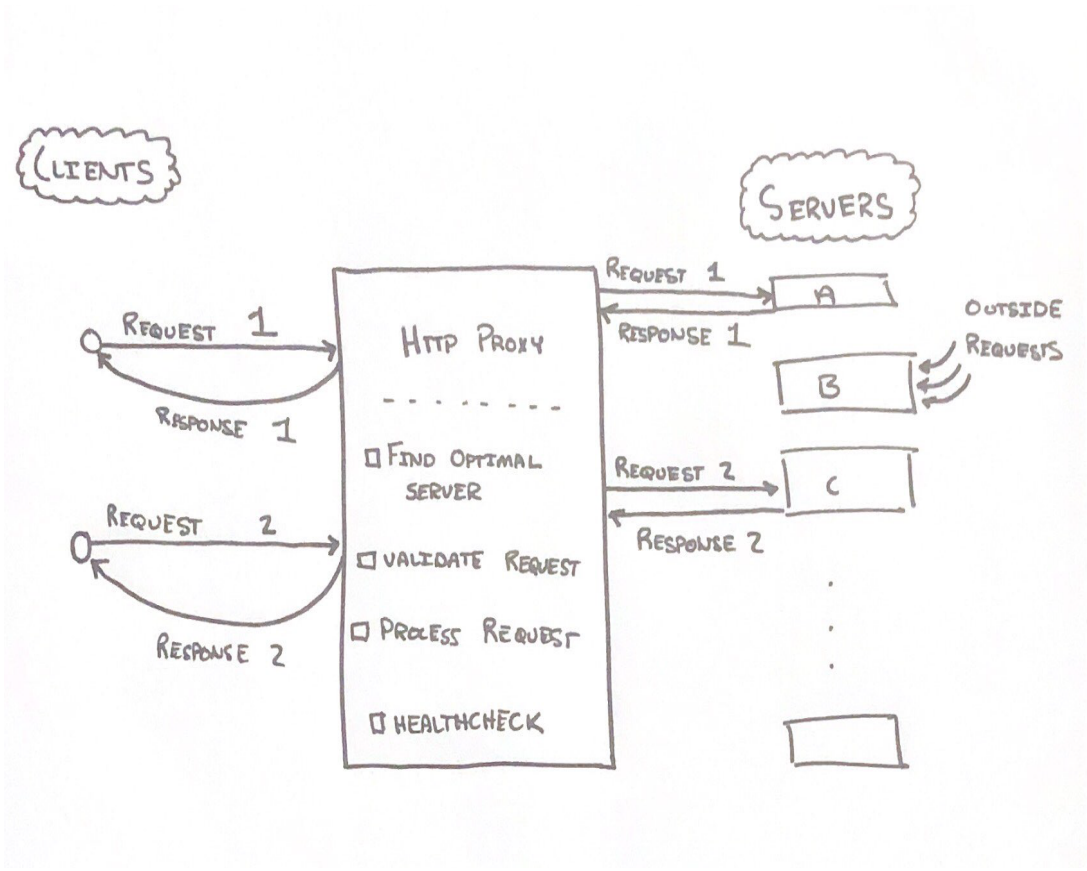
```
typedef struct Arguments {
    int proxyPort;
    int connections;
    int healthchecks;
    int cacheSpace;
    int cacheSize;
    int caching;
} Arguments;
```

```
typedef struct File {
    char fileName[20];
    char *body;
    char lastModified[100];
    int bodySize;
    int serverPort;
} File;
```

```
typedef struct cacheQ {
    int head;
    int tail;
    int length;
    int count;
    File *arr;
} cacheQueue;
```

2. The Client, Proxy, and Server Relationship

The following diagram depicts a zoomed out view of the relationship between a client, proxy, and server. Please observe that the HTTP Proxy control flow, as well as each component of said control flow we be respectively expanded upon in the following sections three and four of this design document.



From this diagram, we can see that the primary responsibility of our HTTP proxy is to transparently process the request from a client by sending and receiving the client's data to the appropriate server, and then replying to the client with the server's response. As illustrated, outside requests are possible to servers.

The client to server communication channel can be broken down into two connection sockets, the **client to proxy connection**, and the **proxy to server connection**. In terms of implementation of *httpproxy.c*, each of these connections are a socket. This relationship also shows how our HTTP proxy functions both as a server, as well as a client throughout its execution. When done correctly, our client will not even realize that a proxy is running.

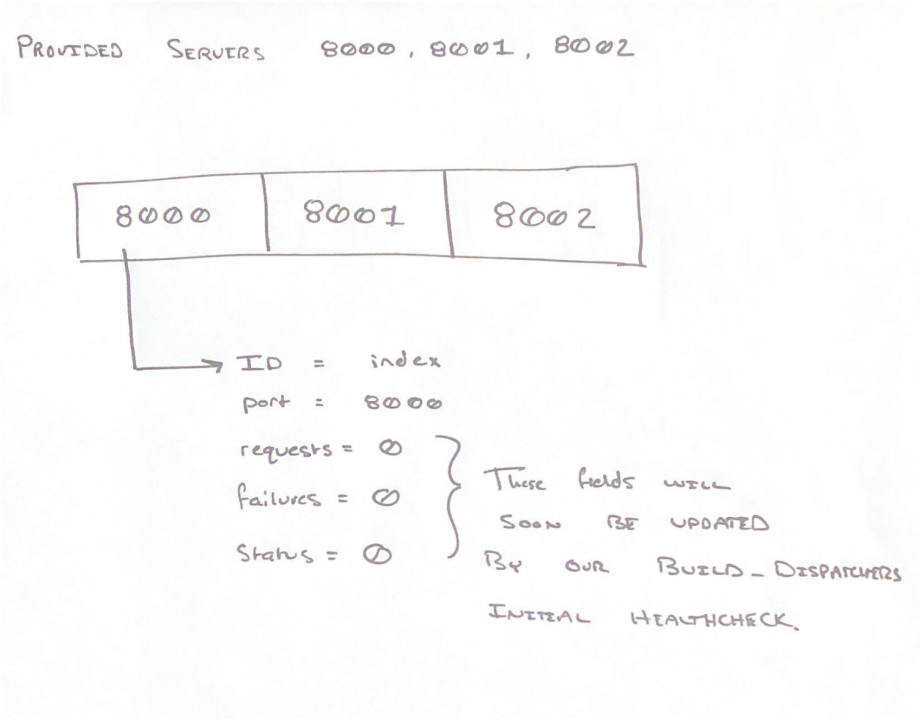
2.1 - Storing Servers

Our HTTP proxy stores the servers inside a global server struct array. The server structure's type definition is as follows.

```
typedef struct
Server {
   int ID;
   int port;
   int requests;
   int failures;
   int status;
} Server;
```

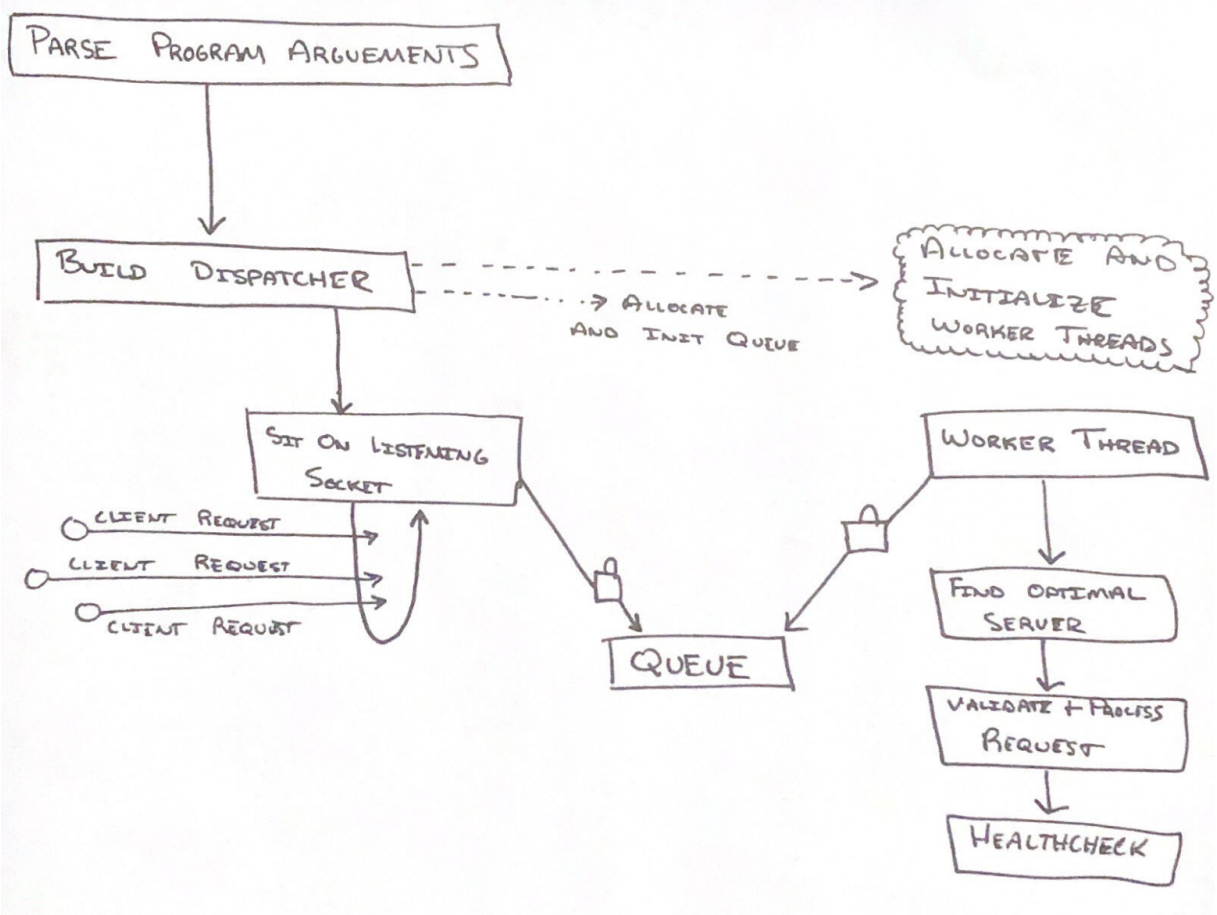
By storing a global array of servers, which we parse out during our command line parsing in main, we allow anywhere in our code to access and check these values. However, our server's structure becomes a **shared resource amongst threads.** Because the servers are modifiable, they are considered a critical resource and any interaction of the threads with the server can be defined as a **critical region.** Therefore, servers are protected with a server lock.

The following diagram visualizes the server container structure in my HTTP proxy implementation.



3. HTTP Proxy Control Flow

The following flowchart depicts the overall control flow of my implementation of *httpproxy.c*. Please note that many components of this chart (Worker/Dispatcher handoff, optimal server calculation, request processing, etc…) are further depicted and explained in their respective sections.



This diagram shows the first critical region of our code, the queue data structure which mediates the hand-off of the client request connection sockets grabbed by the dispatcher on the listening socket to the worker threads. This design is fairly similar to the setup for my *httpserver.c* implementation in assignment two.

4. Program Functionality

The following section will address the following topics…

- Parsing Command Line Arguments and Default Values
- The Worker Dispatcher Interaction
- The Queue Implementation
- Optimal Server Calculation
- Request Validation
- Request Processing
- Healthchecks
- Caching
-

4.1 - Parsing Program Arguments

Program arguments are parsed utilizing the *getopt()* command paired with a switch statement. Parsing is the first step in our program because any initialization of worker threads, queues, or caching structs are dependent upon the client's specific data in the command line. The following pseudocode demonstrates my approach to parsing command line arguments.

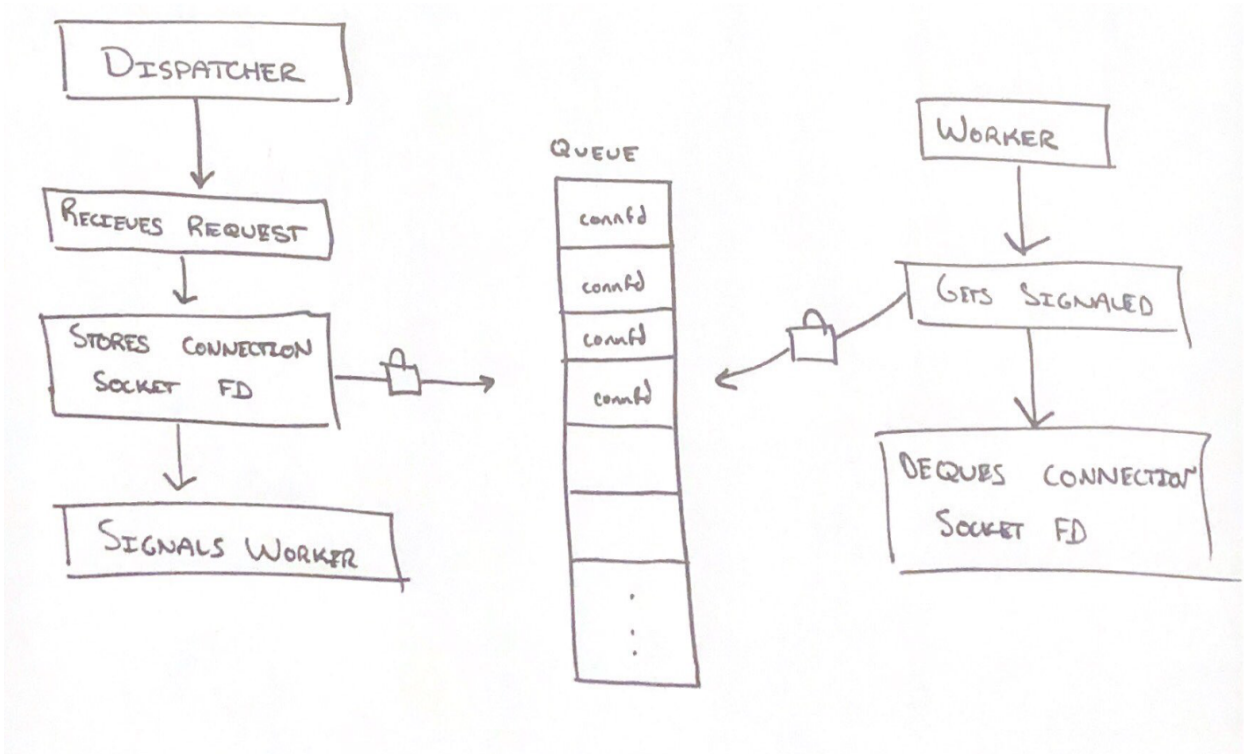
```
while getopt() != -1

    switch()

        case ‘N’ -> update number of parallel connections from default of 5 to input

        case ‘R’ -> update healthcheck interval from default of 5 to input

        case ‘s’ -> update cache size from default value of 3 to input

        case ‘m’ -> update cache size from default value of 1024 to input
```

Parsed Arguments are stored into the following globally defined structure. Clearly, parsing program arguments is straightforward enough and does not really require any more explanation besides this and the *getopt()* man pages if desired.

```
typedef struct
Arguments {
    int proxyPort;
    int connections;
    int healthchecks;
    int cacheSpace;
    int cacheSize;
    int caching;
} Arguments;
```

4.2 Worker Dispatcher Interaction

The following chart visually depicts the interactions between the dispatcher and worker threads.



Clearly, the client connection sockets that we need to pass from the dispatcher to the worker is a key piece of data. What is even more important than the client connection sockets themselves are how we store and protect them whilst many threads compete to grab one and process it.

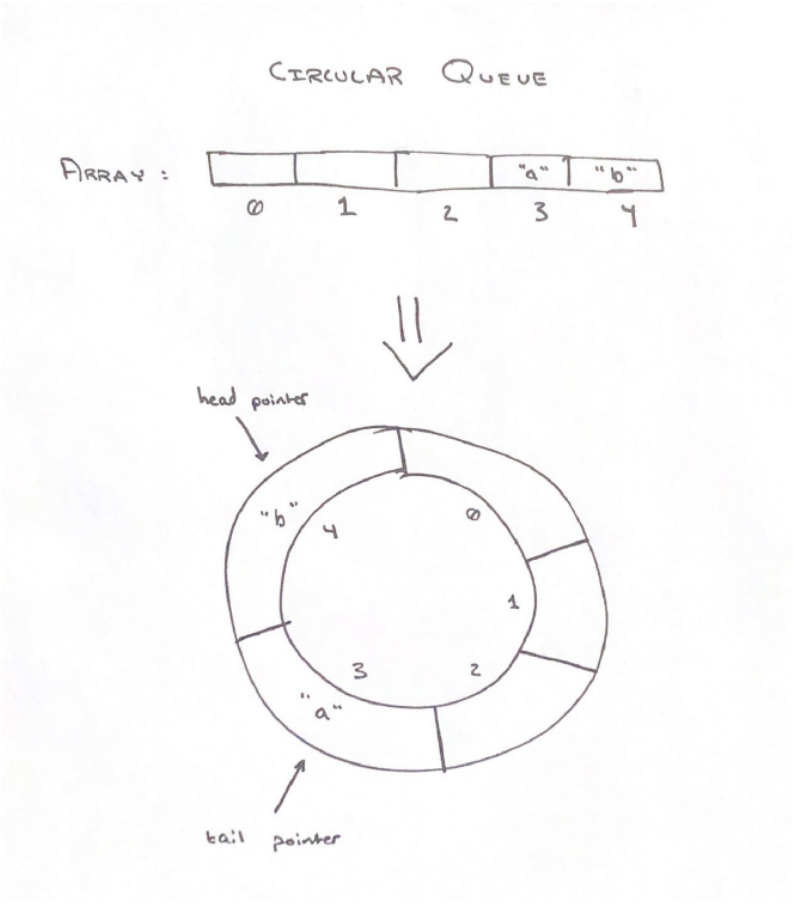
The Queue holding the client connection sockets is a **critical region** because it is a modifiable data structure which is **shared amongst threads**. We need to ensure that only one thread is interacting with the Queue because we only want one socket being processed by one thread, and we also want to avoid the numerous issues that would arise within the memory itself if multiple threads are enqueuing and dequeuing data without any synchronization.

We maintain a **thread-safe** implementation by using locks and condition variables. In junction, these allow our worker threads to sleep until signaled by our dispatcher that they have put something in the queue. Once signaled, the threads will race to grab the lock. One thread will win, and they will in turn process the request.

4.3 Queue Implementation

The first thing addressed in our design of a queue, is why use a queue at all? I chose to implement a queue because of two reasons. First, our queue of tasks is a critical region. This means we want to minimize the time a thread takes in it as much as possible. Both of our threads interactions with the queue, enqueue and dequeue, have O(1) runtime. This means our critical region is locked and our program is therefore bottlenecked for a minimal amount of time. The second reason is that managing a queue as a critical region is conceptually simple. We only need one lock to keep track of, and just wrap our calls to the queue inside this lock.

For my implementation of a queue, I utilized an old circular queue I had lying around from CSE101. A circular queue fits in as a nice upgrade to my existing array of tasks because the underlying data structure for a circular queue is an array.

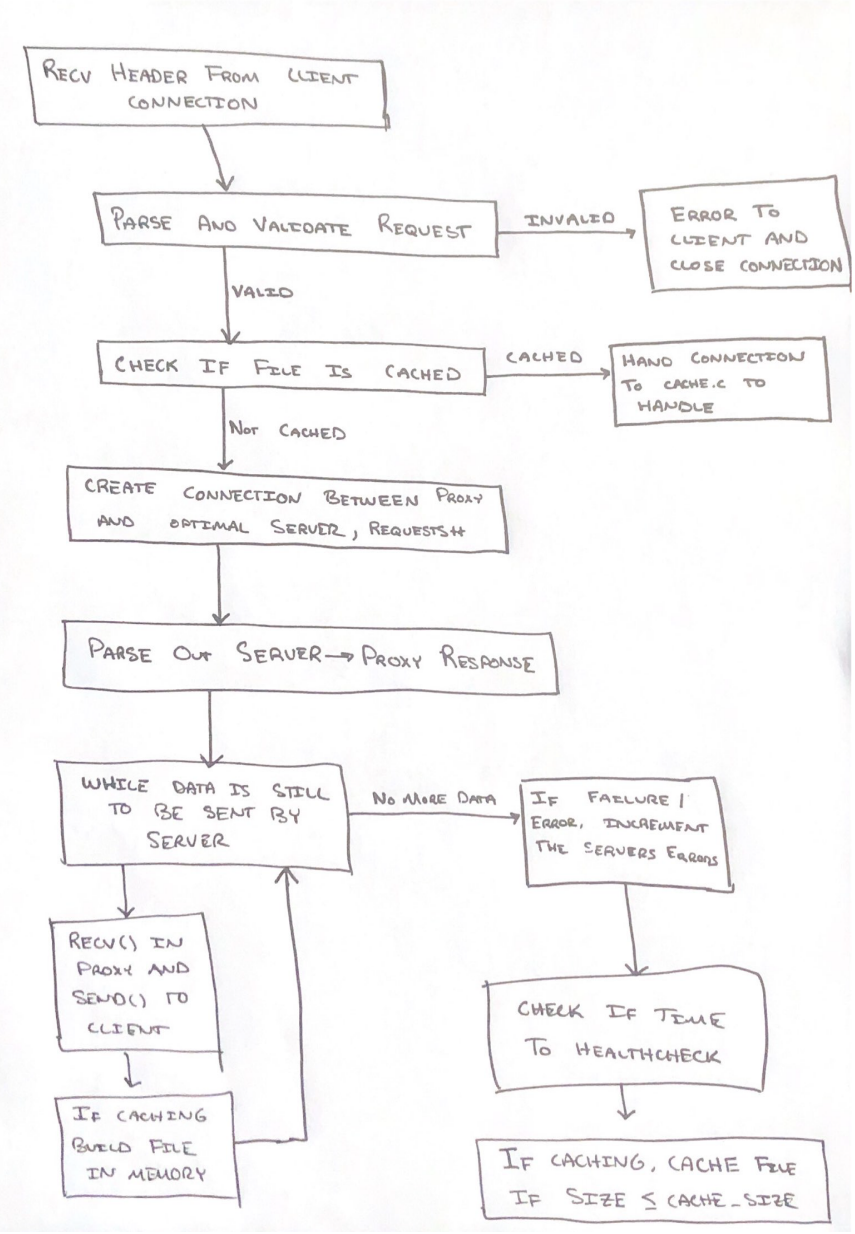


The diagram shows an array extended into a circular queue. In the context of assignment two, “a” and “b” would be integers representing connections made to our server for each thread to grab and process.

Another reason why I chose a circular queue is because it is not unbounded. This will introduce **backpressure** to clients if they send many requests, by having a full enqueue wait for a dequeue to signal that space is available.

4.4 Request Processing

Request processing is conceptually simple but somewhat arduously implemented due to the amount of parsing required to achieve connection forward between the client, our proxy, and the server. The following flowchart depicts the control flow of the *process_request()* function.



As observed, the control flow or implementation of the *process_request()* function is similar to that of the *handle_request()* from assignment one. Any parsing logic can be found in either assignment one or two's documentation.

The primary difference in this design is that we need to account for caching. We first see after parsing the header from our clients request if our file is already cached. If it is, we hand off the control flow to *caching.c* where whilst checking our cache for the file name, it will respond to the client and close the connection if a file is found and identical to the requested one.

4.4 Request Processing (continued)

```
typedef struct File {
        char *fileName;
        char *body;
        char lastModified;
        int bodySize;
        int serverPort;
```

In the event that we want to cache, and our program has not detected a file in the cache with an identical last modified timestamp, we want to insert a File struct that our *httpproxy.c* builds during its request processing to our cache utilizing our *cache.c* library. We achieve this by using a File struct, which is again defined in this section.

While our request handling is in its while loop and continuously handing off data from our server to the client, we use this as an opportunity to build a File struct. A variable is used to track the size of the File body, as well as to check if the body is capable of holding the next batch of data. The pseudocode for building our file struct is as follows

```
while we have not sent the client all the bytes the server told us it would send

    bytes read = recv(server into buffer)
    bytes sent += send(buffer to client)

    new file size = file.bodySize + bytes read

    if our new file size is not larger than our allowed size for the file
        memcpy(file body offset by file.bodySize, buffer, bytes read)

    update file.bodySize to = new size
```

Simply, we utilize an integer value to keep track of what the size of our file's body would be if updated. In the event that the update would not exceed the size allocated for our file's body, we use an offset to in essence concatenate the new buffer which holds the data from our server to proxy response to the existing file body.

Overall, the majority of our request proxying is **thread-safe**. There are few checks to the cache, a **shared resource** that we must lock, but that implementation is discussed in the following section. Our request processing however, can be run synchronously for it's most expensive operations, parsing, our recv() and send() while loop, and the construction of our File structure.

4.5 Caching

Caching is implemented as a library of functions and a custom data structure in a separate file, *caching.c*, which is utilized by *httpproxy.c* during its request proxying period. *Caching.c*'s implementation is centered around a slightly modified custom queue. The cacheQueue data structure, as well as its included File structure are again shown.

```
typedef struct File {
        char *fileName;
        char *body;
        char lastModified;
        int bodySize;
        int serverPort;

typedef struct cacheQueue{
        int head;
        int tail;
        int length;
        int count;
        File *arr;
```

The cacheQueue is initialized by our build_dispatcher() function as shown in the previous diagrams. The cacheQueue is allocated to a length equal to the cacheSize variable that was parsed out during the initial command line parsing. A cacheQueue would be an array of size cacheSize that holds File structures, where each File structure is a struct that has been built and populated during the request processing period by our *httpproxy.c*.

The reason I selected a circular queue structure again is because we can utilize a tail and head pointer to easily maintain a FIFO structure because it is given to us at compile time the size and space of our cacheQueue.

4.5.1 - in_cache()

The first function in our caching file is in_cache(). This function will search through a cacheQueue for a file name. If it finds a matching file, it will compare the last modified timestamp of the file currently residing in the cache with the File struct built from the client's request. In the event that the cached file is the same or newer, our proxy will instead return the cached file as opposed to sending a request to the server to be handled.

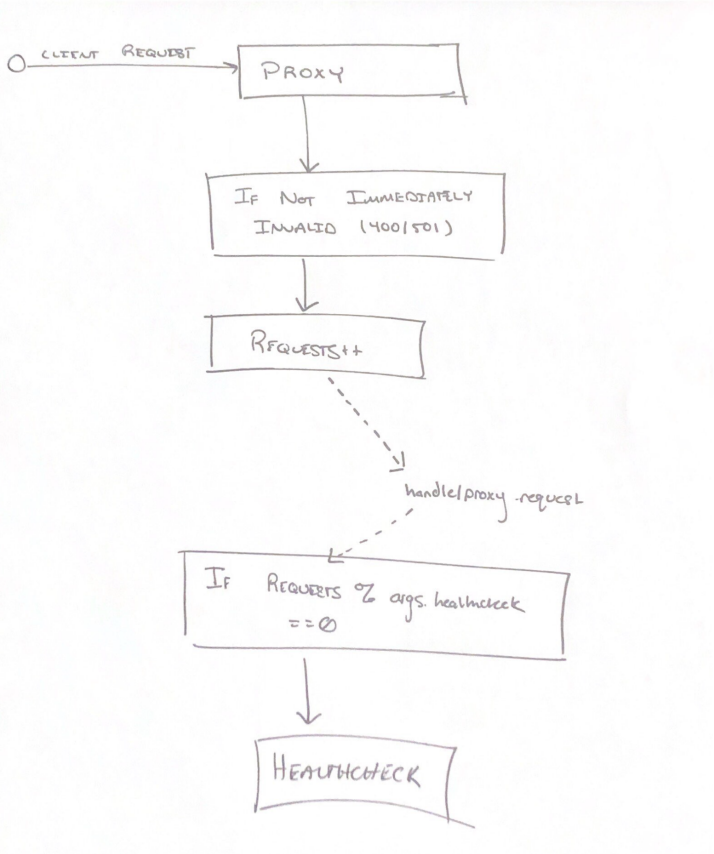
4.5.2 - insert_cache()

Insert_cache() will take a File struct provided by *httpproxy.c* and attempt to insert it into the existing cacheQueue data structure. The only case in which insert_cache() will fail is if the size of the body of the File struct is larger than the capacity of our cache. Remember from the request processing implementation that the File structs size will always be incremented, but we will only store memory up to the allowed cache capacity. Insertion is simply done by enqueueing the File struct. Enqueue() for this cacheQueue structure will overwrite the oldest file in the cache.

Caching is home to a couple of **synchronization concerns**. The cache is a shared resource amongst threads. Since our insert_cache() function will modify the cache itself, this means that any access to our cache resource is a **critical region**, as multiple threads working with the cache at once could introduce a host of errors and non deterministic behavior. To combat this, we utilize a lock around our cache. Anytime that our caching library is utilized by our HTTP proxy, we lock the cache and then unlock it after returning.

4.6 Healthchecks

Healthchecks are sent after -R commands received by our proxy. We compare our globally shared request counter against our global argument structures healthchecks field. The difference between these two global variables is that our argument structure is **shared amongst threads but not modified**. On the other hand, our request counter **is shared amongst threads and modified**. This means that our request counter is a critical resource, and any interaction with it is a **critical region**.



The proxying process for choosing requests to count looks like the diagram to the left. The pseudocode of the healtcheck_servers() function itself will be discussed in the following section.

The drawback of implementing health checks at the end of request processing is that we need to lock our worker threads to ensure a **thread-safe** implementation of our httpproxy logic. If I had more time, I would redesign and create a dedicated healthcheck thread which would sleep until signaled by a worker thread, instead of having to lock all worker threads to process a healthcheck at the correct interval.

4.6 Healthchecks (continued)

As shown in the diagram, when the healthchecks have reached the required number of requests to initiate a healthcheck (requests % argument.healthchecks == 0) it makes a call to the healthchek_server() function.

Because healthcheck servers examine the server resource, a **shared modified resource**, we need to ensure that we have locked our worker threads. This means that every call to healthcheck_servers() is wrapped by a lock that is shared amongst the threads.

The healthcheck_servers() pseudocode is as follows:

```
for (int i=0; i < number of servers; i++)

        build a healthcheck request to the current server in request[] string

        create connection socket between proxy and current server
        if connection fails
                set servers[i].status to be down (-1)
                continue;
        send request string to current server

        if parsing response fails, or response code is not 200
                set servers[i].status to be down (-1)
                Continue;

        parse our requests and failures form servers response to proxy
        Increment requests to account for the healthcheck itself

        set the server[i].failures and server[i].requests to our parsed failures
        snd requests from the server
```