Design for Assignment 3: HTTP Proxy

1.0 Introduction:

1.1 Objectives

The goal is to create a load balancer to work with multiple HTTP servers, and to forward multiple requests from clients for reading files. To do so, the program will have the behavior of both a server and a client. This project left off previously with a preceding program, “httpserver”, a multi-threaded server that executes commands like “GET”, “PUT”, and “HEAD”. This load balancer, named “httpproxy”, will be limited to accepting “GET” commands only, and connect with multiple instances of the server, as well as adding a caching option for the proxy, and a health check feature to assess the workload of each server to assign work accordingly.

1.2 Statement of scope

This project is intended to create a server to support multiple users, or a user with multiple concurrent requests, to read files using simple Linux commands.

The proxy is expected to interact with two types of users: the host, and the client.

<u>Host</u>

The only tasks available for the host is to start or end the proxy. When launching the proxy, one must provide a mandatory port to connect to clients, at least one additional port for a server, and additional arguments for the proxy behavior.

The first step is to build an executable file. By opening a Linux terminal, this can be done by running the command “*gcc httpproxy.c -o httpproxy”*, or simply run “*make”*. The second option is made possible due to the Makefile included with it.

The second step is to run the executable, starting with “./httpproxy”, followed by other arguments like client port number, server ports, parallel connection count, health check frequency, cache capacity, and cache max memory. For example:

"./httpproxy 1234 8080 8000 -N 7 -R 3 -s 5 -m 2048"

The order of arguments is irrelevant, but it is mandatory to include a port number for the client as the first argument. If the "-N" and "-R" options are left out, the server will by default set the values to 5. If the "-s" and "-m" options are left out, the server will have a cache storing up to 3 files under 1024 bytes.

The design of the program is to start and run indefinitely. To close the server, simply go to the terminal and use "command + C" keys to force terminate the process.

Client

From the client's point of view, there is only one type of legal commands to perform on a file:

"GET": Fetch a file from server to client.

When submitting a request, the client will receive a response identical to one of a functioning server.

1.3 Software context

The project is developed to be a customized Linux client-server system as a learning experience for a computer science class. It is expected to inherit the behavior of any typical load balancer. This program is placed in a school setting, so it should meet the standards of clients like students and academic staff, who would run test programs for all functionality. Instructional staff will execute trial runs to match expected outcomes for academic evaluation and grading, based on overall performance, stability, and accuracy. This program may continue to be developed and assessed for future assignments, which may require implementing additional features to the project.

1.4 Major constraints

One constraint for the server design is to run flawlessly in a Linux environment like Ubuntu 18.04. On the other hand, client requests are unbounded and may be from any source, not limited to terminal commands, browsers, or other client programs.

1.5 References

Fall 2021 Assignment 1 sample starter code, Peter Alvaro, https://canvas.ucsc.edu/files/4456847/download?download_frd=1

Spring 2020 Assignment 1 sample starter code, Michael Covarrubias, https://git.ucsc.edu/mdcovarr/cse130-section/-/blob/master/week-4/server.c

Spring 2020 discussion session code, Clark Hilbert, https://git.ucsc.edu/cse130/spring20-palvaro/cwang120/-/blob/master/asgn1/httpserver.c

Spring 2020 Assignment 1 submission source code, Chien-Yu Wang, https://git.ucsc.edu/cse130/spring20-palvaro/cwang120/-/blob/master/asgn1/httpserver.c

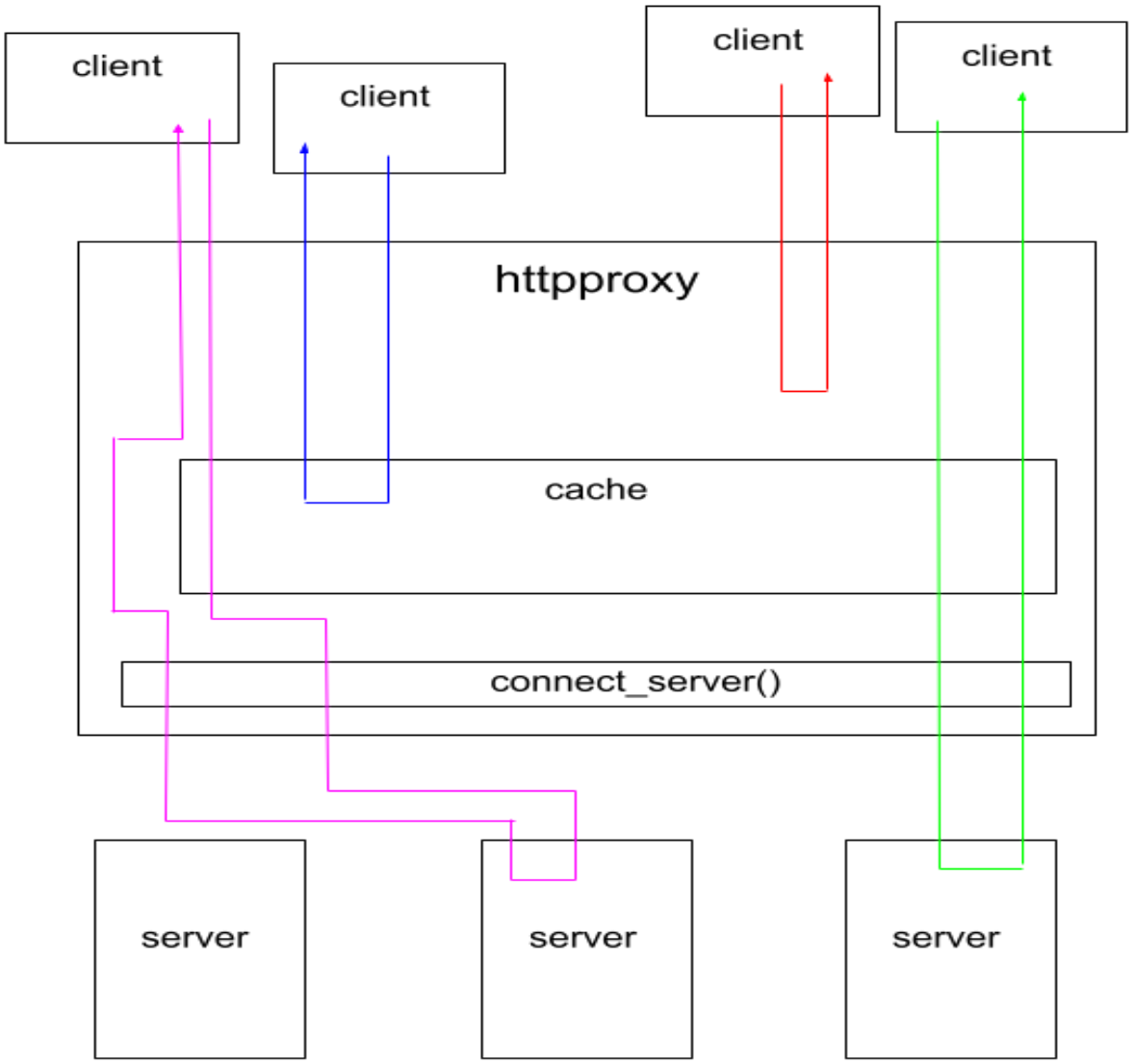
2.0 Architectural and Data Design:

2.1 Architecture

The following component diagram illustrates the overall architecture and design of this server. On the most surface level, a client expects identical responses from an actual server when accessing the program. Internally, however, when “httpproxy” receives a number of requests, it is evaluated if there is a necessity to connect to one of the available servers, or if the program can handle a request on its own.

2.1.1 Load Balancing

The program monitors the workload of each server to evaluate the best candidate to forward requests to. After completing a fixed amount of requests, it sends a health check to each server and chooses the one with least tasks performed, as well as the error count when there is a tie.



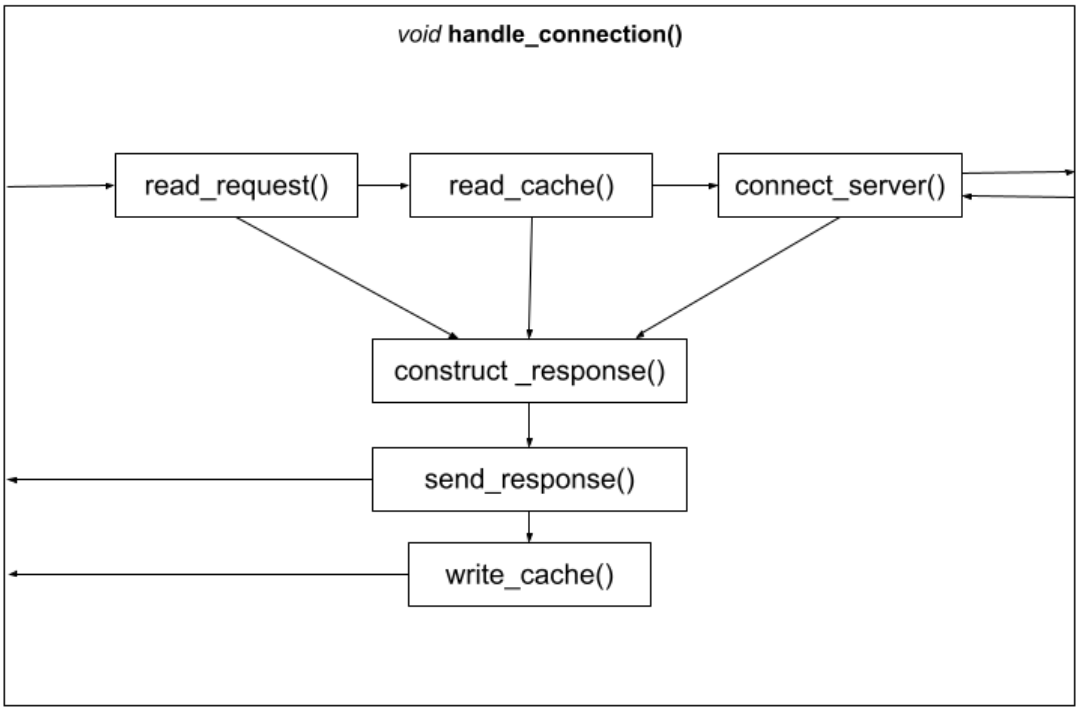
2.1.2 Connection Forwarding

The diagram above is an example of an instance of the program handling 4 client requests for 3 servers. The green-colored path on the right demonstrates a successful request that was forwarded to a working server. For this to happen, the request must be valid, the cache needs to not have an up-to-date version of the file of interest, and there is an available server to take the request. Once this happens, a valid response from the server will be fully forwarded to the client, and the file will be stored in the cache if the file length is within the allowance.

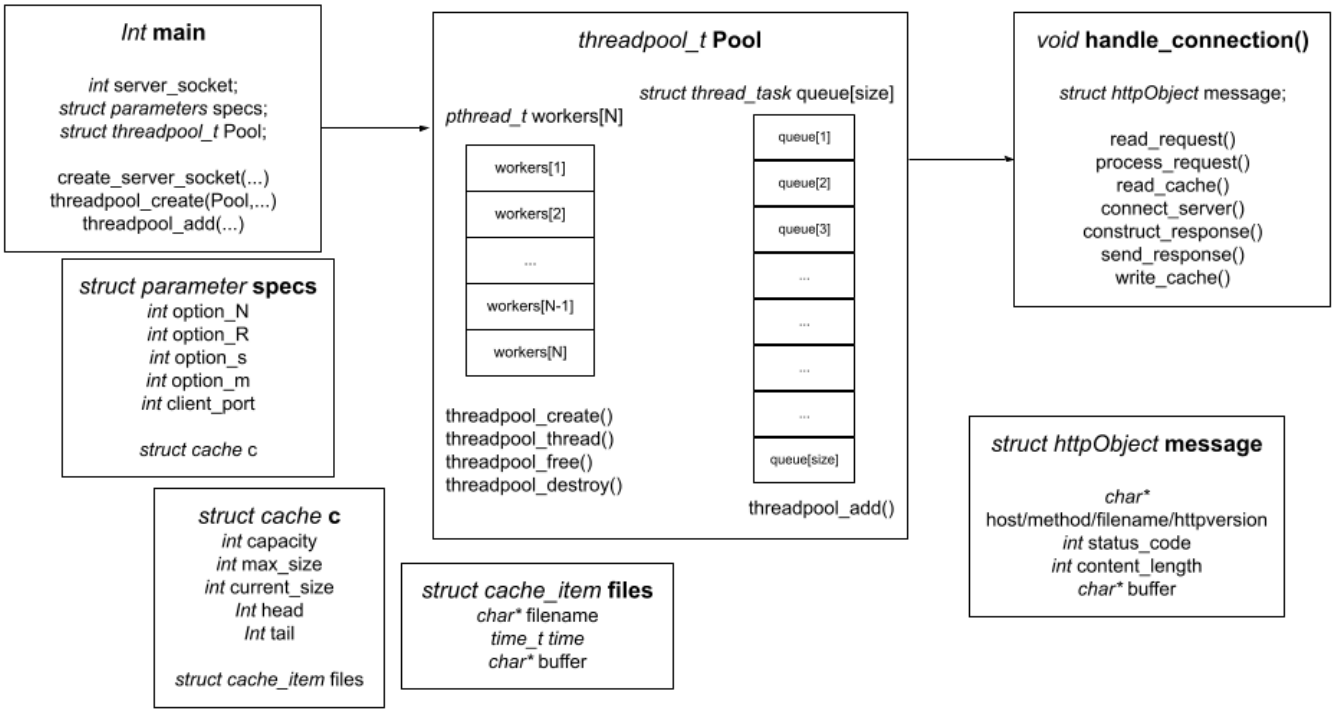
If a request is invalid, “httpproxy” will skip the servers, generate a response with an error message, and respond to the client right away. This can be shown as the path colored in red.

If a request is valid, and the file was found in the cache, “httpproxy” performs a version check. It sends a HEAD request for the file to the server to retrieve the “Last-modified” time and compares it with the current cached time. If the cached file is confirmed to be up-to-date or is newer, the program takes the content from the cache to send back to the client. This is shown as the path colored in blue. If this was not the case and the cached file is older, then the request is forwarded to the server as the green path.

If a valid request to GET a file was received and responded by the server, but the content is larger than the allowed cache size, then caching is skipped. This is shown as the path in pink.



2.1.2 Threads



This design was copied from the previous "httpserver" program, as our load balancer would essentially behave like a multi-threaded server for the clients. When the server is created, the main behaves as a dispatcher and starts the port connection. Afterwards, a *struct parameter specs* is given, to store the arguments passed in by the server launch command.

By default, the program supports up to 5 parallel connections (-N 5), performs a system-wide health check to all servers every 5 requests (-R 5), and creates a cache of 3 files (-s 3) with 1024 bytes of length (-m 1024). The host of the program can provide optional parameters to modify these values, or to turn off caching by setting either the -s or -m to zero.

The dispatcher then creates a threadpool struct named *Pool*, which contains many threads in an array of given size. From here, the dispatcher completes the setup process, and goes into a while loop to wait for client requests, and adds the task to

the queue when there is one. To run these two functions, the object *specs* will be passed in as an argument with server specifications.

The struct *Pool* contains an array of pthreads named *workers*, and an array of "struct thread_task" named queue, as it follows the "first in, first out" rule. *Pool* first unpacks *specs* to get the size of the thread array. The worker threads will be created to sleep, and wait for a signal to wake up one at a time to do a task when there exists a task in the queue unattended. In this program, that task will always be to run "handle_connection()", will always pass in *specs,* but may pass in a different socket file descriptor each time for different clients or requests.

Finally, the function *handle_connection* unpacks the *specs* to get the pointer to cache, creates the struct *message*, stores the request string in *message->buffer*, and *message* gets passed around the functions within to complete the task: read the task, do the task, generate a response message, and send it back to the client.

3.0 Data Structure

Many of these structures were re-used from the previous "httpserver" project, with the addition of structures for caching.

3.1 "struct httpObject"

The struct "httpObject" exists within "handle_connection()", and the purpose of this local object is to allow data be passed around within the many sub functions inside "handle_connection" to complete the task. Upon each thread performing a job, one unique "httpObject" will be created to pass around the data and be freed up upon completion.

A structure named "httpObject" will be created as a local structure within handle_connection() to store HTTP messages, whether the message is being sent to the server or to the client. This object parses a properly formatted HTTP message and stores them into the structure members of relevant data type and

identity. Structure members include: method, filename, http version, content length, header length, status code, header string, and content string.

3.2 “struct parameters”

The struct “parameters” is created in Dispatcher, and contains arguments of the server launch command. The arguments it holds contain the specifications of the server by the host.

This object has three major differences in behavior compared to “httpObject”. One, the arguments it stores and gets initialized with remain unchanged throughout the whole lifetime of the server. Two, rather than existing in a single scope, this object will be passed around between Dispatcher, *Pool, and* “handle_connection”. Three, the “parameter” object is shared between all worker threads, even though they will each have their own “httpObject”.

Understanding the differences between “httpObject” and “parameters” was the key to decide which objects belong in which structure.

For example, a default call for *“./localhost 1234 8080 8000”* will make this “parameter”:

option_N : 5
option_R : 5
option_s : 3
option_M : 1024

While a call with options like *“./localhost 1234 -N 7 -R 3 -s 5 -m 20488080 8000”* will result in this:

option_N : 7
option_R : 3
option_s : 5
option_M : 2048

NOTE:

There are also other objects stored in this structure, such as a client port number and a struct cache object. The port number is decided by the health check result for the server with least work, and the cache object will be explained below.

3.3 “struct cache”

This is the structure for the cache of the load balancer. It contains six arguments: an array of “struct cache_item”, max capacity, max length, current capacity, head index, and tail index. The head and tail index is necessary to create a queue behavior for the array of cache_item object. Details of “struct cache_item” is provided below.

3.4 “struct cache_item”

This object is the file stored in cache. It has a filename, last-modified time, and a content buffer.

3.5 “struct threadpool_tasks_t”

This is the structure for the queue object in the threadpool. It contains two arguments: function, and function argument. In our design, it will always store “handle_connection()”, and the arguments it must accept to finish the task: a connection file descriptor, and a “parameters” object with the server’s specifications.

3.6 Other

In addition to “httpObject” and “parameters”, there are also global string variables for system error messages, such as “500 Not Implemented”.

4.0 Component-level design

4.1 main

*int main(int argc, char *argv[]);*

Referenced: This module was partially provided by Professor Alvaro.

4.1.1 If successful, creates “struct parameters” object, for example named *specs*, to store host arguments for log file name and thread count.

4.1.2 Creates a “struct threadpool_t()” object to start the array of worker threads.

4.1.2 Run “threadpool_add()” in a while loop to always queue up a task to pool.

4.2 handle_connection

void handle_connection(int connfd, struct httpObject message);*

Referenced: This module was partially provided by Professor Alvaro.

4.2.1 Performs four tasks: read message, process request, creates response, send response, and close connection. Except for closing the connection to file descriptor, the other four tasks are done by calling the subsequent functions for each task (see 3.3-3.6).

4.3 read_http_request

void read_http_request(int connfd, struct httpObject message);*

4.3.1 When the message is interpreted as a good request, the whole string is stored in “httpObject” inside the buffer.

4.3.1.1 If not, change the status of “httpObject” to 400.

4.4 connect_server()

void connect_server(struct httpObject message, int connfd);*

4.4.1 When successful, extract the whole request message from the buffer within “httpObject”, and forward to the server.

4.4.2 Retrieves the full response from the server and stores it in the message.

4.5 construct_http_response

void construct_http_response(int connfd, struct httpObject message);*

4.5.1 Reads status code within “httpObject” and creates a response message, by putting together httpversion, status code, and the corresponding header string.

4.5.1.1 If the status code is 200, attach an additional content length data to the message, indicating that the task is completed successfully.

4.6 send_http_response

void send_http_response(int connfd, struct httpObject message)*

4.6.1 Sends header string to client. This will be the status message they receive.

4.6.1.1 If the string buffer in “httpObject” is not empty, send that to the client as well.

4.7 read_cache()

int read_cache(struct httpObject message, struct cache* c)*

4.7.1 Iterates through all cached files to search for target file

4.7.1.1 If it exists, call is_updated to check if it is updated.

4.7.1.1.1 If so, load the message with this file and return 1.

4.7.1 If any of the conditions above fails, return 0.

4.8 write_cache()

void read_cache(struct httpObject message, struct cache* c)*

4.8.1 Looks for empty cache space.

4.8.1.1 If the cache is full, dequeue the oldest file.

4.8.2 Store the file in the allocated cache space.

4.9 threadpool_create()

void threadpool_create(int tMax, int qMax, struct parameters specs)*

4.9.1 Create and initialize a new *struct threadpool_t Pool*

4.9.2 Allocate space for the size of thread array and queue array by given size.

4.9.3 Initialize the locks within *Pool.*

4.9.4 Use a for loop to create each thread in the *Pool*, and run "threadpool_thread()".

4.10 threadpool_thread()

*void * threadpool_thread(void *tPool)*

4.10.1 Runs in a while loop to remain active indefinitely.

4.10.2 Locks the thread and runs in a nested while loop while there's no tasks.

4.10.3 Ends the while loop when signaled with a task to do

4.10.4 Unlocks the mutex.

4.10.5 Performs the task.

4.11 threadpool_free()

*void * threadpool_free(struct threadpool_t *Pool)*

4.10.1 Free the queue.

4.10.2 Free the worker threads.

4.10.3 Lock and destroy all mutexes.

4.10.4 Free the pool.

4.11 threadpool_add()

*int threadpool_add(struct threadpool_t *pool, void(*function)(*void), void* args)*

4.10.1 Assign function and arguments to the top of the queue.

4.10.2 Signal the condition variable to change status when needed, to trigger worker threads to wake up.

4.12 Additional helper functions:

4.12.1 string to unassigned int

uint16_t strtouint16(char number[]);

Referenced: This module was fully provided by Professor Alvaro.

4.7.1.1 Converts string to unsigned integers.

Used in main() to get the port number from client entry.

4.12.2 create_listen_socket

int create_listen_socket(uint16_t port);

Referenced: This module was fully provided by Professor Alvaro.

4.12.2.1 Creates a socket to listen for connections.

4.12.2.1.1 If unsuccessful, return an error message.

4.12.3 clear_httpObject

void clear_httpObject(struct httpObject message);*

4.12.3.1 Reset the given “httpObject”, and empty out all data stored within.

Used in main() after each request is completed, to reuse the “httpObject” for the next message.

4.12.4 send_full

*ssize_t send_full(int fd, uint8_t *buff, ssize_t size, int filedesc);*

Referenced: This module was partially provided by Instructor Hilbert.

4.12.4.1 For a large buffer, runs send() repeatedly until the end of the string.

Used in send_http_response to send large buffers.

4.12.5 recv_full

*ssize_t recv_full(int fd, uint8_t *buff, ssize_t size);*

Referenced: This module was partially provided by Instructor Hilbert.

4.12.5.1 For a large buffer, runs recv() repeatedly until the end of the string. Designed and created in response to send_full but was not used in this project.

4.12.6 is_awake

int is_awake(pthread_t t)

4.12.6.1 Checks if a thread is awake

4.12.7 is_valid_resource_name()

int is_valid_resource_name(char name)*

4.12.6.1 Checks if a file directory is valid

4.12.8 is_valid_host()

int is_valid_host(char host)*

4.12.8.1 Checks if a host entry is valid

4.12.9 is_bad_request()

int is_bad_request(char resource, char* httpversion, char* host)*

4.12.6.1 Runs functions above (4.12.7-9) to detect bad requests.

4.12.6.1.1 Returns true if any of the tasks are true.

5.0 Restrictions, limitations, and constraints

The constraints for creating and running the httpserver are the use of a system terminal in a LINUX environment, like Ubuntu 18.04.

6.0 Testing Issues

6.1 Connection forwarding

To compare the functionality of connecting with HTTP proxy versus using a server directly, any platform or browser with access to localhost can submit trial requests to the server. In this case, a client can be created by using languages like Python or C:

Examples can be as such:

Client Program

Port: 1234

"GET /sample.txt HTTP/1.1\r\n\r\n"

Alternatively, run CURL on a terminal can work, the corresponding lines are:

Terminal

curl http://localhost:1234/sample.txt

6.1.1 Expected software response

During execution, the server may return two types of message, success code, or error code.

The status code for successful run are 200 ("OK") and 201 ("Created"). An example responding to the requests in 6.1 can be as such:

HTTP/1.1 200 OK\r\nContent-Length: 6\r\n\r\nHello!

In addition, the majority of testing will be done using the "*diff(1)*" command.

Looking at 6.1, using the first curl example in, after using the client_input.txt file to create sample.txt, running the command "*diff client_input.txt sample.txt*" should show that these two files are not different. Another example is, if we modify the second curl and generate the output from GET into a file, like such:

curl http://localhost:1234/sample.txt --output proxy_output.txt

curl http://localhost:8080/sample.txt --output server_output.txt

After running the line above, the "*diff proxy_output.txt server_output.txt*" line should also return true if proxy forwarded the messages correctly.

6.2 Cache

Before starting any tests, it is helpful to enable print statements for read_cache() and connect_server() to see if a file is retrieved from the server or the cache.

A convenient way to do so is to open the source code in a text editor, and look for "#define Debug 0" on line 22 and edit it to "define Debug 1". Doing so will activate print statements for the program behavior, but has no effect on program functionality.

To test if the program is using cache correctly, the procedure is followed as:

- Launch the server, start the proxy, and perform a client request.
- Repeat the same command and store the output in a different file..
- Run "*diff(1)*" for the cached file with the server-returned file.
- If diff returns nothing, then the files are identical, and the test is a success.

Repeat for binary and non binary files, small and large, with updated and outdated cases.

Retrieved files bigger than cache size will never be stored in cache.

Test for caching more files than the capacity allows, and make sure FIFO is enforced.

6.3 Healthcheck

As a client, run a health check with HEAD, and check that the server updates the most optimal server.

An important test was to run a health check as the first task executed when launching the server. Even when there were no prior tasks performed, a server should still be selected to perform the first tasks. If there are no servers available, the program should return 500 errors to the clients.

6.4 Multi-threading

A shell script is written to send up to 10 requests at the same time. The request is sent with a mixture of GET and non-GET requests that are expected to fail or succeed. While doing so, there are many aspects of multi-threading to test, including:

- Successful creation of right number of threads
- No race conditions when accessing files or cache.
- All tasks are completed exactly once, no more and no less.
- All tasks received the expected response body.