Design for Assignment 2: Multi-threaded HTTP server

1.0 Introduction:

1.1 Objectives

The goal is to create a HTTP server with multi-threading, so that it can support multiple requests from clients to read and write files. This project started off with a preceding program, “httpserver”, a single-threaded server that executes commands like “GET”, “PUT”, and “HEAD”. The multi-threads will be implemented on top of that, as well as adding a logging option for the server, and a health check feature for the clients to read the log history.

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 and write files using simple Linux commands.

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

Host

Like the previous project, the only tasks available for the host is to run or close the server. However, this time the user gets to decide the number of threads at work, and whether to turn on the logging feature.

The first step is to build an executable file. By opening a Linux terminal, this can be done by running the command “*gcc httpserver.c -o httpserver”*, 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 “./httpserver”, followed by other arguments like port number, thread size, and log file name. For example:

“*./httpserver 8080 -l log_file -N 4”*

Thread size and log file name must follow the “-N” and “-l” option respectively. The order of arguments is irrelevant, but it is mandatory to include a port number

for the socket. If the “-N” option is left out, the server will create 5 threads by default. If the “-l” option is left out, the server will not write any logs.

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 customer’s point of view, there are three types of commands to perform on a file:

“PUT”: Send a file from client to server.

“GET”: Fetch a file from server to client.

“HEAD”: Provide information, such as the headers, on existing files.

In a nutshell, running “HEAD” generates a similar output with running “GET”, but will not print the content in the file.

To perform a health check, run a GET command on *“healthcheck”*. Instead of getting a file named “healthcheck”, the server will return a message indicating the count of errors returned and count of tasks performed. To assure this feature behaves properly, the user is not allowed to create a file named *“healthcheck”* in the default directory, and attempting so will only result in a *“Forbidden”* error.

1.3 Software context

The project” is developed to be a customized Linux command head as a learning experience for a computer science class. It is expected to inherit the behavior of any typical running server. 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 server, such as multi-threading.

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

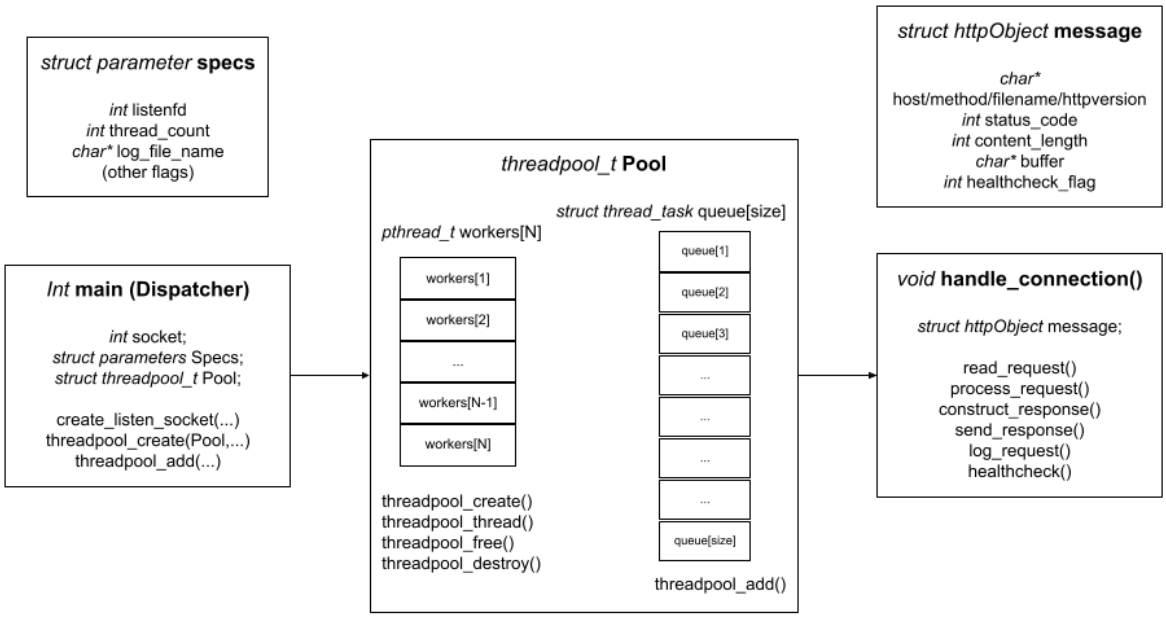
Spring 2020 discussion session code, Clark Hilbert, https://git.ucsc.edu/cse130/spring20-palvaro/cwang120/-/blob/master/asgn1/httpserver.c

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. Mainly, there are three core divisions of task handling: the dispatcher thread (main function), the threadpool (a structure with an array of threads and a queue of tasks), and handle_connection (a function that responds to client requests). In addition, two other structures play important roles to complete the design: *"struct parameter"* and *"struct httpObject"*.



2.1.1 Dispatcher

When the server is created, the dispatcher starts the port connection. Afterwards, a *struct parameter specs* is given, to store the arguments passed in by the server launch command. If given an "-l" option, Dispatcher creates a log file with the given name and directory. The dispatcher then runs creates threadpool struct named *Pool*, which contains many threads in an array of given size (or 5 by default if not given). 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.

2.1.2 Pool

The struct *Pool* is the backbone of the multi-threading behavior. It 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

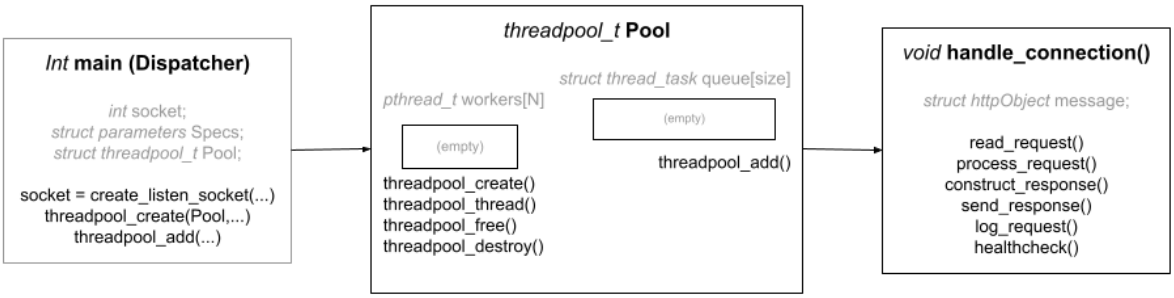
unattended. In this program, that task will always be to run "handle_connection()", will always pass in *specs,* but may pass in a different socket file descriptor each time for different clients or requests.

2.1.3 handle_connection()

Finally, the function *handle_connection.* This was created in the previous project, and only serves to interact with the client's request. It unpacks the *specs* to check for log file names, 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.

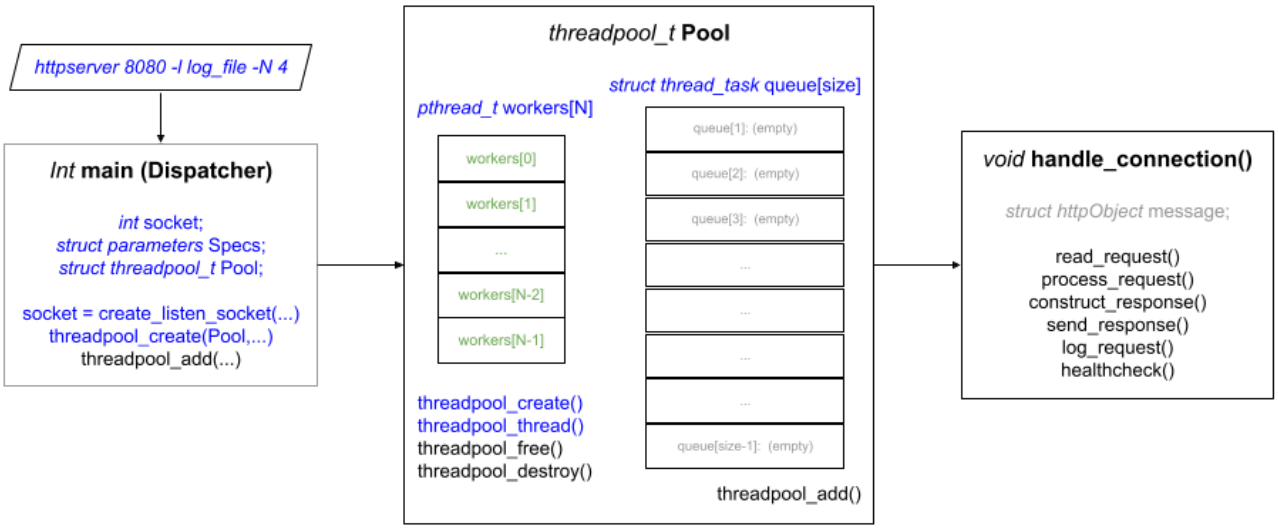
2.2 Flow chart

2.2.1 Before execution



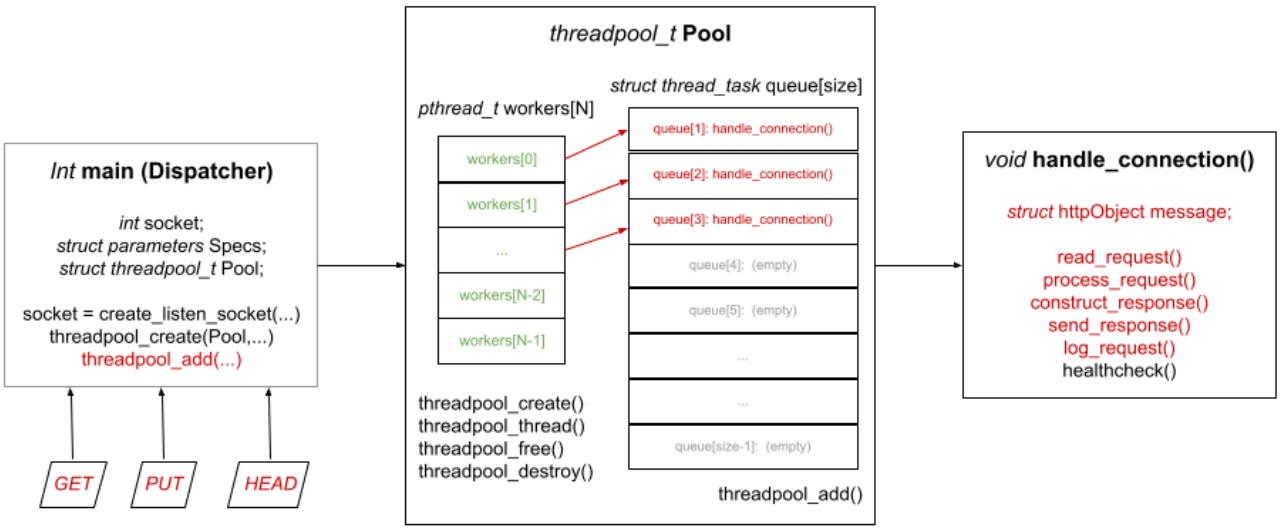
Everything is in an empty state.

2.2.2 Creates server



After being started with a command like "./httpserver 8080 -l log_file -N 6", the Dispatcher initializes the objects, passes the arguments to *specs*, then uses *specs* as the argument to run "threadpool_create" once and "threadpool_add" in a while loop. This then goes into *Pool* and creates the two arrays of worker threads and tasks. The workers are up and ready, but the queue is empty at this point.

2.2.3 Client requests



The dispatcher accepts the client's connection file descriptor, then passes it, along with *specs*, into "threadpool_add()". These arguments will be stored into queue

with the task “handle_connection”. Once this happens, one worker thread will be signaled to wake up and take the job, running “handle_connection”, creates the object *message*, and finishes the task. Once completed, the task will be removed from the queue, the worker thread is freed, and looks for more tasks before returning to sleep.

3.0 Data Structure

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 global structure 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.

For example, a client may send in the following message:

PUT /ABCDEFabcdef012345 HTTP/1.1\r\nHost: 10.0.0.5:8080\r\n

Content-Length: 3\r\n\r\nab

In this example, the message is broken down and stored into “httpObject” as such:

method:	*PUT*
filename:	*ABCDEFabcdef012345*
http version:	*HTTP/1.1*
header length:	(empty)

header:	(empty)
buffer length:	*3*
buffer:	*nab*

As the server unbundles this "httpObject" and reads the message, it finishes the tasks and replies with the following message:

HTTP/1.1 201 Created\r\nContent-Length: 8\r\n\r\nCreated\n

This message will be stored as such:

method:	*PUT*
filename:	*ABCDEFabcdef012345*
http version:	*HTTP/1.1*
header length:	*12*
header:	*201 Created*
content length:	*8*
buffer:	*Created*

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.

For example, a default call for *"./localhost 8080"* will make this "parameter":

thread_flag :	0
thread_count:	5
log_flag:	0
log_file_name:	""

While a call with options like "./localhost 8080 -N 6 -l log_file" will result in this:

thread_flag :	1

thread_count: 6

log_flag: 1

log_file_name: log_file”

NOTE:

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.

3.3 “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.

3.4 “struct task_args”

This structure takes all the arguments a thread needs to perform a function, and packs together into a single object. As mentioned above, an item in a queue is a “threadpool_task_t” struct for a thread to complete. However, for a thread to do a task, it can only accept an argument, hence the need to create a structure if that function to be executed needs more than one. In our case, it will always pack together the socket connection and the “parameters” object.

3.5 Other

In addition to “httpObject” and “parameters”, there are also global string variables for system error messages, such as “404 Not found”.

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 status of "httpObject" to 400.

4.4 process_request

void process_request(struct httpObject message);*

4.4.1 When successful, extract the whole request message from the buffer within "httpObject", parse the data, and fill in the remaining structure members, including method, filename, httpversion, status code, header length, header, buffer length, buffer.

4.4.1.1 If not, change status code to appropriate error type, and fill in the buffer with corresponding message, defined in global string variables.

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 log_request

void log_request(struct httpObject message, struct parameters* specs)*

4.7.1 Opens log file in designated directory

4.7.1.1 If it doesn't exist or is forbidden, then return the error message.

4.7.2 Use pwrite() to record the success or failure message accordingly.

4.8 health_check

void health_check(struct httpObject message, struct parameters* specs)*

4.8.1 Records tasks performed in a designated log directory.

4.8.1.1 If the content is over 1000 characters, only log the first 1000.

4.8.2 Read and count the number of successful and fail entries

4.8.2.1 If any line is corrupted, then return the error message.

4.8.3 Return the count of errors and successes recorded in the log.

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 HTTP Protocols

To compare the functionality of HTTP server, 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: 8080

"PUT /sample.txt HTTP/1.1\r\n\r\nContent-Length: 6\r\n\r\nHello!"

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

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

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

Terminal

curl -T client_input.txt http://localhost:8080/sample.txt

curl http://localhost:8080/sample.txt

curl -I http://localhost:8080/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/1.1 201 Created\r\nContent-Length: 0\r\n\r\n

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

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

The error codes include: 400 ("*Bad request*"), 403 ("*Permission denied*"), 404 ("*Not found*"), and 500 ("*Internal server error*"). They will all follow the same print format, with an example can be as such:

HTTP/1.1 404 Not found\r\nContent-Length: 0\r\n\r\n

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:8080/sample.txt --output client_output.txt

After running the line above, the "*diff client_output.txt sample.txt*" line should also return true if GET executed correctly.

6.2 Logs

To test if a log entry is correct, the procedure is followed as:

- Launch the server and perform a client request.
- Create a text file and manually enter the expected output.
- Use the Linux command to diff the log file and the manually created file
- If diff returns nothing, then the files are identical, and the test is a success.

Repeat for PUT/GET/HEAD, on binary and non binary files, small and large, with failing and successful cases.

6.3 Healthcheck

As a client, run a health check with GET/PUT/HEAD, and check that only GET is not forbidden.

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 log_file should still exist to return a "0 error 0 entry" response, rather than "File Not Found". For this reason, a log_file is created upon server launch, rather than when the health check is requested.

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/PUT/HEAD 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
- All tasks are completed exactly once, no more and no less.
- All tasks received the expected response body.