Assignment 2 Design Document

Chris Sclipei

CruzID: csclipei

CSE 130, Fall 2019

1. **Goal**

   The goal for this assignment was to essentially extend the functionality of Assignment 1 by allowing for multithreading and logging. This would be done by prompting options through the command line when running the code, which would allow for faster computations and logging of requests made by the client to the server. Options would be specified by either: -l or -N and would follow the log file specified to write to and the number of threads intended, if none specified, it defaults to 4.

2. **Assumption**

   For this assignment, I am assuming that my code from the previous assignment is sufficient enough to be run effectively once multithreading is implemented correctly. I will also assume that *curl* will handle the client side like it did in the previous assignment, so I won't need to implement it myself. Additionally, I'm expecting the user to input valid arguments and commands in order to run the code correctly. Furthermore, that multiple requests will be processed via a shell script, instead of typing the requests in manually. Responses to the requested commands should also be one of the following:

   - 200 – OK
   - 201 – Created
   - 400 – Bad Request
   - 403 – Forbidden
   - 404 – Not Found
   - 500 – Internal Server Error

   Any additional responses are outside the scope of this assignment and will not be accounted for.

3. **Approach**

For this assignment, I will start off by reusing my old assignment’s code to further extend the functionality of this code. I will essentially be using everything that was already present but be adding multithreading and logging as additional functionality.

Multithreading

Multithreading will be down by creating threads and then using a mutex to enable a thread to handle a particular request. Once that thread is finished handling the request the mutex will be freed up again to allow for the next thread to handle a request. I will implement this by parsing in the option from the command line. Whatever follows the -N option will be how many threads I create for my server. If nothing is specified, then it will default to using only 4 threads. I will need to modularize my code, however, to allow for the possibility of multithreading. I will attempt this by splitting my request calls and putting them into a sperate method, which will then be used by the threads states. I can create a *for* loop to create the number of threads specified, which will then use a *dispatcher* thread to signal other threads to run when a request is made. I can implement handling request by introducing a *queue* for the sockets. So when there is a request, it will push it to the *queue* and then go to the *dispatcher* thread, which will then pop it from the *queue* and have a thread run that request.

Logging

Logging will be specified if and -l is satisfied with a corresponding log file to be written to. This will have the requests along with the length of the files and the content, converted to hex, be written to a file on the server side. I will proceed to implement this by introducing a flag option, and if the option is parsed through the command line then it will create a writeable file with the specified file name. Then whenever a thread finishes processing a request, depending on the response code, the logger will log the details of the request process to the log file. However, since this may be down concurrently, I will have to implement a mutex in order to reserve privileges for one thread at a time, since otherwise, it could overwrite existing content. Then if the content is to be printed into the log file, I can *sprint()* to modify the content and convert it to hex in order to print it correctly. And each subsequent, request should be printed until the server is closed.

4. **Pseudocode**

Here’s some pseudocode of the proposed code detailed above. For the sake of simplicity and to avoid redundancy, I will omit the code that was used in the previous assignment since that will be in its own method and called by the threads to execute.

```
for 0 < number of sockets
    pthread_create(call dispatch thread)
while true
    for every request made -> push socket to queue
end while


void dispatch thread ()

    while true

        mutex.down()

        if Q.size() > 0 then
            run thread
            Q.pop()
        end if
        mutex.up()

void logging ()

    if flag = 1 then
        open file with crate and write permissions
        mutex.down(write)
        use sprintf() to convert content into hex
        use pwrite() to write to file
        mutex.up(write)
```

*Logging occurs only after each request has been handled and validated with the appropriate response code