Assignment 3 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 and 2 by implementing caching to allow for faster client-server interactions. This would be done when the user prompts caching by inputting the option *-c* in the command line. Similarly, logging is also prompted when the user inputs the option *-l* with a specified log file name. The log file, if caching is enabled, will keep track of requests that are either in the cache or not, otherwise it will print like the log file in Assignment 2 did.

2. **Assumption**

   For this assignment, I am assuming that my code from the previous assignment is sufficient enough to be run effectively once caching 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. I am also assuming that if there's nothing in the cache originally, that the program will write to both the cache and the disk. Furthermore, I am also assuming that GET requests won't change the outcome of the cache unless it's the first time prompted, whereas PUT will need to update cache requests. 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 caching to improve latency.

Caching

Caching will be handled when the user prompts the option *-c* in the command line. If caching is enabled, then there will be 3 vectors: one for the file name, one for the content, and one for the content length. The vector for the content length is intended to differentiate PUT request in order to update the cache accordingly. Each of these vectors will have a *reserved* size of 4, in order to satisfy the maximum size condition, since only 4 requests will be cached at a time. At the start of the program, when the vectors are empty and a request is made, the file name along with the content and content will be pushed into the vectors simultaneously, in order to keep track of the pairings. Then if another request is made, it will be checked against the requests already in the cache. If the same request is in the cache with the same file name and content length, it will take the file name and content and from the cache and write it to the disk. It will stay in the cache until it is updated by a PUT request or if there are already 4 requests in the cache and there is another new request. I will then implement FIFO in order to preserve priority and allow for further caching of different requests.

Logging

The only difference logging will have in the assignment compared to the previous one is accounting for whether or not caching is active. If caching is active then the log file will keep track of whether or not the requests have been cached. If they haven't been cached then the log file will log the content in hex as well as append *[was not in cache]* to its header. However, if the request was in the cache then the log file will print out just the header with a length of 0 and append *[was in cache]*. This can be done by using *sprintf()* to account for the different scenarios and whether or not caching is enabled.

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.

if *caching is enabled* **then**

if *vector is empty()* **then**

push file name, content, and content length to specified vectors

if *vector is full* **then**

remove the first element in all 3 vectors to ensure consistency

if *request is in cache* **then**

write from cache to disk the contents that were saved to the vectors

for *each element in the vectors* **do**

check to see if elements match for a ***put*** *request*

if *true* **then**

delete the original request and update the cache with the new request

void logging ()

if flag = 1 then

open file with crate and write permissions

use sprintf() to append caching options

use sprintf() to convert content into hex and pad the beginning bytes

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