Design Document: Implementing PATCH requests

Robert Hu

CruzID: ryhu

1 Goals

The goal of this programming assignment is to modify the original HTTP server with the functionality of being able to utilize aliases. To accomplish this, we need to implement the ability to process PATCH requests using alias. We will keep a persistent hash table and mapped keys that are new names or aliases with values for the associated names, whether they be existing names, aliases, or resource names. The code needs to maintain being thread safe so there needs to be more locks/unlocks to prevent hash table collisions.

2 Design Changes - PATCH and Alias

2.1 Importants notes to keep in mind

- Existing file names can be either an http name that follows the previous pattern or an already existing alias
- New names can be any number of allowable characters
- Aliases can chain to other aliases
- New options to handle for initial server operation
    - -a mapping_file
        - If no mapping file is specified, exit with a usage error
        - if the mapping file does not exist, then create a new one with no aliases
        - Since the mapping file must be persistent, if the file exists, the server and any requests must use those aliases present in the file already
- The new additions need to be thread safe as well, following the previous designs of assignment 2

2.2 Hash table and mapping

- **NOTE:** Original hash table functionality adopted from previous CMPS 12B/M class, taught by Prof. Tantalo
- Create initial hash table with room for 8000 pairs of keys and values for aliases and the corresponding resource name
    - **Note:** make the hash table hold room for more than 8000 pairs since we are assuming that we are only handling up to 8000 pairs and this helps avoid collisions
- Each key and value pair has a max size of 128 bytes or characters including the null terminating char

- Allocate each “bucket” of the mapping file 128 bytes to allow the use of pread() and pwrite() easier handling
    - This way the offset in pread and pwrite is always a multiple of 128 bytes
- Insert function for storing key/value pair into hash table
    - If a key already exists, update the value associated with that key to the new value in the ALIAS request header
    - If a key doesn’t exist and there is still “an empty bucket” available in the mapping file, then insert the pair into an empty 128 byte bucket.
    - If there is already 8000 pairs filled, return an error
- Update mapping file after access is over
    - write changes to mapping file as aliases are changed

2.3 Changes to original design to allow for Patch implementation

- Addition to if get else put statement to allow for patch requests
    - If (get) ...
    - else if (put) ...
    - else ... //Alias
- Changes to the GET request handling to allow for the processing of aliases
    - The server follows an alias until it reaches an object or nothing at all
        - If there is no object corresponding to the alias, return a *404 Not Found* error header. Content length is then appropriately set to 0
        - Aliases can chain to other aliases; the resource name isn’t necessarily an http name
        - Keep resolving aliases until the current name is 27 chars long and follows the http name requirements
        - Example:
            - lock() mapping file
            - while alias is not http name
            - // resolve alias
            - if new name is http name the break out of loop
            - otherwise continue resolution

2.4 Thread Safety

- Since the hash table is available to all threads as a shared resource
    - New mutex, hash_mutex, for handling access to hash table file
    - Lock and unlock mutex when processing aliases
        - This prevents multiple threads from accessing the mapping file at the same time
        - Also this keeps multiple threads from writing to the same “bucket”

3 Original Design - Multithreading and logging

3.1 Original design

- Parsing using strtok to separate received messages and realloc to allocate memory
- Send/recv and write/read to input from and output to data and/or failures to sockets

3.2 Multithreading

- Check for the *-N* flag
    - If present, set the number of threads to the number that follows the flag
    - Otherwise set the number to the default of 4
- There needs to be the same number of worker threads as are specified by the number of threads. Using *pthread_create()* for both creating the n worker threads as well as the single dispatcher thread using
    - *pthread_create(&dispatcher_thread , NULL, dispatcher, (void*) &arguments)*
    - *pthread_create(&worker_thread[i] , NULL, worker, (void*) &arguments)* inside of a for loop from between 0 and N.
- Synchronization needs to be there to modulate and control access between shared resources and access to the entries in the listen queue that are to be released by accept(). Code to use
    - *pthread_mutex_t* to create the initial locks that will be used
    - *pthread_mutex_init* to initialize the values to the locks that are used
    - *pthread_mutex_lock* and *pthread_mutex_unlock*
        - This code will occur around accesses to the log file, if it exists
        - Otherwise, the unlocks and locks will maintain access to the available resources for requests
- New functions *dispatcher()* and *worker()* to handle what each thread will do
    - *Dispatcher()* will listen for new connections and distribute each new request to a different worker thread and lock and unlock appropriately
    - *Worker()* will wait and sleep until the dispatcher sends them a request to do. After processing the request, the worker thread will return to sleep.

3.3 Logging

- Check for the *-l* flag
    - If it exists, then open/create a logging file that is specified in the command
    - Otherwise no logging will be done
- Logging will need to handle failed cases as well by logging a Fail along with the error response code that follows
- In order for Logging to be done, only one thread may access the log at a time. This can be achieved by using a *pthread_mutex_t* to lock and unlock before and after the log file is accessed by a thread. This will prevent other threads from accessing the log file while another file is actively logging.
- Format of logs

- Success: PUT abcdefghij0123456789abcdefg length 36 00000000 65 68 6c 6c 3b 6f 68 20 6c 65 6f 6c 61 20 61 67 6e 69 20 3b 00000020 65 68 6c 6c 20 6f 54 28 65 68 43 20 72 61 29 73
    - length is Content Length
- Fail: FAIL: GET abcd HTTP/1.1 --- response 400\n