Design Doc for a Multithreading Server with Logging

Shahar Dubiner,
sdubiner@ucsc.edu

Objective

Based on the http-server written in assignment 1, design and implement a multithreading server in c, using pthreads. The server should be able to support several clients, each with its own thread. The server will also include logging, keeping all requests, responses, and data records (as hex). The supported requests are HEAD, GET, PUT as before, with the additional health-check request that provides the number of requests handled and the number of error responses.
Usage: ./httpserver [<port_number>] [-N <number_of_threads>] [-l <path_to_log_file>]
All parameters are optional, and they can appear in any order.
Default and Assumptions:

- Number of threads = 4.
- If a log-file is provided, it is truncated (no append writes).
- "healthcheck" is never treated as a name of an actual file. HEAD/PUT /healthcheck should return 403 (even if not logging). GET /healthcheck should return 404 if not logging or the statistics of error-responses and recorded requests if logging.

Multithreading Considerations

Having several threads enables handling several requests at the same time. So, if a request to GET or PUT a large file arrives, other threads can still handle additional incoming requests. The downside of having multiple threads is that they use more memory, are harder to implement, and can also use more CPU, when many requests are coming in. Thus, it makes sense to limit the number of threads. The default is set to 4, but any positive number can be used.

When using several threads, I need to make sure only one thread is handling each coming request. Also, if logging data, I need to make sure the logged data makes sense and does not end up mixed or interleaved.

Multiple Requests

I can think of three approaches to handling multiple requests with a limited number of threads (pool of workers):

- Have a dispatcher (main) place requests (client_sockd) in a queue and threads access the queue using lock/unlock mutex mechanism. Pro: no need to rely on the unix limited

mechanism to handle a queue of requests. Con: somewhat complicated code and more memory for the queue.

- Keep for each thread (worker) a condition (pthread_cond_t) and a client_sockd (default is -1). Have the main serve as a dispatcher that assigns requests to threads and wakes the thread up (with pthread_cond_signal of the worker's condition). Con: if too many requests arrive, the workers would not be able to cope and handle all of them. Pro: I can rely on the unix mechanism to keep a queue of requests and have simpler code that uses less memory. Also, it would be harder to crash the server.
- Following the previous idea, there is no real need for the main to serve as a dispatcher or for all the cond variables. Simply have threads grab hold of the role of a server (using pthread_mutex_lock and unlock). When a thread "becomes the server", it can accept a request and in effect contact a client. Then the server role is released to be locked by another thread. Con: as before, if the number of requests is very large, the server will not be able to handle them. This basically depends on the backlog argument used in listen(server_sockd, backlog), which amounts to the maximal length of the queue unix supports. Pro: simplest logic, less memory, hard to crash the server.

At first I implemented the third option described above (taking advantage of the unix queue). However, when I failed some of the tests I decided that it might be the culprit and so I decided to take control of the queue myself. Thus, I ended up implementing the first approach.

Logging

I can easily take care of several workers accessing the same file and writing at different offsets using pwrite(). I will need to use mutex lock and unlock to make sure each thread uses their own offset in a consistent manner. In order to achieve that, I will use a global variable log_offset. When a worker prepares to write to the log-file (assuming -l was provided), it first needs to calculate the total number of bytes it will need (based on the command and the content_length). It then locks the mutex to access the global log_offset variable, keeps that offset as a position to write in the log-file, and increases the global log_offset by the size it intends to use. This should be straightforward to implement as far as threading is concerned and only requires careful consideration of the various situations regarding commands, errors, and relative offsets between size of arriving buffer and length of lines in the logfile.

Data Structure

Each thread has a structure with relevant information about the current request handled, similar to the one used in assignment 1. The main difference is that there are several such structures (num_threads). Also, each thread-structure has the offset needed when writing to the log-file, and the number of bytes already written to the log-file for the message.

Additionally, we hold a queue of the requests (first in first out). The queue holds pointers to client_sockd. If the queue is empty, the dequeue method returns null, otherwise it returns a

pointer to an actual client. Both enqueue and dequeue are done within mutex_lock and mutex_unlock.

Flow

main():

- Parse the command line (using getopt) and set port, num_threads, and log_file.
- Create a server (as in assignment 1).
- Open a log-file, if desired (using open()).
- Create a pool of N threads, each with httpObject that includes a buffer and additional information regarding a client's request and logging related information.
- Repeatedly wait for client requests. Once a request arrives, enqueue() it safely using mutex_lock and mutex_unlock and alert workers using a condition (pthread_cond_signal) that shows that the queue was updated.
  The reason the condition is needed is that otherwise the workers will waste CPU checking the queue for new requests.

thread[i]:

- Each thread starts in a thread_function where it sets its httpObject data and goes into an infinite loop.
- In the loop, the thread repeatedly
  - Check the queue to see if there are requests (dequeue).
  - If there are no requests, wait for a condition signal (pthread_cond_wait) and once awoken check the queue again.
    The reason I start by dequeuing without a raised signal from main is that if many clients make requests during a short period of time, then not enough condition signals would be raised for all of them.
  - If a new request was found, the thread calls the process_request method, which is based on the method used to process a request in assignment 1.

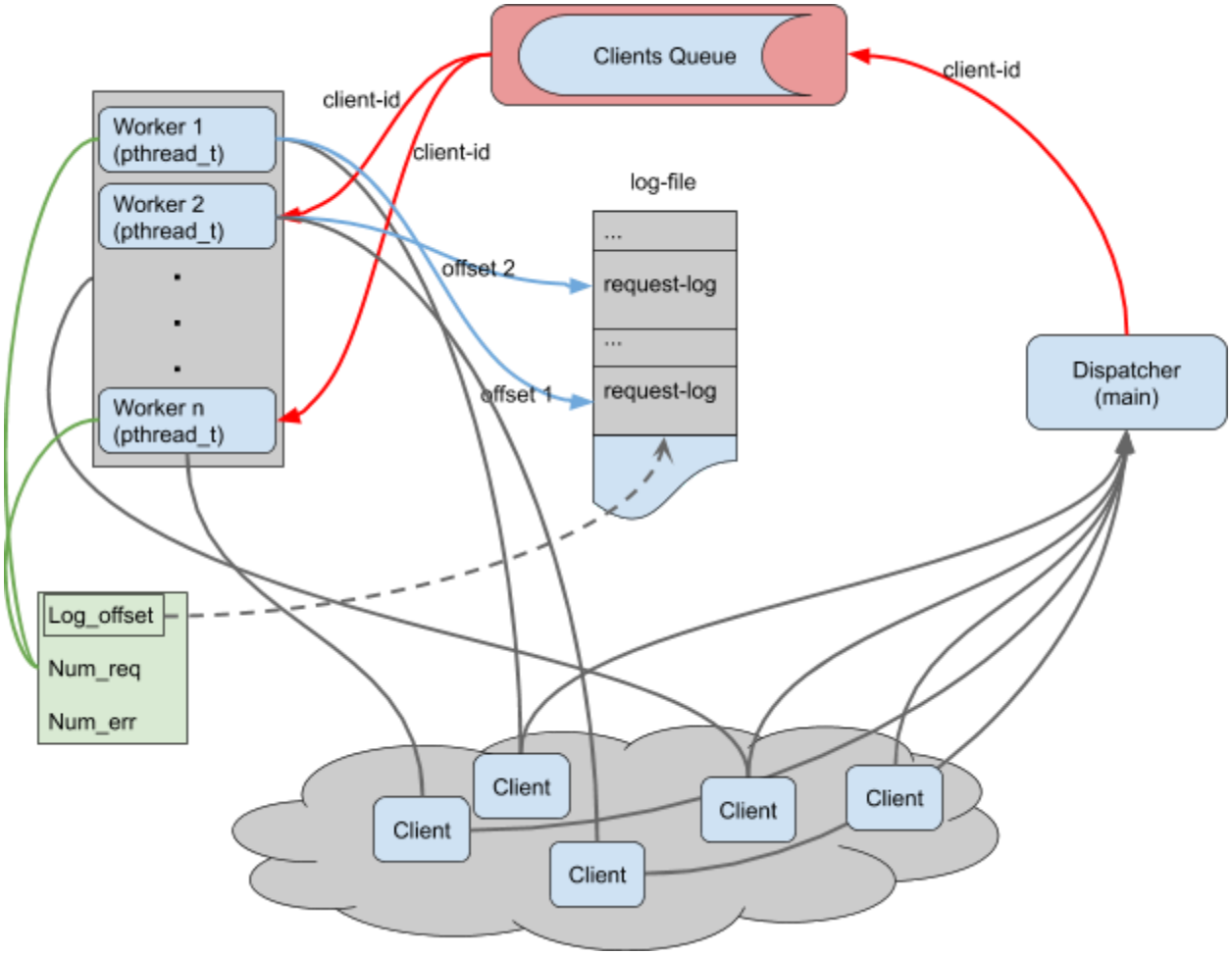
logging:

- When we first need to log something for a specific request, calculate the total length of the logged data, use mutex lock to get the global log-offset variable, as described above, and increase it by the calculated size.
- During process_request(), whenever we report an error, send a reply, recv or send a buffer of data, prepare a buffer with data to be logged, once the buffer is full, use pwrite(..., offset) to update the log with the request, error, or data passing between the server and client.

health check:

- Before replying to a "GET /healthcheck" request, assuming logging is done, use mutex lock to access global variables holding the number of errors and number of replies logged so far.

- Once the current values of errors and logged requests are known, unlock the access to the variables, so they can be updated.
- Each handled request updates the number of errors and number of replies logged so far, but only once it is done logging the request and/or error.



We use two mutex variables: one (red) controls access to the clients-queue, and the other (green) controls access to logging variables.

- The dispatcher accepts clients and enqueues their ids.
- The workers dequeue client-ids, and then communicate with their current clients to send data and responses.
- When logging is enabled, workers also get offsets to the log file and update the log offset based on the amount of data they intend to write (using the green mutex). The workers write to the log files independently of each other using their offsets (blue arrows). When they are done handling a request and sending a response, they also update the number of requests and the number of errors (safely with the green mutex).

Questions:

1. How does a thread know how much space it would need, without knowing if there would be an error in the process of handling the request (such as connection lost)?

⇒ I assume it reserves the maximal needed space (the exact space, if all goes well). If, for example, bytes stop coming in, the log would have some missing data.