WRITEUP

Testing: I performed a whole system testing by running the shell.sh script that was provided to us on the lab. I worked on multithreading first, and then tested multithreading. Then I worked on logging, and testing logging separately. Finally I tested them both together. I have my Get requests working completely fine for both multithreaded requests as well as with a properly functioning log. I was unable to get multithreading completely working for PUT, and for the PUT statement’s log I wasn’t able to split each line with 20 bytes and add the leading zeroes. I tested this with multiple lines of code.

• Using either your HTTP client or curl and your original HTTP server from Assignment 1, do the following:

• Place four different large files on the server. This can be done simply by copying the files to the server's directory, and then running the server. The files should be around 4MiB long.

• Start httpserver.

• Start four separate instances of the client at the same time, one GETting each of the files and measure (using time(1)) how long it takes to get the files. Perhaps the best way to do this is to write a simple shell script (command file) that starts four copies of the client program in the background, by using & at the end.

• Repeat the same experiment after you implement multi-threading. Is there any difference in performance?

• What is likely to be the bottleneck in your system? How much concurrency is available in various parts, such as dispatch, worker, logging? Can you increase concurrency in any of these areas and, if so, how?

After Starting four separate instances of the client at the same time and using the GET command on each of the files and then implementing multi-threading, I can tell the performance is a lot better with the multithreading. The multithreaded server is more efficient as the real clock time is slower than single threaded because multithreading allows the server to process multiple client requests at the same time. For a single threaded server it would have to wait for the entire process to finish before accepting a new request. The bottleneck in this would be the critical region/file we're calling. This is when multiple threads are trying to access a shared variable and the variable is occupied and required for another thread to process. The dispatch and worker help with the control and flow of multithreading. The less critical region there is, the higher the concurrency will be.