# Report on myhttpd – A Primitive Webserver

## CS 537 3/4/2017 Steven Clark

The architecture chosen for this primitive webserver was a multi-threaded server with each thread handling a single HTTP request. The main function, which listens on the main port and dispatches worker threads to handle client requests, was originally built entirely from the example echo server given in class. It was upgraded to choose its own port if not provided one and to detach threads so they don’t need to be joined. Most of the imported code was not rewritten as it was dominated by function signatures and just changing variable names seemed a waste of time. The majority of the work was isolated to a handler function that was called as worker threads which processed and serviced each request. This service thread (svcthread()) was aided by two helper functions, one that managed the socket and a buffer and provided tokens called filesep() which ensured input handling stayed consistent, the other output an appropriate message and response to the client for a given HTTP error code.

The service thread would read and validate the first, second, and third tokens from the socket as the HTTP request method, entity, and protocol version. After that it would continue to pull tokens from the socket, potentially recording key-value pairs (only Host was recorded but not used, but the architecture was meant to handle more if needed), until enough empty strings were returned as tokens to indicate that an empty line had been read. After that it progressed to responding to the request. If the file name indicated by the entity existed the fixed portions of the response and response header were printed, otherwise a 404 error response would be set and the error reporter called. After the fixed portions of the header, the file statistics associated with a read entity would be printed for GET and HEAD methods, including Content-Length, Last-Modified, and Content-Type if the file extension of the entity indicated it was a JPEG. An empty line was then printed. For GET requests, the contents of the entity were read from file and written to the socket in pieces. For POST requests, the contents of the socket would be read and written to the entity file in pieces until the socket closed. The socket would then be closed if there was no error, and no post, both of those cases closing the socket themselves.

Performance testing was accomplished through the unix application wget, and compliance through Firefox and Chrome web browsers. An index.html was placed in the project directory that referenced 5 test JPEG images downloaded from the Internet (from a w3 test of baseline JPEG handling <https://www.w3.org/MarkUp/Test/xhtml-print/20050519/tests/A_2_1-BF-01.htm> , and a comparison of small JPEG and JPEG2000 files <http://www.fnordware.com/j2k/jp2samples.html>) of various sizes and color-formats also located in the project directory. Initial testing revealed that the times provided by wget and the unix time program (not the bash time builtin), being based on the userspace time system, had a precision of only 10ms (there being 100 jiffies in a second on most linux systems). Transfers, as a result where in the 10s of megabytes a second and times were so short as to be useless for comparison. Additionaly no access to a normal HTTP server to compare against was available on campus.

To solve these problems a high-latency server was created. To do this an Amazon EC2 t2.micro instance was spun up in Mumbai, the lighttpd server installed, and the project and test-site copied over. With both servers running the test page and all its files were requested using “wget –resursive –delete-after <URL>” 10 times with each server and the statistics from the last two lines of each run were recorded. The output of wget confirmed it was using a persistent connection with lighttpd. The persistent connection, as well as whatever else it did differently, resulted in lighthttpd performing significantly better than myhttpd did. Lighttpd gave a mean of 286.94KB/s with few outliers, to the 175.71 KB/s mean of myhttpd which swung wildly as low as 70KB/s and as high as 300KB/s. Latency was also lower with a mean “total wall clock” time of 14.24s (10.27s with one wild outlier removed) while myhttpd had a mean of 24.2s (in two populations one of about 17s the other of about 36s). The performance disadvantage of myhttpd with regard to latency was even more severe than with bandwidth.