CS118 Lab 1 Report

Aditya Raju UID: 104407651
Srikanth Ashtalakshmi V Rajendran UID: 804478782

Specific instructions to build and run our code:

To build our project you simply need to enter “make”. The executables created are web-client, web-server, web-client1_1, and web-server1_1.

The executables web-client and web-server (multithreaded server) support HTTP 1.0. To test extra credit, read next paragraph.

The executables web-client1_1 and web-server1_1 (asynchronous server using select()) contain all the extra credit. They implement HTTP 1.1 and support pipelining and timeout.

High level description of our server and client:

Description of the server (HTTP 1.0 Version):

The multi-threaded server takes in arguments for host-name, port number, and working directory. Afterwards, it creates a socket and binds it. Then, a listening thread is created, and for every accepted connection, a new thread is formed. In this thread, the server “serves the client”. The server receives bytes until the server detects a “\r\n\r\n”. It then processes the request, and sends a response with the status 200 OK if the file is found and a payload is attached. It sends a response 404 Not found if the file does not exist in the working directory and a 400 Bad request if there is some syntax error in the request. The server then closes the connection.

Description of client (HTTP 1.0 Version):

The server takes in one URL as an argument. It parses the URL and finds the file name wanted and the host name to get the file from(the server). It sends the connect request to the server, and after that is accepted, forms a HTTP request for the file. The request is sent to the server, and then the response is received. If there is a content-length, it receives until it detects that all the bytes have come in, and if not, it receives until the server closes the connection. Then the payload is stored in a file under the same name as the requested file.

Description of server (HTTP 1.1 Version):

This server takes in the exact same arguments as the HTTP 1.0 server, and is implemented using asynchronous sockets and using select(), allowing multiple requests from the same client

and requests from other clients alike to be served in a single instance of a HTTP connection. The server accepts pipelined requests from any active client socket and serves these requests on the go. The server closes all connections after a fixed time interval (15 seconds in our case) if none of the sockets receive input from their respective clients (timeout).

Description of client (HTTP 1.1 Version):

This client takes in multiple URLs as arguments. The URLs may have different host names and port numbers. The client creates HTTP request messages pertaining to all URLs and sends pipelined requests to each server. The client then appropriately identifies responses from the server(s) and creates a file for each set of payload received. If the client does not receive a response from the server for a fixed time interval, then the client closes (the client times out after 15 seconds in our case).

Problems we ran into and how we solved them:

1. Identifying the content length as the response was being received. We solved this by continuing to poll the data and checking if the content-length field was found.
2. Ability to support multiple connections in a server. We fixed this by creating multiple threads, one for listening, and one for serving a specific client after the connection is accepted.
3. Figuring out how to support no content-length received in HTTP 1.0. We solved this by reading Piazza and finding out that recv will return 0 if the connection is closed by the server.
4. Learning how to use select() so we could do the extra credit and support HTTP 1.1. We went through tutorials online and the sample TA code and were able to figure it out.
5. When we were pipelining requests, the responses were coming back in the same buffer, so we had “multiple” content length fields (and responses) in one response. We fixed this by making the buffer size much smaller.
6. A big problem we had was the ability to pipeline requests to multiple servers. We solved this problem by creating a hashmap of hostname/port numbers to the file names requested, and then pipelined requests for each hostname.

How we tested our code:

To test our code we usually ran our client against our server. We also tested our client against servers like Google and Yahoo which usually responded with 404 errors. By using the client against these established servers, we were able to validate our client, and therefore validate our server.

We started with the simple testing of HTTP 1.0 GET from our client to our server. We tested that the payload length was correct and it was extracting the payload correctly, and tested the 404 and 400 errors. Besides the testing of the simple GET request, we tested our multi-threaded

server for ability to handle multiple concurrent clients. To do so, we made the server hang after accepting connections. We could therefore see multiple clients connecting to the server. We then let the server continue, and saw the clients get served. This validated that the server could handle multiple connections.

Because the HTTP 1.1 versions of the our client and server are built upon the code that we used for HTTP 1.0, our test cases moved from just testing simple request and response resolution to being able to support timeout and pipelining. We checked for timeout by simply not sending data to the server after accepting the connection, and vica versa from testing it with the client. We then tested that even after it timed out, that the connection could be reestablished. To test pipelining, we sent multiple requests to a server and saw that the server could handle these requests and send responses back, and our client could handle the multiple responses. Our last test case was to check if the client could pipeline requests to multiple servers. We opened multiple localhost servers with different port numbers and saw that they were both connected to and both sent data back.

Contributions of each team member:

Srikanth: Worked on mostly the client side operations and creating a select server
Aditya: Worked on the server side operations and URL parser
Both: HTTPMessage and children classes