CS118 Project 1 (HTTP Client/Server)

HIGH LEVEL DESIGN

Client: create a request object based on the command line URL argument, set appropriate host and port based on input. Use getaddrinfo() to get the available ip's (we only take the first one). Attempt to connect to the server using the port and ip address info in the serverAddr struct. Upon successful connection, we start reading in data into a buffer using a while loop until all the data is read. Depending on the status code, it will either save the file or display an error. A timer was added to make sure that the client is not waiting too long. After 30 seconds, if the client is not done receiving the file, it will close the connection.

Server: The HTTP server works by first creating a socket, then binds that socket to a port. It then listens on that port for any incoming connections. If it received a connection request, it accepts thta and holds onto the file descriptor linked to the connected socket. This describes the processes of TCP connection between the server and a client. After this connection is established, the server continues to receive until it finds the end of an HTTP request header denoted by '\r\n\r\n'. It then looks to see if it has the file being asked for. If it does it then creates a HTTPResponse object which contains the request header, and then the payload (requested file) is appended to the end. This message is then sent to the client. After the server finishes sending, it closes the connection with the client.

There are two implementations of the server which deal with concurrency in different ways. One does so synchronously through multithreading, while the other uses select() to achieve asynchronous listening and send/recv. In the threaded server, each new client is placed onto a new thread, allowing the server to continue to receieve connection requests.

Both of these concurrent servers are able to connect to many clients concurrently, however when sending and receiving HTTP messages, later clients have to wait until the requests of previous clients are satisfied.

HTTP Message: The HTTPMessage class along with subclasses HTTPRequest and HTTPResponse allow the client and server to make objects from HTTP messages and also to make HTTP messages from objects. Specifically, HTTPRequest objects can be built from input URLs or input messages, and an HTTP request message can be built from a HTTPRequest object. HTTPResponse objects can be built from input messages, and an HTTP response message can be built from a HTTPResponse object.

Overall: The three main parts of the project are Client, Server, and HTTP Message object. Everything about HTTP messages is abstracted by the object. HTTP Request and HTTP Response objects extend the parent class, and are built specifically for the needs of the Client and Server classes respectively. Client class can create an HTTP Response object by passing in a url, and get

the connection the server set up with the needed information. Once the client wants to send a request, it calls the BuildRequest() function of the HTTP Request object that constructs a well formatted character vector in the form of an http request message containing all the relevant information within in the object. The Server class, when established connection with a client, will listen on a socket to read in the character vector. It then calls the responseToObject() function, which converts a request formatted string into an HTTP Response object. From this object, if it is a valid http request, then it will load the requested file into a character vector, and send it back to the client through the connection. Once the response is received, the file will be saved if status successful, otherwise will show received error.

PROBLEMS YOU RAN INTO AND HOW YOU SOLVED THEM

Client: Had a problem correctly interpreting the URL passed in, and the ip return would always say it is not found. Turns out that the parsing method for our HTTP Message was incorrect, and we were pointing to a temporary copy of memory. This meant that by the time the server ip was needed, it was already gone from memory. Fixed this by saving a copy of the host as a string, then it worked correctly. Had a problem with the client timeout asynchronous thread that said it was terminated before actively being explicity called. Turns out need to call detach on thread for it to run asynchrounously.

Server: When forking, we failed to close the newly connected client's file descriptor in the parent fork, causing the first message send to be received by the parent branch. ended up refactoring the code in the parent branch to consist soely of closing that file descriptor. There was also an issue with reading from the client socket until a certain delimiter, for ex: '\r\n\r\n'. The hard part was understanding how to use recv to read in only a specific part of the message. After figuring out that it is possible to use recv byte by byte, the parsing of the incoming request was much easier. We first implemented the transfer using a string, and this worked fine for text based files like html. However, it did not work for transferring images for example, so we changed our implementation to use a vector of chars instead.

We also came into troubles when a header label was incorrectly typed. What happened as a result was that when requesting from the server, nothing would be sent besides the header. We solved it by figuring this typo out and correcting the spelling. Another issue was that when we received a bad request, the server would shutdown. We solved this by not exiting on a bad request but instead returning from the thread, and in the case of the asynchronous server, closing the connection to the client, removing it from the file descriptor set, and then continuing to the next iteration of the accept loop.

HTTP Message: Error checking was the main issue. Parsing and extracting the data fields wasn't bad when we were expecting correct and valid HTTP requests, but once we had to deal with bad requests, things got a little bit more difficult, because there are so many ways a request could go wrong. Right now, our server checks that the method was GET, and that the http version is either 1.0, 1.1, or 2.0. It sends the bad request message to any other method and any other version

number. We also check for the formatting of the requests, so that strings not in the specified format will be rejected. We accounted for this by adding loops to skip white space (because multiple spaces are apparently still valid), by adding checks in a few places to make sure that the message doesn't end prematurely or have carriage returns and new lines at invalid places, and by creating a new member method isValid() that allows the client and server to check whether or not an object was successfully created from a valid http message. Also, we had to check that the method was indeed GET, and that the version was specifically 1.0, 1.1, or 2.0.

Overall: Had to refactor our code to use a vector of char when sending the payload of the message, as binary data is improperly interpreted when put into strings, specifically because of the escape character '\'. This leads chars like '\0' to be misrepresented when sending the file. Other issues that came up involved determining what headers were necessary and the correct way to tell if a request header was valid or not. We solved them by focusing on the most important and frequently uses cases. Thankfully most problems centered around typos and not gross misinterpretation of how to create a TCP connection and send HTTP messages between client and server.

ADDITIONAL INSTRUCTIONS TO BUILD YOUR PROJECT

Added web-server-async to Makefile

HOW YOU TESTED YOUR CODE AND WHY

Client: Made requests to different websites, tested receiving large files, both text and image (binary) files. To test timeout, set the timout length to a small value, to make sure that if time limit exceeded, would exit gracefully. Send invalid requests, to make sure it could handle reading errors and displaying them.

Server: Tested communication with the client to see that the header was being properly created (headers), and transmitted. Tried to connect to it with multiple clients and simultaneously download a large file. We saw that they were both receiving concurrently.

HTTP Message: Gave it a few random urls to which it would create the Http request and response. We then looked to make sure that the format matched that of a normal http header, with proper spacing and carriage returns. Also tested error handling by editing the requests that our clients make. We then sent out some bad requests to our server, and made sure that our code was able to catch that the requests were poorly formatted and therefore invalid.

Overall: One common thing we did to test things was to edit the HTTP Message code so that our client would purposely send bad requests. This allowed us to check the functionality of the server. One specific time we used this was when our server was stopping after a bad requests, rather than simply sending the bad request and then continuing to listen for requests. We tested

all variations of the server and client with our own versions and production ones. This involved using wget with our server and using our client to request images and other files from real websites.
Partial response
Partial request
Bad request and see how you handle it

CONTRIBUTION OF EACH TEAM MEMBER AND THEIR UID

Testing and debugging: Everyone
HttpMessage: Max Chern (304290221)
Client: Brandon Pon (504403364)
Server: Alexander Fong (304287624)