CS 118

Project 1

HTTP-CLIENT-SERVER

REPORT

Zi Ming Li [Michael], Hansen Qiu, Richard Sun

Design

Server:

Our server is designed in a similar way to the sample code. It obtains all the command line arguments before setting the socket, binding the address to the socket, and telling the socket to listen.

From here, the server gets put into a loop that is constantly looking to accept incoming connections. Once it does, it creates a new thread and returns to waiting for connections to accept. In the new thread, the server continually receives from the client until a double CRLF is found. After obtaining all the data needed, the server parses through the data using methods from the HttpMessage and HttpRequest classes. These classes provide the necessary functions to break down the Http request into the appropriate headers and file paths.

The file is then opened and the contents are put into an Http response using the HttpResponse class and sent back to the client in the same thread. The server detects timeout if the client opens a connection and takes too long to send a request. The server can be synchronous and single threaded through a command line option.

Client:
For each URL passed to the program, the client creates a new thread to handle that request. The client parses the URL into a hostname, port, and file path. It then sends a GET request for that file over a TCP socket. After sending the request, the client reads the header of the HTTP response in order to check the response status. The client then saves the response's body to a file in chunks. The client also checks for timeout by detecting when a certain amount of time (currently hard-coded as 5 seconds) passes without receiving a packet. The client supports multithreading and sends out requests for each URL in parallel.

HTTP classes:
HttpMessage - the base class which implemented the general structure of the Http message. Headers and values were implemented using a map while there were specific variables for things such as Http version and URL.
HttpRequest - derived class that extends HttpMessage. Used to create and parse through Http requests. Based off of the design provided in class.
HttpResponse - derived class that extends HttpMessage as well. Used to work with the response of the Http message. Also based off of the design provided in class.

The HTTP classes only dealt with the HTTP header. The body was handled separately by the client.

Problems

We had issues in properly sending a file of size around 1Gb. The issue was two fold: not only did we have issues with timeout, the server was not able to properly hold the string for the entire file size. To fix this, we changed the web server to send packets as it was reading it from the file instead of waiting to send the entire file at once. We also properly implemented the timeout to avoid losing the data.

We also had problems being able to properly read from a binary file. This was because there would often be “null bytes” among the binary output. We fixed this by changing the way we read bytes in the client (e.g. using append instead of +=).

Instructions
Run “make” to build the project. There are no additional dependencies outside of the C++ standard library.

Testing
We made a big ~1GB file using random bytes from /dev/urandom to place from the server. Then, we made a shell script to have many clients download the big file concurrently from the server. We ensured that the server was able to handle the multiple connections and send the massive amount of data, while the client was able to download the massive amount of data. We also ensured that all the files we downloaded were exactly the same as the original file using the diff command.
To test how the server handles HTTP requests, we wrote a script to send different bad headers to the server and ensured that they returned the correct status code: 404 for bad headers, 404 for non-existent files, 501 for non-implemented methods (i.e., not GET), and 505 for non-implemented HTTP versions (we didn’t implement HTTP/1.1).

Extra Credit
The client and server can handle timeouts.
The server is asynchronous by default, but can be made synchronous by passing a command line option (./web-server localhost 4000 . --sync). The client likely will time out if multiple large files are requested using the synchronous server.

Team Contribution

Michael Li (904446502) - Http Classes, Server and Client setup, Testing
Hansen Qiu (004490085) - Server implementation, Http classes
RIchard Sun (904444918) - Client Implementation, Testing, Extra credit work