CS118: Project 1 Report

Leslie Lam, 804-302-387
Dominic Mortel, 904-287-174
Kevin Huynh, 704-283-105

May 1, 2016

1 HTTPMessage Implemenation

The HttpMessage class is an abstract class that is extended by the HttpRequest and HttpResponse classes. These classes are used to handle the encoding and decoding of HTTP messages. The classes are defined in http- message.h and http-message.cpp. The HttpMessage class contains several functions; these are a few of the more important ones:

- decodeFirstLine: Abstract function that decodes first line of HTTP Message.
- encode: Abstract function to encode the HTTP Message.
- encodeHeaders: Encode the HTTP Message headers as a vector of chars.
- decodeHeaderLine: Given a vector of bytes, decode the header line and store in map.

The HttpRequest class contains the following important functions:

- decodeFirstLine: Decodes first line of HTTP Request.
- encode: Encodes the HTTP Request as a vector of chars (bytes).

The HttpResponse class contains the following important functions:

- decodeFirstLine: Decodes first line of HTTP Response.
- encode: Encodes the HTTP Response as a vector of chars (bytes).

The biggest difference between the HttpRequest and HttpResponse classes is their implementation of the function decodeFirstLine, due to the differences in the first line of each type of HTTP message. The most important function in the HttpMessage class is the decodeHeaderLine function, which takes a string or stream of bytes and extracts the header and its value. These are then placed in a map that maps headers to values.

These classes were used to implement HTTP messages in our web server and web client. The server and client build the HTTP message from scratch with from a stream of bytes with decode. The messages are sebt as a stream of bytes after calling the encode function.

2 Web Server

The web server is a basic web server that supports HTTP 1.0 with timeouts and works as follows:

Listening for Connections: The server creates a socket that listens to the hostname and port specified in the arguments. If no hostname and port are specified, then the server listens to localhost at port 4000. The server then serves files from the directory specified in the argument list, otherwise it serves from the current directory.

Accepting Connections: When the web server receives a new connection, it creates a child process with the fork() system call. The child process then handles serving the requested file. Because of these parallel processes, the web server has the capability to handle concurrent connections. The server supports three different status codes: 200 OK, 400 Bad Request, 404 Not Found, 408 Request Timeout. If the server does not receive data from the client connection for one minute and the whole HTTP request has not been transmitted yet, then the server will time out and send a response with the 408 Request Timeout status.

Parsing Request: The first line of the request is parsed separately to get the requested file information. The remaining headers are decoded line by line; the method is the same as that used in the client, which is described below.

Sending Response: After parsing every line of the request, the server then attempts to send the requested file. The URL of the request is parsed to find the first singular slash. Everything after the first singular slash indicates the file requested. The server attempts to open the file, and sends a 404 Not Found response if the file could not be opened. After opening the file, the server sends a 200 OK response and then the file if possible. After the response has been sent, the connection is closed.

3 Web Client

The client side works as follows:
Parse URL: The client first parses the URL argument to obtain the host name, port number, and the requested file name. Using these parameters, we create a TCP socket to connect to the server. The HTTP GET request message is always the same no matter the server, with only the URL and host name being changed depending on the use.

Read Response from Server: The client reads the response from the socket using the “recv” function, storing the values in a 4096 byte buffer. We decode the response data into two sections: 1) the headers and 2) the payload which are then stored in an HttpResponse object. These two sections were parsed using the following:

1. Each header is separated by a CRLF (\r\n) and we utilize this property to properly decode them.
   - Initially we parse only the first header which contains status information about the request by processing data from the beginning of the buffer until the first CRLF.
   - From there we move our iterator forward by two bytes so that it is located at the beginning of the next header, move the second iterator to the next CR and decode the line. We repeat this process until we decode all headers, using “recv” each time we reach the end of the current buffer data.
     - If the end of the buffer is reached before the header portion of the message is completely received, then the first iterator is set back to the beginning of the buffer.
     - The client implements some logic so that if there are two consecutive CRLF bytes, it knows that any more data afterward will be a part of the payload.
2. We extract the length of the payload from the “content- length” header of the response and we use that length to create a buffer for the payload. The data that we use to fill the payload buffer are the characters that come after two consecutive CRLF characters, which are known from the step prior. This data is extracted in the following method:
   - We keep a count of how many bytes we need to read (the content-length) and how many we have read so far, dubbed “bodybytes_read.” We loop using “recv” to read bytes into our payload buffer until “bodybytes_read” has reached the content length.
   - If the “recv” calls return 0 for a long enough period of time before the entire message has been received, the client will timeout.
   - The entire message has not been received if either two consecutive ‘\r\n’ bytes appears next to each other or bodybytes_read is less than content- length.

Download File: After obtaining the entire HTTP Response header fields and the payload, the client looks through the headers of the message:

- If the status of the response is '404' or '400,' the client is informed of the error, nothing is downloaded, the socket is closed, and the program finishes.
- If the status is a success, the client will use the file name and create a file with that name using ofstream. A special case is implemented to change the file name of '/' into 'index.html.'

The downloading is then accomplished by copying the entirety of the payload buffer into the new file. The socket is then closed and the program finishes.

4 Problems Encountered

For the web server, the biggest issue that we encountered was handling edge cases when parsing the URL of the request. We had a lot of bugs (corrupted downloading, file not being created, etc.) stemming from an incorrectly parsed URL. Because the request may not necessarily have http:// at the beginning, we had decided to parse the URL by searching for the first singular slash and keeping everything after it. We ran into issues when the first singular slash was the last character or if the slash was not present. We first tried to use sscanf to find the first slash; however, we ran into bugs when the slash was at the end of the string. We eventually implemented it iteratively with a for loop and some if statements.

Another problem we faced was how after finding the content length of the payload, to check whether all of the payload has been read by recv and find out which index to have recv read into the payload buffer. We solved this by creating a counter that kept track of all of the bytes read into the buffer. Depending on what recv returned for the number of bytes read, bodybytes_read would be appropriately modified so that a future recv always copied into the right position in the payload buffer. This portion of the code took a lot of work because any mistake in the counter would cause the payload to become corrupted or incomplete. Another problem we faced was integrating the server and client together and not knowing whether bugs that occurred originated in the client or server source code.

5 Testing

When we built our web server, we tested it with a web browser first. We connected by forwarding a private IP address and port number from our vagrant instance and having our web server listen to that IP address and port number. First we tested serving basic files such as text files and html files. We then expanded our file types to include jpg, png, and pdf files. We defaulted our content type to octet-stream so that if a file type was not recognized, it could still be downloaded by the client. We then tested our web server with our web client to ensure that the server could handle interesting URL cases, such as if the client requests a directory instead of a file.

We tested our client by creating different types of files (.html, .txt, .jpg, etc..) in our server's directory and downloading from our server. Some of the things we tested were attempting to download a directory, different types of files, files within subdirectories, and a 'index.html' file using '/' as the URL. We did these tests because the client should be able to download many types of files, including files in subdirectories. We did not want any corner case to be missed. This testing was important because we discovered a lot of bugs with both our client and server by attempting to download other types of files.

Next, we tested our client by downloading from other servers. We attempted to download images off of random websites and the CS 118 Project Specs to test our client. These tests were done to make sure that our client could work independently of the server and that the file would be downloaded correctly even if it was a very large size.

6 Contributions

Leslie Lam (804-302-387) wrote the web server. Kevin Huynh (704-283-105) and Dominic Mortel (904-287-174) wrote the web client and corresponding sections in the report.