CS-118: Simple HTTP Web-Client and Server

Devin Bedari (204-318-490)
Max Woo (004-330-742)
Vilius Vysniauskas (704-289-956)

May 6, 2016

1 Introduction and Requirements

The aim of this project is to construct an HTTP Web Client that supports sending HTTP/1.0 GET requests and parsing HTTP/1.0 responses, as well as a HTTP Web Server that supports parsing HTTP/1.0 GET requests and sending HTTP/1.0 responses. The design requirement consists of three layers:

- Obtaining and Abstracting the parsing of HTTP/1.0 GET requests, and responses.
- The Web-Client, that encompasses the creation of a socket and the sending and receiving of HTTP requests via this socket.
- The Web-Server, that hosts files in a given directory, on a plausible network host-name and port of the user's choice. The web-server should be able to accept multiple incoming connections from multiple web clients.

2 High-Level Design Description

2.1 HTTP Request/Response Abstraction

This section is straightforward. We split the two implementations into two separate class, the HTTP Request class, and HTTP response class. Both classes are created in the same way. The class uses default constructor for the case where a GET request is to be sent, and parameterized constructors in the case that a GET request needs to be parsed. The parsing assumes that we have the complete is done by checking the first character position, seeing if it matches a header that we have incorporated in our class implementation; if this is the case, we parse the next n characters (where n is the string length of the header) and see if they match the header name. Finally, we parse the information after the header until we reach a carriage-return/newline character.

2.2 Web Client

The web client is designed on a loop, where the number of iterations depends on the number of URLs provided as input by the user. For each loop that executes, we evaluate the following procedure:

1. Open up a socket descriptor in TCP mode
2. Send over a file descriptor, with a generated HTTP/1.0 GET request for the file in question
3. Recv data from the descriptor, looping until we obtain two successive carriage return/new line characters, or if there is content, until we obtain the complete content length, or until the server closes the connection with a client

2.3 Web Server

The web server works in a similar fashion; the server creates a single socket descriptor, which it will use to open a connection with an arbitrary client. The server performs the following actions in a synchronous multi-threaded client loop:

1. Open up a new file descriptor in TCP mode with the client sending the request.
2. Receive a HTTP GET request from the client, and proceed to parse it using our class defined above.
3. Open up the file in the directory, and construct the HTTP Response class with the payload containing the message buffer that we just created.
4. Send the generated HTTP response over the file descriptor opened up in step 1.

3 Installing and Usage

3.1 Installation

Assuming that the user is using Ubuntu 14.04, the following commands install the client and server files:

1. For only the client, run **make client**
2. For only the server, run **make server**
3. For both implementations, run **make**
4. To uninstall, run **make clean**

3.2 Usage

```
./web-client <URL1> <URL2> <URL3> ...
./web-server <host_name> <port_number> <host_directory>
```

The first command is the web-client implementation. The client accepts multiple URLs as input, with the hostname, followed by a colon preceding the port number, and ended with the file path that the user wants to get. The client will assume that if no port is provided, the default port number is 4000. If no file is specified, the client will send a request for "/".

For the web-server, the syntax is straightforward. If any argument is missed in the web-server implementation, the default value for host name is set to

"localhost", the port number to "4000", and the hosting directory to the current directory. The host returns an error response in the case that the a file is not accessible by the server (or does not exist).

4 Testing and Difficulties Encountered

4.1 Testing Parser

We added a separate unit to our design, that took an arbitrary, hard-coded HTTP Request and Response, and checked if the HTTP class would parse both the request and response. The only problems we ran into here, were compile time errors like cstring casting, null-pointer exceptions.

4.2 Client and Server

Tested Client with the implementation of server, and also tried to get index pages from random URLs. We tested the server by using **wget** on the parameters that the server was running on. The problems we encountered here were numerous, from the client not obtaining the complete response, to a halt, because both client and server were in recv() mode. We fixed this by constructing a state machine that decided on what we needed to do at every stage of the sending and receiving of the message.

5 Work Distribution

5.1 Devin

- Single threaded client and server
- HTTP requests and responses classes

5.2 Max

- Multithreading server
- Recv() and Send() Handling

5.3 Vilius

- Parsing input to client and server
- Contributions to HTTP request and response classes

6 Conclusion

Code seems to run in blocking i/o through reads and writes to the socket descriptor. Similarly, **wget** on the server, and sending arbitrary website index.htmls to the client seems to work too. The multithreading is done through synchronous pthread creations.