CS 118 - Project 1

Alex Crosthwaite – Jacob Nisnevich – Jason Yang

May 2, 2016

1 Design

From a top-level perspective, we implemented four different classes, utilizing object-oriented abstractions, to create the web client and web server. These include the following classes: `HttpRequest`, `HttpResponse`, `Client`, and `Server`. In the following sections we will describe our high-level design decisions in implementing each of these classes.

1.1 HTTP Request and Response

For the HTTP request and response abstractions we chose to make separate classes for each message, with slight differences. Both `HttpRequest` and `HttpResponse` have `encode` and `consume` methods that encode and decode the HttpRequest string respectively.

In both cases, the `consume` methods take in an encoded request or response string and parse it to the appropriate class member variables. These functions work in two steps: first splitting the string by new lines and then using `std::regex` to parse the first line and then each of the following header lines.

The class differ in how they treat the header fields however. For the `HttpRequest`, each line of the request string following the request line is parsed to an `unordered_map` of header name to header value. For the `HttpResponse`, all header fields other than the `Content-length` header are ignored due to the implementation of the web client. As such, no map data structure with an ambiguous number of headers is required.

1.2 Web Client

The web client class, `Client`, is instantiated in the `web-client.cpp` file, which takes as parameters one or more URLs. Before being passed to the `Client` class, each of the URL arguments is parsed to a `url_t struct` consisting of host, port, and file path. Each of these `url_t`'s is then added to a `map` of host-port pairs to file paths. This map is created because each individual host-port pair corresponds to a unique socket, allowing for multiple file requests to the same host-port pair to utilize HTTP/1.1 persistent connections.

The `Client` class has one constructor that takes two parameters: the host-port to file path map and the number of URLs. Using the second parameter, the client decides whether to use HTTP/1.0, for a single request, or HTTP/1.1, for multiple requests. Then, for each host-port pair it creates a socket and initializes a connection. For each file path in the vector of file paths for the host-port pair, the client sends an HTTP request to the server, waits for a response, and writes it to a file, assuming the response had a 200 status code. Note that our implementation of HTTP/1.1 persistent connections does *not* use pipelines.

1.3 Web Server

The web server class, Server, is instantiated in the web-server.cpp file, which takes as parameters a hostname, port number, and directory. The parsing of the command line arguments is handled in web-server.cpp and three separate parameters are then passed into the server constructor.

The server uses these three parameters to choose a directory to serve files from and a port number to bind a socket to. The server then begins to listen to incoming requests. Upon receiving one, it spawns a new thread to process the request. The request is processed based on what version of HTTP it has requested. Our server can handle both versions. For 1.0 requests, it sends the data back and closes the connection. For 1.1 requests, it sends the data back and waits for any incoming requests. Since we also have timeout implemented, if it does not receive any requests in the next 5 seconds, the server will close the socket.

2 Problems and Solutions

2.1 Client File Reception

When the client is receiving the response and data associated with it, knowing when the entire file has been transmitted can be a real issue, particularly due to the variety of possible scenarios that could occur, such as a closed connection, a persistent connection with a `Content-Length` header, or a persistent connection with no `Content-Length` header.

For non-persistent connections, we simply waited for the server to close and added all the received data to the buffer. For persistent connections we used the `Content-Length` header and simply read that many bytes from the buffer. Unfortunately, we don't offer support for persistent connections that don't provide `Content-Length` headers.

2.2 Client Multiple URL Handling

When parsing multiple URLs with multiple host, port, file combinations, one of the challenges we had was finding out how best to structure this data. At first, we simply used a `unordered_map` of hostname strings to vectors of `url_t`'s, where `url_t` is defined as follows:

```
struct url_t
{
    std::string host;
    std::string port;
    std::string file_path;
};
```

Unfortunately, this solution resulted in the same socket being used for two URLs with the same hostname but different ports, which is not the correct behavior. As such, we ended up using a `map` of host-port pairs to vectors of `url_t`.

2.3 Handling Servers that Only Support HTTP/1.0

Another issue we ran into was servers that only supported HTTP/1.0, with no persistent HTTP/1.1 connections. As our server defaulted to persistent HTTP/1.1, we had to make sure that we closed and reopened a new socket if the server the client was communicating with could not properly handle persistent connections.

To handle this, we made the `process_request` function additionally return the HTTP version attached to the HTTP response. When the client is looping through the requests to send for a particular host-port pair, it will check to see if the version is 1.0. If it is, it will close the current socket and reopen a new one with the same port and hostname.

3 Extra Credit

For the extra credit portion of the project, our group chose to implement client/server HTTP request timeout and the HTTP/1.1 persistent connections. For the timeout portion, both the server and client timeout after 5 seconds without a response. For the persistent connections portion, the client chooses HTTP/1.1 if there is more than one URL requested for a host-port pair. We did not use pipelining in our implementation of HTTP/1.1.

4 Build Instructions

For the most part, we did not modify the Vagrantfile or Makefile. However, we did add two lines to the Vagrantfile:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
...
sudo apt-get install -y g++-4.9
```

These two lines add the g++-4.9 repository and then installs the new edition of g++. Our implementation of the client and server required this version of g++ in order to use `std::regex` in parsing HTTP requests, responses, and URLs.

We also slightly modified the Makefile to account for this change, replacing `g++` with `g++-4.9`.

5 Test Cases

5.1 Receiving One Localhost Object

The most basic test of the web client and web server was simply to start up the web server with the default settings

```
./web-server
```

and then send a request from the client for a single local object, in our case a file located in the test/ directory.

```
./web-client http://localhost:4000/test/make_test
```

This test case would simply ensure that the client properly sends HTTP/1.0 closed connection requests and receives and properly outputs the response from the server. This also ensures that the web server properly sends an HTTP response and sends the local file being requested correctly.

5.2 Receiving Two Localhost Objects

The next test case we used was two local objects being requested by the client on the same port.

```
./web-client http://localhost:4000/test/make_test
             http://localhost:4000/test/test.h
```

This test case tests the client's ability to send multiple HTTP requests over a persistent HTTP/1.1 connection, as well as the server's ability to send multiple files over the same connection.

5.3 Receiving Two Localhost Objects and a Remote Object

Another test case we used was with two localhost objects and a single remote object.

```
./web-client http://localhost:4000/test/make_test
             http://localhost:4000/test/test.h
             http://awechords.com/index.html
```

This test case checked that the client was able to handle multiple URLs with different hosts, ports, and even HTTP versions properly. This test didn't really test the server as it would be have the same as with the previous test case.

5.4 Server and Client Timeout

As we implemented the HTTP request timeout extra credit assignment for both the client and the server, we had to come up with some method for testing this feature. We ended up simply using `gdb` to stop the client from receiving data or the server from sending data and making sure the other process timed out.

6 Contributions

6.1 Alex Crosthwaite

- Server (50%)
- HTTP Request and Response (10%)

6.2 Jacob Nisnevich

- Client (50%)
- HTTP Request and Response (70%)

6.3 Jason Yang

- Server (50%)
- Client (50%)
- HTTP Request and Response (20%)