CS 118 - Project 1

Alex Crosthwaite – Jacob Nisnevich – Jason Yang

April 29, 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 choe 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 feile, 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

2 Problems and Solutions

2.1 Client File Reception

Problem: How does the client know the entire file has been transmitted

Solution: use content length

2.2 Client Multiple URL Handling

Problem: When parsing multiple URLs with muliple host, port, file combinations, how do we structure our data.

Solution: Use a map from host-port pairs to file path vectors

3 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.

4 Test Cases

5 Contributions

5.1 Alex Crosthwaite

- Server (50%)

5.2 Jacob Nisnevich

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

5.3 Jason Yang

- Server (50%)
- Client (50%)