3D3 Computer Networks

Project 1 - HTTP Server and Client

Group 32

High level design

HTTP Messages

Both the client and the server make use of a common set of abstractions, including the `HttpRequest` and `HttpResponse` classes. These two classes are quite similar in structure. Both have a number accessors to essential elements, such as `status_code`, `method` or `path`, and a list of headers. Both classes also have a static `consume(std::string)` and a non-static `encode()` method. These methods greatly simplify the construction/sending of the requests/responses, encapsulating a large amount of logic rather than leaving the client/server to deal with it.

Server

Upon execution, the server obtains the list of IP addresses that correspond to the address given by the user (either IPv4 or IPv6). It opens sockets, binds one to each address, and begins to listen on these sockets for incoming connections using `select`. When a new incoming connection is available, it starts a new thread using `std::async` to handle the request. This allows it to keep listening for other new connections on the main thread at the same time as responding to connections already underway.

To handle a connection, the server keeps reading data from the connection into a `stringstream` using a buffer. It does so until an error occurs, the connection is closed by the client, or the end of the header, `\r\n\r\n`, is received. This is sufficient, as the supported HTTP methods (GET and HEAD) do not have a request body. The downloaded data is then passed to `HttpRequest::consume(std::string)` for the construction of a `HttpRequest` object. The `HttpRequest` is passed to SimpleHttpServer, which constructs a `HttpResponse` that fulfills the request. This response is then sent back to the client using `HttpResponse::encode()`.

The to prevent interleaving output, errors and messages are printed on the main thread. To achieve this, the `std::future` returned by the `std::async` call is stored and regularly queried for a return value or an exception.

Client

The client is somewhat simpler than the server. Each URL given by the user is parsed and an according `HttpRequest` is constructed. A connection socket is made, and the request is sent using `HttpRequest::encode()`. The client keeps reading the response until the connection is closed by the server or the full message is read in accordance with

the `Content-Length` header. `HttpResponse::consume(std::string)` is then used for constructing a response object, the body of which is saved to a file in the current directory.

Challenges encountered

It was a challenge initially to understand the old, C-based POSIX networking API. The large corpus of questions on StackOverflow and the manual of the functions in question proved very useful.

It was also challenging to find a way such that the entire message would be read from the socket, as TCP does not carry that information. The solution is to keep calling `recv()` as long as there is data to read (return value > 0). After each call, the contents of the buffer would be appended to and `std::stringstream`. In the case of the server, the connection is not closed at the end of the message, but it can be assumed that there is no message body. Thus it is sufficient to terminate after a sequence of `\r\n\r\n` has been detected. In the case of the client, the message can be deduced from the Content-Length header, or the server closing the TCP connection.

At the beginning of designing HTTP abstraction, we only thought about writing two classes which were `HttpRequest` and `HttpResponse` and each of them has a function `addHeader(name, value)` with the public variable headers in the type of vector. However, when attempting to decoding the HTTP GET message, we found that its only possible to add one line of header which is not efficient especially when it comes to a big file containing many header lines. Thus, we decided to create a third class for Header and as a result, whenever we want to add header when decoding the http message, it will be very efficient and decrease our workload.

Testing

The code is tested manually as a single unit, i.e. the executable output is compiled and linked, and tested against a number of inputs. These inputs include valid requests, as well as edge cases. Both the server and the client were tested for correct transmission of existing files, missing files, and invalid requests/responses. They were also tested for invalid user input, such as a bad URL given to the client.

For a commercial HTTP application, this amount of testing would be extremely insufficient. However, for the purposes, requirements, and timeframe of this project, it can be deemed adequate. Given more time, the code would need to be broken up into several disjoint logical blocks, where each block should be individually unit tested, verifying correct operation and the assumptions made, using an automated testing framework. This automated testing framework could also be used to perform integration tests, which are akin to the manual tests performed.

Group members

Aron Hoffmann - 15321585

Contributions: Network facing logic (`web-server.cpp`, `web-client.cpp`, `SimpleHttpServer`)

Siqi Wei - 14330586

Contributions: most of the `HttpHeader` / `HttpRequest` / `HttpResponse` abstractions