CS 118 Project 1: Simple HTTP Client and Server

Jahan Cherian, *104436427* Omar Ozgur, *704465898* Kevin Xu, *704303603*

Abstract

The purpose of this project is to become accustomed to the basics of socket programming and the HTTP protocol. This is accomplished through the development of a server and client. The client would use specificed URLs to create and send HTTP requests to the server, which would process the requests, attempt to retrieve the requested data at the directory specified in the URL, and return the corresponding HTTP responses. The client would take the response and parse it in order to determine properties such as the status code and content-length, as well as to write the requested data to a file.

To further enhance our understanding of the client-server model and socket programming, we design the server to be able to handle multiple connections from different clients. Also, we overcame the challenge of handling HTTP/1.1's persistent connections in addition to HTTP/1.0's non-persistent connections.

1. Build Instructions

In order to build the executable files, simply navigate to the folder containing the source files and run the "make" command. This will create multiple executables ("web-client", "web-server", "web-client-1.1", "web-server-1.1") that can be used to run the client and server, with non-persistent and persistent connections respectively.

2. Server Design

See *figure 1* for an overview of the client-server model and the steps required to set up the sockets.

The web server takes three arguments: a hostname, a port number, and a file-directory, whose defaults are "localhost", "4000", and ".", respectively. The server first resolves the IP address from the hostname and port number with DNS using the Linux function *getaddrinfo* and our own *resolveDomain* function. The server uses *bind* and *listen* and *accept* to set up a socket to listen for requested connections at that address. When the connection is established, it receives the HTTP request message with *recv*, and processes the request to determine the file name and path of the requested object, starting at the directory specified by the given file directory. It tries to retrieve the file and read as a stream of bytes the contents into the data vector that we use to send back to client.

For HTTP/1.0 we use a multithreaded approach for the server to handle multiple clients to serve them each the files requested. Note that if multiple objects are requested for by the same client, because of its non-persistency, the client will close and open a new client for each object even though the host is the same. This is analogous to requesting each object per client file descriptor.

For HTTP/1.1 we use an asynchronous approach using event driven programming through the system call *select*. Essentially we run through a list of file descriptors that and check if the file descriptor is set, if it isn't, then we accept the client file descriptor connection if any, and set it to be watchable, and on the next run, we will find that this client is requesting some data, and so we run into a do-while loop that runs until a *timeout*. In this do-while loop we basically run the same set of steps as we did in the 1.0 server, that is we get the request, decode it, create the filename for the object we look for, grab the file data from the server and create a response object with the associated status code, and then we send the message, update the current timer and loop through again for the next request from the same client.

3. Client Design

The web client accepts any positive number of URLs as argument. The client first creates a socket, which would be used to send messages to a server. For each URL, The server first attempts to establish a connection with the server. Once a connection is made, it uses *send* to send an HTTP request, constructed by parsing the URL, for a web page specified by the URL. The client then parses the response it receives with *recv* to get the status code, content length, and message body (data). To do this, we check for specific substrings and maintain a counter to detect "\r\n\r\n", which delimits the header and indicates the start of the requested data. If the status code is 200, the client creates a file and writes the requested data to the file. Otherwise, the file is not created, and an error is printed.

In the case of HTTP 1.1, most of the client functionality is the same. However, the addition of an unordered map object allows for connection objects to be saved and looked up based on URL information. If the client is connecting to a specific host and port number, and the corresponding connection object is available, the connection is used for sending data instead of creating a new object. After all of the requests and responses have been sent and received, the client looks through the map to find and close any connections that are still open.

4. Obstacles

There were several obstacles in our way, especially during the construction of HTTP/1.1. Below are some key challenges and how we overcame them.

The first main issue came with file corruption, wherein due to ineffective reading on the client side files such as mp3 and mp4 were coming incomplete. We fixed this by looking through the recv section and send section on the client and server respectively. After printing out the contents of the data being sent from the server, we noticed an initial issue in reading the file as characters instead of bytes. We fixed this using a byte iterator to go through the file stream, and then on the client we made sure to effectively parse the received data to make sure we got the correct bytes. This resulted in the eventual fix.

The other big issue we faced was with HTTP/1.1 when we tried to use the same client to get multiple objects within the same connection. Initially this was hanging both the server and client, and so we decided to completely re-write the server to be asynchronous, and use timeouts embedded in the Requests. We used a content length field in the Response to let the client know how many bytes to read at a time within the data. We also realized that when reading from the client, we ran into an infinite loop because we forgot to break out of the read when the content length reached 0.

5. Testing

We developed our software with a modular style so that testing would be easy. Because each component was independent of one another, we could test each one individually. We gave them separate error handling and tested with specific scenarios to check whether the expected behavior occurs.

Testing was primarily done by trying to get any and all types of objects both from public websites such as http://jahancherian.com/ and http://google.com/, and local objects stored on the machine at the specified directory given to the server. Most of the testing was done on a localhost server but the client was tested with any and all kinds of connections. We tested the transfer of all types of objects including mp3 and mp4 to test that the file was not corrupted in anyway during the transfer.

We then proceeded to use Vagrant to emulate the Ubuntu environment to make sure of compliance with the project specifications. To make sure of robustness we ran the web server on any web browser such as Chrome and Firefox to make surer that the web-browser can send it's own types of HTTP Requests (GET) and that the server properly responds with the necessary objects to render the content in the browser. A good example is running the web-server on Jahan's machine in his website's directory and try and render the entire website through the browser which did recursive GET's.

We also tested using telnet to make sure that our server could handle any types of GET requests.

6. Team Contributions

Here is some information regarding the work that each team member did during the project.

Omar Ozgur: Omar mainly developed the web-client implementation for both 1.0 and 1.1 while creating custom Connection and URL objects to assist in the process of generating one-to-one connections per client for the former and one-to-many connections per client for the latter. He helped in debugging and testing the implementations, as well as adding some code to the server.

Jahan Cherian: Jahan focused primarily on writing the HTTP Messages (Request and Response) infrastructure,

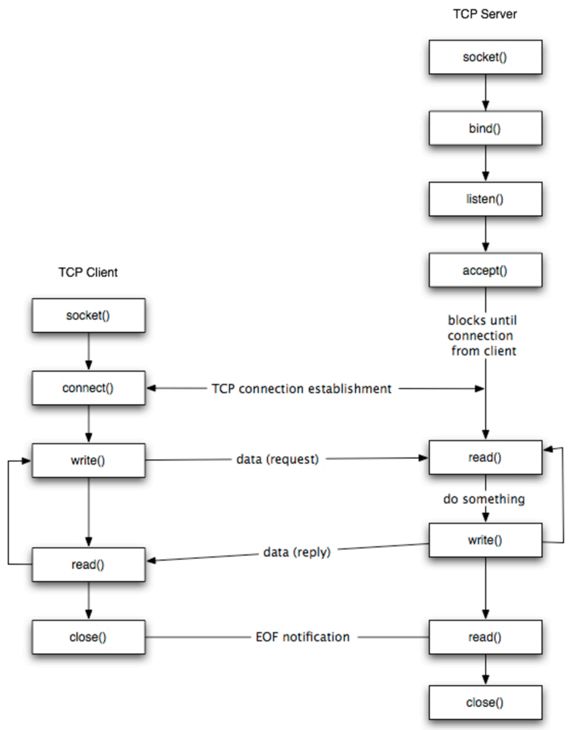

Figure 1: Overview of the steps to set up communication between client and server

and built a large portion of the web-server for both 1.0 and 1.1, using C++ POSIX threads for the former and Asynchronous Event driven programming for the latter, while being helped by Kevin. He helped debug and add some changes to the client.

Kevin Xu: Kevin helped write code for data parsing and reading on both versions of the server, while helping debug 1.0 and 1.1 client-server models. He also helped in building the infrastructure for HTTP Messages with Jahan. He also focused on writing the project report.