**CS 118 Project 1: Simple HTTP Client and Server**

Omar Ozgur, *704465898* Jahan Cherian, *104436427* 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 files (“web-client” and “web-server”) that can be used to run the client and server.

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 “/tmp”, respectively. The server first resolves the IP address from the hostname and port number with DNS using the Linux function *getaddrinfo*. 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 web page, starting at the directory specified by the program parameter. It attempts to retrieve the data from the web page and sends back the appropriate HTTP response.

The server is able to handle multiple connections with multithreading. Since *accept* creates a new file descriptor for each client, the program creates a new thread for each new file descriptor.

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. Then, the client creates a file and writes the requested data to the file.

4. Obstacles

While creating the project files, a few main problems had to be overcome.

In one case, there was an issue with sending bytes to the server while creating the client code. This was first noticed because the client would successfully finish the “send” function, and then it would hang after running “recv”. At first this was thought to be a problem with reading the data, but we came to the realization that it was actually due to the fact that a vector cannot be directly passed to the “send” function. Instead, the vector had to be converted to a fixed-size buffer before sending the data.

Another difficulty we had was having the client parse the response to get the status code and content length. It was a challenge implement extra logic to accomplish the task, as we required a counter and several flags. We later found that this task caused the client to write incorrect data to the file. However, the bug fix was relatively easy.

5. Testing

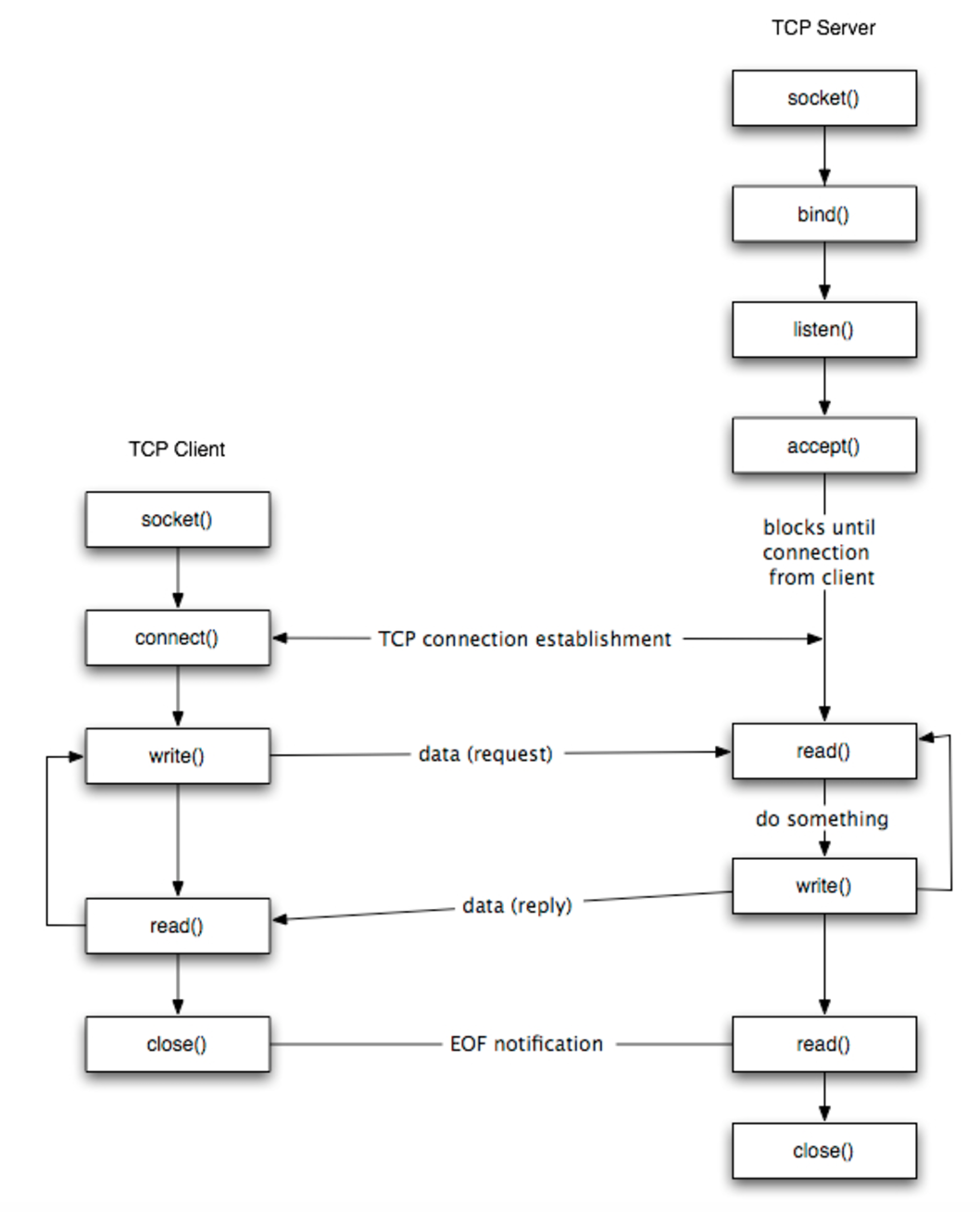
We developed our software with a modular style so that testing would be easy. Because each component was independent of one another, we could test each one individually. We gave them separate error handling and tested with specific scenarios to check whether the expected behavior occurs.

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

One helpful way of testing the client is by requesting a web page that we know exists, such as <http://web.cs.ucla.edu/~lixia/index.html>. When we run web-client with this URL, we would result in a file named “index.html” in the working directory. Then wget the URL to a different directory and run *diff –u* on the two files. We verify that the client works if there is no difference output between the result of web-client and wget programs.

On the other hand, a useful way to test the server is by connecting to it with a trusted web browser, such as Google Chrome. We create a file in directory *tmp* named *test.txt* with the text “*I am a test file\n”.* Then we request the URL [*http://localhost:4000/tmp/test.txt*](http://localhost:4000/tmp/test.txt) on Google Chrome. If the browser displays the text “*I am a test file\n”,* then we have verified that the web-server works as expected.

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, and created abstractions for URLs and connections.

Jahan Cherian: Jahan mainly focused on the web-server implementation, as well as abstractions for HTTP requests and responses.

Kevin Xu: Kevin helped write code for class abstractions for the HTTP request, IP address resolving in the web server, establishing a listening socket, and receiving a request. He also wrote a function to test for bad requests.