Ramzey Ghanaim

CMPE 156 Lab 2

High Level Collaborators: Andrea David

February 14, 2018

Introduction

The purpose of this lab was to create a basic client-server tormenting model where one file can be downloaded from multiple sources while being robust and prone to server failures.

Because the error handling is so integrated into the design and functionality of how my program works, error handling is discussed in my Design section rather than the protocol section. Message types and how messages are sent are defined in the protocol section.

Client-Server Torrent Design and Error Handling

Following the standard Linux/Unix client server model and standard C libraries, I first made a client server model. Both the socket and client created a socket over a TCP connection just like in the previous lab. When running the client, the user is expected to enter a text file containing a list of servers they want to use, the number of servers they want to connect to, and the file they want to download.

Once the client starts, and checks for the proper arguments, it then processes the text file, splitting up each port and IP address into an array for each. Then a connection is attempt is made. My logic finds the minimum of number of servers in the text file, and the number of servers the user wants to connect to. This minimum will be the maximum possible servers the client can connect to.

After the set-up process is done, the client then tries to make connections by iterating through the port and IP lists that were made earlier. Next assign the ports and IPs to sockets and attempt to connect. If an attempt to connect fails, the IP and Port of the failed attempt are re-assigned with the next port and address in the list (if available). This method avoids the need to re-assign socket and address types like `AF_INET` and `SOCK_STREAM`. This process, also allows me to use the iterator as the number of connections successfully made. Next check to see if no connections were made successfully and exit if no servers were available.

Next, the file name entered by the user is put into a message and sent to all servers. My protocol section explains this topic in further detail. Once the server gets the file name, only the first server in the list responds with either the size of the file or "does not exit". If no response is gotten from the first server, the client checks to see if there are more servers in the list to use. If there are, the client tries to connect to other servers in the list. Otherwise the client prompts the user that servers were disconnected, and the client exits.

Assuming there is at least one server in the server file working, the client will now receive the size of the file. Next, the file size will be divided by the number of servers available. The goal is to send each server a `minByte` and a `maxByte` to send back. After the division is taken place, my logic calculates the appropriate min and max bytes for each server, including a remainder if the file size is not divisible evenly by the file size. My protocol is to simply have the last server available send the remaining data with the original data requested by the client.

On the client, a structure called `data` is created for each server that is connected. The structure contains the min and max bytes the server needs to send as well as an error flag, socket file descriptor, an index, and a buffer. Once all the fields are populated to respective servers, (error flag defaults to 0) threads are fired for each server.

Inside each thread the client takes the `minByte` and `maxByte` data and compiles it into a message that my server can understand (see protocol section for more detail). The message is then sent. The server reads the message and breaks it up into usable data. The file requested in the beginning is opened with the standard C `fopen()` function. Next my `sendData()` function is called, where the file pointer is set to `minByte` and we read the file up until we reach `the` maximum size of a temporary buffer. This buffer is simply a subtraction plus one of max and min. The buffer is then sent to the client thread that called the server.

Back on the client, if there is an error in `recv()`, the error flag for the particular struct turns on. Once all threads complete, I calculate the number of server errors/lost connections. If there are no errors, the data is written to a `finalResult` buffer where each row is associated with the index of the struct in my thread array. This is used to prevent data from being written out of order in the final file. For the structs with error flags, I loop through them and re-assign a `socketFD` to a server that did not fail. I then create more threads for these errored servers. If they are successful, the error flag is set back to 0 and the total number of errors is decremented. While the total number of errors is not 0, I repeat the process of changing socket file descriptors and firing threads. But if the number of errors becomes the number of servers originally connected, then all servers failed. The user is prompted, and the client is exited to prevent an infinite loop since error is never not zero in this case.

Once errors are gone, the buffer for each struct is placed in the appropriate row of the `finalResult` buffer based on the index associated with each struct. Now that data is in the right order in the `finalResult` buffer, the data is copied into a new text file called “newFile.txt”. I implemented this, so I did not have to worry about file names colliding when running the server and client in the same directory. The user is then prompted of the new file name if all the data has been sent to a file successfully. The program then cleans up, by freeing data and closing sockets before exiting.

Message Protocols

Since the entire workings of the project was discussed in the Client-Server Torrent Model section, this section will be reserved for the message protocols I made for this program.

In the beginning I want to send the file name to all servers, but I only want one server to respond with the file size. To do this I have two different messages when sending the file name. First, the file name is attached to a string that contains “getFileSize**F** ” (including a space at the end). The server that gets this message is the **F**irst working server in the server list and is responsible for getting the file size and sending the result to the client. (see the previous section for description of what happens when the first server fails). The other servers simply receive the file name attached to the string that contains “getFileSize ”. As in the last lab, I insert an “@” to the end of the message to denote the end of the message. I did not have time to implement continuous sending for large file sizes, so my file name size is limited by buffer size - message size, which comes out to be $1280 - 12 = 1268$ characters. If a file is not found, the message “File Not Found@” is sent to the client.

I have my logic detect which message arrived and deal with it accordingly. When the first working server gets the file size (in units of bytes) and appends it to the string: “file size ”. Once again, the size of the file in bytes is limit to the buffer size, 1280 times the number of servers in use. Since my design only deals with text files, I do not think I needed to handle more.

The last message type used is the “dataSize ” message. The structure contained in each thread contains a `minByte` and a `maxByte`. These numbers are appended to the “dataSize ” message and sent to the server. The server is programed to assign the first number following the message is the `minByte` and the second number is the `maxByte`. A space is inserted between the two to denote when one ends and the other begin. At this point the data size being sent is known to both the client and server, so no message protocol is needed to denote when/if data is sent successfully. The size of the data received determines the success of sending the data.

Usage

These instructions assume the user is in the main directory of this project

1. To build the program simply type:

   “make”

2. To run the server, in one terminal in the bin folder type:

   “myserver <portNum> <e>”

Note: The <portNum> must match ports specified in server-info.txt to succeed.

Note 2: Type “e” as a second argument to have the server exit after a connection is established to test if the program can complete if a server cuts out after connections are made.

3. Next run the client in a different terminal in bin:

   “myclient server-info.txt <serverNum> <fileRequested>”

   The file created will be stored in “newFile.txt”

4. When one wants to clean the projects simply type:

   “make clean”