Ramzey Ghanaim

CMPE 156 Lab 3

February 26, 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.

Client-Server Design

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 UDP connection. 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. If 4 servers are running and the user types 1, and the first server found fails. The program will fail to download the file.

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_DGRAM`. Next check to see if no connections were made by waiting for an ACK or a file size message type. Each packet is a struct called "message". Each message has a server and client sockaddr_in the packet is coming and going to along with a buffer for data, and a message type that can be FileName, ACK, gFileName, FNF, or, givBytes

Next, the file name entered by the user is put into a message struct 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 "FNF" if the file is not found. 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. If successful, the server forks a new process and provides the client with a new socket to for the rest of communication. If no servers work, the client prompts the user that servers were disconnected, and the client exits. During this time, I have a timeout of 2 seconds per server, and I try to connect 5 times.

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. With a data buffer of size 1000 for each packet, the server will determine how many packets it needs to send based on the total chuck size of `maxByte - minByte`.

During the sending of the packets, if even one packet is not received after 5 tries, I try to get a different server. This means if all servers have any percentage of packet failure, the file will fail to download. This made the design of my program much easier at the expense of being robust.

Once the chunks are all downloaded, each server is sent "done" message so the servers can kill the child process that handled the existing connection. Once again, the server waits for the message for up to 30 seconds. (5 sec time out with 6 tries).

Message Protocols and Error Handling

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 placed on the packet's buffer, while the string "gFileName" is placed on the message type (msgType) attribute of the packet. 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. The other servers simply receive the message type of "FileSize". These servers only send ACKs back to the client. I did not have time to implement continuous sending for large file sizes, so my file name size is limited by the buffer size which is 1000 characters. If a file is not found, the message "FNF" is sent to the client. If the server is not the first I try re-transmitting up to 5 times if the client does not receive the FileSize or an ACK. When the server gets either the file name message, it forks a process to handle this client on.

When sending the actual data, the server fills out the `outOf` and `index` fields of the message structure are filled. This keeps track of every packet and out of how many total packets are going to be transferred. If even one packet fails, a new server is looked for.

After the transfer, the client sends back a "done" message type telling the child process in the server the connection can end. The server sends and "ACK" message to confirm. I retransmit if failures occur up to 5 times. Other than re-transmissions, I do not handle lost ACKs thought the entire project.

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 `serverBin/` 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 the `clientBin/` directory:

   "myclient server-info.txt <serverNum> <fileRequested>"

   The file created will be stored in the directory of the myclient program

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

   "make clean"