Adam Homann, Alex Preston, Bikram Singh, Zur Nachshon

Study Simulator for Protocol 4, 5 using Client Server

http://www.supergeek.ca/wp-content/uploads/2014/01/global-network.png

Professor: Mortezaie Faramarz

Abstract

This project made use of sockets in the C programming language, while simulating Protocol 4 and 5. The idea is simple, yet the implementation was very complex and took a lot of time. Protocol 4 and Protocol 5 were some of the protocols learned in this course (CS 158A) and so making a simulation to show it in action was very relatable to it. In order to pass information there has to be two “stations” and so therefore, a socket has to be created so that information can be passed from one station to another to demonstrate these two stations. This project required two terminal windows and ran one as the server and the other as the client, ultimately simulating two stations.

Introduction

To simulate two real world stations, there are many ways to code this, but the way it was done for this project was to use two C programs called server.c and client.c and set up a socket between them. And as the names suggest the server.c file is for running the server, and the client.c file is for running the client. Each of these two files will run within two different terminal windows. The first window will simulate the server, while the other window simulates the client. The server is started first, and then the client, and it has to be in that order because the server must be active so that the client can connect to the same port. The user will type anything into the client side, this is passed to the server side, and then passed back to the client side. If the user types in “Protocol 4” then the Protocol 4 loop is started. If the user types in “Protocol 5” then the Protocol 5 loop is started. In order to exit these Protocols at any point within the protocol loop, just type in “Exit” and the user will be back out in the opening simulator mode. If the client is terminated by ctrl-c, then the server is terminated as well, but not the other way around. The socket in C is a struct that runs in the background and is the reason for this behavior.

The way the simulator is set up is that the user types in data which is stored with the use of fgets(), and then written into the socket struct that the server then reads from on the other side. Every time there is a write to this, there must be a read before another write can happen, otherwise there is an error in which the client doesn’t prompt the user for a message. This is what made Protocol 5 hard to implement, but more on that later. There are while loops within the Protocols so that when the users enter these loops,

they stay within the Protocol until an “exit” is typed in as mentioned earlier. Every time that the server receives the data from the client, the server sends back an acknowledgement. The client then prints this out onto the screen to show the user everything went through fine. The server also prints the data out to prove that the acknowledgement was sent due to being received. This project demonstrates the use of setting up two stations, with the use of ports and sockets, to implement a simulation of Protocol 4 and 5, all of which was learned from this course.

Body

Communication and Protocols: In the real world information is passed from station to station, and there needs to be a way to make sure no collisions happen, the channel is utilized to a good amount, and that the receiving end did receive each frame. Protocol 4 makes sure no collision happens by sending one frame out at a time, and also makes sure that the receiving station got the data by sending an acknowledgement back to the sender, but the utilization of the channel is terrible. This is due to the sending station having to sit idle until the acknowledgement for the current frame has been received before it can send out the next frame. This is where Protocol 5 is a better protocol. Protocol 5 makes sure no collisions happen, that the receiving end has received the frames, and that the utilization of the channel is high. Protocol 5 sends out multiple frames at once, and keeps record of which have received an acknowledge back, and which ones havn't. The amount of frames it can send out is called its window size, and this window size should be optimized to allow so that the channel is utilized at every point along the signal path. So if the bandwidth and distance make it so that 5 frames at any moment should be sent in order to make full use of channel, then the window size should be 2^3 = 8 in order to hold the necessary acknowledgement. In this project the window size was made to be size 8. Any larger size gets hard to demonstrate for academic purposes. So, using sockets in C and simulating Protocol 4 and 5 is relatable to the course and is therefore what this project is about.

Client Server Socket: Two instances of the sockaddr_in are made: server_addr and client_addr. Then to create a socket do: sds = socket(AF_INET,SOCK_STREAM,0) ; AF_INET stands for Address Family from the Internet and is an int with value 2. SOCK_STREAM is also just an int but of value 1. These tell the socket what type of IP to allow, and in this case it is for Internet Protocol v4 addresses. Using the atoi() function to convert the ascii characters into numbers, take the address argument located at argv[1] and set it to the port number. Set the server_addr address family type to Internet Protocol v4. Take the port and make sure it is in big-endian format by using htons() and then store this result into the server_addr port. Finally bind the socket with the server_addr struct, and again with client_addr struct. This part was hard to understand but it all worked with the help of this tutorial: https://www.youtube.com/watch?v=V6CohFrRNTo

Protocol 4:

Our code simulates protocol 4 by using a char array for the frame. Initially when the user types the command “Protocol 4”, the code switches to protocol 4. When a user types a message, it simulates that message as if it were a packet. It takes input and puts it into a char array named frame, and it then tacks a 0 onto the end. It is then sent to the server, where it is read and evaluated. The server, on the terminal side prints the message with the 0 changed to a 1 to show what it sent back if it was received. If the frame expected is 0, it then changes the ack 0 to a 1, and sends back a message that says “Ack Received.” This is displayed on the client side. To simulate that this is

working, the code waits a given amount of time before the ack is sent back to the client side.

Protocol 5:

We implemented part of protocol 5. Similar to protocol 4 we used, char array represent package and then additional char array to create a frame. User input is used as input package from stdin. Then we add sequence number to the frame and store the frame to a two dimensional array. Frame is then written to the socket and made available to the server who is listening on the same port. We are using 3 bit sequence number to transmit frames. After the server receives the frame it rips off the sequence number and send back to the client as acknowledgement.

Conclusion

This project helped put what was learned in the course into practice. The language of C can be tough to grasp at times and this was no exception. It was tough figuring out how to set up a socket and writing and reading between the two connected endpoints. Learning about Protocol 4 and Protocol 5 in the course and then simulating them in C was challenging. Dealing with multiple writes without a read was troublesome and tricky when implementing Protocol 5. Besides strengthening the skills learned from this course, getting good at using C again was very helpful since compilers such as Python are written in it, and it is a language right above assembly. As a group is was good to work with Git; all of the teammates had worked with it before, some better at it than others, but it was good to get back into the swing of using it.

The things that would be done differently would be to make the code more readable and cleaner. So much time was spent on figuring out how to implement the code, and not enough time was spent on cleaning it up and making it simpler such as using more functions instead of spaghetti code. The parts that could have been added to this project were Protocol 6 and checksum. Checksum would have been fun to implement, but the group ran out of time and had to focus on making what was already done presentable. Much was learned from this project, and it was really nice being able to apply the subject matter learned from the course to this project.