Computer Science 118
Computer Network Fundamentals
Project 2: Simple Window-Based Reliable Data Transfer

Robert Geil, *UCLA Computer Science Department,* UID: 104916969
Levi Weible, *UCLA Computer Science Department,* UID: 104934015

Abstract

Utilizing the sockets available within the C++ library, a server and client to send and receive files through packets can be built. When the server is run, a UDP connection is opened,waiting for a client to connect to that socket. When the client is run with the specified socket and file name, a three way handshake happens establishing a connection between the client and server. Then, the file is split up into packets containing headers with file information. The packets are sent over the UDP socket by means of Slow Start and Congestion Avoidance TCP protocol. Once the whole file is transmitted, the server saves the file to the current directory, and the client notifies the server that it will close the connection, and both the client and server close that connection. The client exits while the server waits for the next connection.

1. Summary

1.1. Project Design

The server side first opens a connection on a UDP socket based on the command line port number. Once the socket is opened, the server enters a loop to check for incoming connections. Once a connection is found, the server accepts the first SYN packet from the client, printing out the message. Each packet is build using a packet constructor which holds all of the information in the packet. The RECV and SEND messages are printed based on packet functions and which is applicable. Once the SYN message has been received, the server sends an SYN ACK and then goes into another loop to begin receiving the data. There, it waits for packets from the server. Using the UDP socket built in timeout, if 10 seconds have elapsed, the server leaves the file alone and closes that connection, incrementing the connection number and going back into the outer loop to wait for a newly client connection. Once a packet containing data has been received by the server, the sequence number is checked, and if it matches the next expected value, the packet's data is immediately written to the output file which is set up as a file stream. If the received packet's sequence number does not match the next expected value and that packet is within the reasonable cwnd, that packet is added to a map that stores packets based on their sequence number. Then, once the next expected packet has been received after the client retransmits it, the server checks the cache and prints any packets in order to the file stream. Once all of the data has been written to the file and the server receives the FIN packet from the client, the server sends a FIN packet, closes that connection, increments the connection number, and defaults to the outer loop to wait for the next connection. If the server is interrupted with *CTRL-C* and the server has an open file stream, "INTERRUPT" is printed to the open file, and the server closes all current connections and sockets. On the client side, a socket is opened to the user-provided IP address and port. A new Packet is created with the SYN bit set, and then sent to the server. Once the server responds with the SYN-ACK message, the client opens the provided file and begins transmission. To run transmission, the client has 3 worker threads running queues to transmit and receive messages. The transmission thread holds one end of a queue and empties the queue into the socket to transmit. The receive thread continually reads messages and pushes them into a queue of ACKs. The retransmission control thread continually reads from the incoming queue and matches ACKs with packets that are "in flight". Those that are matched are then removed from the window, and the cwnd and ssthresh values are updated accordingly. In addition, every 0.5 seconds, if an ACK hasn't been received, this thread retransmits all packets that are currently in flight. The main thread meanwhile reads in from the file, incrementing the sequence number, and queuing up packets whenever there is space in the in flight unordered_set. Finally, once all the packets have been sent, the client waits for ACKs to come back

from the server and then sends the FIN message. Each incoming message is responded to with an ACK for 2 seconds, and then closes the connection.

1.2. Difficulties and Resolutions

One of the difficulties faced when building the server side of the project was being able to save each file no matter the content of the file. Initially the "<<" operator was used along with a file stream to the specific file; however, when testing with large binary files, the packet data was not saved properly. This was fixed by using the "write" function to write directly to the file.

Another issue that was faced was knowing if all of the ACKs had been received on the client side. Originally in the implementation, the client just exited upon sending the last packet, but this would not account for a dropped packet, so an all ACKed flag was used where the client would loop waiting for ACKs until the ACK number of the last packet lined up with the last sequence number transmitted.

A major difficulty that was faced was implementing the cache on the server side for when packet loss occurs. For a while, junk packets were being saved to the server's cache when they were not supposed to, which resulted in the server thinking that it had received a packet when it was really an old packet with the same sequence number. This was fixed by checking the incoming sequence number and asserting that it is within a certain window of possible sequence numbers before being saved to the server's cache

Another major difficulty faced was coordinating the incoming ACKs with the sent packets on the client side. This was done using queues and matching the packets with the ACK numbers to make sure that those packets are not retransmitted after they have been ACKed.

Setting the client to be multithreaded was also a major source of difficulty. A series of locks had to be carefully synchronized to allow each queue to operate successfully.

2. Operation Instructions

To operate the client and server, a relatively simple series of steps needs to be carried out.

2.1. Make

Within the directory for the server, use the `make' command to generate all required files for server operation. Other `make` targets include *clean* to remove all built files from the working directory and return it to a newly opened state, *client* to create the executable file for the client side, *server* to create the executable for the server side, and *dist* to build a tarball distribution of the server program.

2.2. Run

Once both the server and client have been compiled, the server can be first run using the command *./server [port number]*. The port number field can be filled in with the user's choice of port number that the UDP socket will be opened on. After running this command on the command line, the server will create the UDP socket and wait for a client to connect. The client side can then be run from the command line using the command *./client [address] [port number] [file name]*. The address represents the IP address that the client will use for the UDP connection. The port number should be the same as the port number used by the server side. The file name is the desired file that the client would like to transmit to the server. Once this command is run, the client will split the file into packets and deliver it to the server. The output module will display each message sent and received by that side, and the file will be saved with the appropriate name in the server's directory. After completion, the client will close, and the server will again wait for another connection. The server can be shut down by using the *CTRL-C* command in the command line, at which point it will stop waiting for a connection and close.