CS 118 Project 2: Simple TCP-like Transport Protocol Over UDP

Jahan Cherian, *104436427* Omar Ozgur, *704465898* Kevin Xu, *704303603*

Abstract

TCP is a transport-layer protocol that allows for data to be sent reliably over an unreliable network. This generally requires more complexity than an unreliable protocol like UDP, but it is useful for software that cannot tolerate packet loss. This report explains the process of creating a TCP-like protocol over UDP in C++.

1. Introduction

For this project, we were required to implement TCP Tahoe over UDP sockets within C++ to provide a deeper understanding of the transport layer and congestion control in general. The project was carefully designed to be extremely modular and clean. Thus we chose to break the project up into the creation of basic TCP Packets, as per the project specification with 8 bytes' worth of Header data and a maximum of 1024 bytes' worth of data. These packets were then used within a singleton like TCP Server and TCP Client. These singleton-like structures were finally used within the server.cpp and cli- ent.cpp respectively to run the overall process of TCP communication.

This report focuses on the design and features of each of the classes, and how we tackled the issues that came in our way while implementing efficient and reliable file transfer over UDP.

1.1. Build Instructions

To build the executables, simply run the command '**make**' within a shell. This will build two executables, *simple-tcp-client* and *simple-tcp-server*.

We begin by running the server as follows: ./simple-tcp-server [port- num] [filename], where the port number is the port number to bind the server to, and the filename is the name of the file we wish to transfer to the client.

We then run the client as follows: ./simple-tcp-client [dest-ip-ad- dress] [dest-port-num], where the first argument is the IP address of the server we wish to connect to (can be found using the **ifconfig** command within the server instance) and the latter is the port number to send data through.

Executables and other produced files can be removed by running '**make clean**'. There are other make instructions that can run, such as creating just the client or the server. This can be found within the Makefile,

1.2. Testing

Most of the project functionality was tested through the use of Vagrant, which allowed for network parameters such as packet loss and packet reordering to be easily manipulated. By running separate client and server instances on a personal computer, files could be sent with minimal setup. Tests were performed on various file sizes and media types to ensure that character encodings and memory restrictions were properly dealt with. Various combinations of packet reordering and packet loss parameters were also utilized to see how the program worked in unique circumstances. These tests allowed for flaws to be acknowledged and fixed efficiently.

2. Project Overview

As mentioned in Section 1, we chose to approach this problem with somewhat of a singleton design pattern. Note that we say "singleton like design", because the implementation doesn't actually generate a singleton, but there is only ever one instance of a TCP server created and likewise for the client. Because the communication required some kind of data structure to transfer across, we created a TCP Packet object to help contain the key information per transferrable object. We also decided to create an abstract TCP class that contains methods for handshakes, data sending and reception from packet level to file level. From this base, the server and client were built.

2.1. TCP Classes

The key files to pay attention to are those prepended by *packet* and *tcp*. These contain the key objects and the method definitions that are then at a high level called in server.cpp and client.cpp.

2.2. Packet Abstractions

Each TCP Packet is defined to be of *MSS* size, where we defined *MSS* to be **1032** bytes, wherein the first **8** bytes are for the header, and the rest *PACKET_SIZE* (**1024** bytes) are for any data we would like to place. Thus the Packet contained a header object that was designed as a private structure with **4** fields of 16 bit values for the ACK, SEQ, WIN and FLAG fields.

The overall packet also contained a vector of bytes to store data, and contained the ability to *encode* and *decode* to and from byte streams. The only difficulties lied in encoding and decoding the header, as we had to split the 2 byte values into two separate 1 byte fields, by shifting them and breaking them, and then combining.

The encoding was done by taking the bitwise AND with 0xFF which gave us the lower byte, and then the original value arithmetic right shifted by 8 (1 byte) gave us the higher order byte. The decoding was done by taking the second, left shifting and bitwise OR'ing with the lower half. Note that this ensured the transfer was external data representation compliant (XDR) in that the endianness was not an issue.

This packet, was implemented under the rule of three, in that it overloaded the copy constructor and assignment operator, because the encoded data stored was a dynamic byte array. Other key information the packet stored included the time it was sent as a *timeval struct*, whether the packet was acked and sent. This allowed for encapsulated access to start the timer based on the current time, whether the packet timed out, based on the difference in time between the sent time and current time was greater than the justified RTO (time out at **500** milliseconds).

2.3. TCP Handshake

Since both the client and serve shared some functionality, we decided to create a base TCP class. Some of the shared functionality was sending, receiving of data (individual encoded packets), setting socket timeouts and most importantly, the handshake.

The handshake was overridden by the server and client, wherein the client begins the communication by sending a SYN to the server, which then responds with a SYN-ACK with a **random start sequence** number. Upon receiving the SYN-ACK, the server responds with the ACK, now asking for data (all of this is indicated by the flags set). This ends the handshake on the client side, and upon the server receiving this ACK, will end its hand- shake and begin it's *sendFile* function to start chunking the file and sending it over to the client as per TCP Tahoe. Note that within the handshake, we set timeouts on the sockets in order to be able to catch retransmissions, while also checking the flags of the packet in order to ensure we are correctly reading the packets as intended, so that the client and the server begin to behave in phase with each other. This handshake then sets the stage for the file transfer, since the next requested sequence number from the server is known.

2.4. Close Connection

After the server finishes sending a file to the client, it sends a FIN packet to notify the client. The client then proceeds to acknowledge the FIN and send back its own FIN packet. Afterwards, the server sends back an ACK to notify the client that it has received the FIN. At this point, the server uses a timed-wait mechanism to determine when to close the TCP connection. If a timeout occurs after a "sufficiently large" amount of time (2 * RTO in this project), the server determines that the client received the ACK, and therefore didn't retransmit the FIN packet. However, if an ACK was received during the timed-wait, the server would know that the client did not receive the ACK, and that it should be resent. This method is usually effective if the timeout value is high enough. However, it is possible for the server to close the connection before the client receives the ACK, which would cause the client to continuously resend its FIN packet.

3. Server Implementation

Upon analyzing the program as the whole, we recognize that the server requires a lot more functionality than the client. This is obviously because the server needs to fetch the file, break it into chunks, and then send it to the client based on the TCP Tahoe Congestion policy, holding a congestion window, in order packet buffer, and all the while making sure to send the data regardless of the situation of loss or reordering.

The server is initialized to be listening on the specified port at "0.0.0.0" so that the mapping can be made to any NIC on the machine. After opening the socket, binding to it and setting a listener to it, we begin performing more of the TCP related algorithms.

3.1. File Chunking

The break of the file into chunks is done in somewhat of a *lazy* manner, in that we only break up the file when those chunks are required. This takes away from the need to have large memory allocation to store all the file packets *eagerly*. The way we accomplish this dynamic chunking, is by opening the file as a stream, which then sets an internal file pointer. We open the file within the construction of the server, wherein we initially get the overall size of the file in bytes by seeking the pointer to the end and calculating the difference through *tellg*. With this information, we build a *grabChunk* function that takes an optional parameter describing the number of chunks (defaulted to 1). With this we loop reading in chunks of *PACKET_SIZE* (implicitly moving the file pointer), always checking whether this fixed size packet is chunkable from the file given the current file pointer position in the file. That is our very last packet has the possibility of not grabbing a fixed size amount, since there might be < *PACKET_SIZE* data left. Upon creating a file packet of this last *remaining* data, we close the file and return. Note that the file chunks are stored in an internal vector in the server.

3.2. Sending File

This is probably the most complex part of the Server implementation. Essentially, after the handshake, the server begins by grabbing just 1 chunk, and running a loop that continuously runs until there is no more data left to send. In order to send we loop through how much data we can send based on 3 key factors. These are the *congestion window (cwnd)* – determined through the congestion control; *file packet buffer (filePackets)* – grown lazily by grabbing chunks and pushing them into here as needed (this value will always be capped off at a later stage, to prevent memory disasters); *receiver's window limit* – this is information from the client describing how much data it can currently buffer, before it flushes the data to the file on its end. We set our upper bound of how much data we can send as the minimum between these three packets. Sending the packets is done by transferring the encoded data into an intermediary sending buffer that then uses the system call *sendto* to send it through the socket as an *MSS* chunk of bytes. We also use a resend flag to denote if this transmission was in fact a retransmission based on either a timeout of the packet, which we sense before sending, or a duplicate ack which also immediately re-transmits (Fast Retransmit for both stages). We keep sending while we receive no data, which we denote through a do-while loop using a non-blocking *recvfrom* to sense if we receive some data – which would mean we are receiving acks from the client. We then enter the second stage of our *sendFile* function, wherein we run a do-while loop while we are receiving acks through the socket, wherein we check the ack number we receive, and run it through the congestion control functionality as explained in Section 3.4. After receiving the acks, we run through the stored buffer, removing all the packets based on the most recent ack received. Now that we have re- moved some chunk of packets from our stored buffer, we are free to grab more chunks based again on the three factors stated in our sending of packets. In this case we grab the minimum of the difference between the window and buffer size and the difference between the receiver window and file buffer size. This ensures that we will always cap off the in order server packet buffer based on these factors, with the uppermost bound being the receivers window. The function upon finishing the file send, will end the connection by initiating the server teardown procedure, in which the server will send a FIN, receive a FIN-ACK from the client with a corresponding FIN after from the client, and then will send a FIN-ACK to the client, and assume the connection is dead after a *TIME-WAIT* based on 2 * RTO.

3.4. Congestion Control

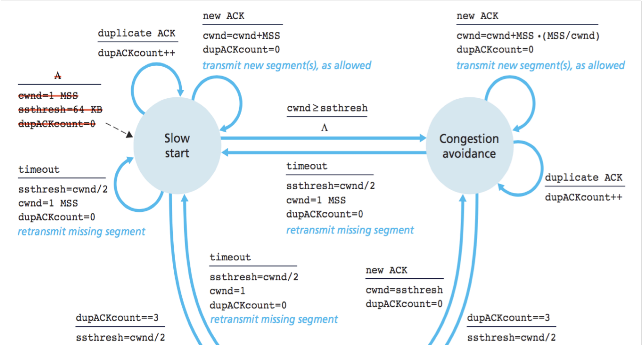

Figure 1: General TCP FSM followed, barring the inclusion of fast recovery, and different values for ssthresh.

The congestion control implements two key states – Slow Start and Congestion Avoidance - based on TCP Tahoe. We essentially follow Figure 1's FSM representation of TCP congestion. We store the last ack received, and if it matches our new ack, then we increment the duplicate count, and if this count ever hits 4 (that is 1 original ack and 3 duplicates), we immediately retransmit the packet as per Fast Retransmit and reset the *cwnd* accordingly. This is analogous in both Slow Start and Congestion Avoidance. If there is no duplicate, we increment the congestion windows by 1 for each packet for Slow Start (*Multiplicative Increase*) and by *cwnd + 1/cwnd* (*Additive Increase*) for Congestion Avoidance. If while in Slow Start we hit the *ssthresh* then we move into Congestion Avoidance for the next packet. If we receive a timeout as mentioned earlier, we always reset the congestion window back to one, half the *ssthresh* and move back to Slow Start.

4. Client Implementation

The key for the client was to perform the *out of order buffering* so that we can ensure the data is always written as it was sent, thus making the overall connection reliable.

4.1. Receiving Data

The client begins by grabbing the header values from the packet it just received. If the sequence number received is an expected sequence number – based on the previous sequence number + the packet data size – then we write the data and then push the packet into a fixed size buffer acting like a dequeue called *m_written* which stores all the packets that have been written to our file. If this dequeue is about to go beyond the receiver window limit, then we pop from the front and push this new packet to the back. This always ensures that we never re-write data regardless of getting re-transmitted packets due to network loss. If at any point the packet received contains the FIN flag from the server, then we proceed to the closing of the connection.

4.2. Sending Data

The client never truly sends any data to the server, but rather creates empty packets with only the header being of importance, since it contains the information about the next expected sequence number the client expects, along with the overall size of the receiver window. In order to send the correct ack number to the server, the client essentially reads how much data it just received based on the return value from *recvfrom* and adds the total packet size to the sequence number sent by the server.

4.3. Packet Buffering

In order to avoid re-sending packets that arrived out-of-order, a buffer is used to store them. When a packet arrives, the client checks to see if it has the expected sequence number. If it does, the data is written to the file, and the buffered packets are checked to see if any other packets can be written to the file. If the client was not expecting it, the packet is placed in the buffer. A separate vector is used to keep track of the data that was already written to the file so that data isn't written multiple times. These mechanisms allow for out-of-order packets to be effectively stored, which prevents the need for unnecessary re-transmissions.

5. Difficulties Faced

This project required a lot of planning in order to ensure that components would work together properly. Although the project was implemented in a modular fashion, the team found it difficult to change certain aspects of the program that became entwined in many different components. It was also fairly difficult to debug the program when errors occurred, since features like congestion avoidance and slow-start mechanisms were hard to analyze with system output alone. It was also difficult to determine how to deal with events that should theoretically be performed in parallel. This was mainly simulated through the use of non-blocking I/O calls and event loops. Although many aspects of the project were challenging, good teamwork and modularity helped in the creation of a reliable platform.

6. Contributions

Omar mainly focused on the client implementation. He added client features such as the TCP handshake, file finalization, sending and receiving, and out-of-order buffering. He also created TCP abstractions that improved the modularity of client code, and helped to implement server logic such as creating file packets and dealing with congestion control. He took on the brunt of the connection closing mechanism.

Jahan primarily built out the packet logic and design, laid the foundation and built the server with Omar and Kevin. He focused on creating the file chunking mechanism, congestion control and file sending. He created the handshake and the construction for both client and server, and worked improving design and modularity of the C++ code. He also worked on the report.

Kevin helped build out the packet logic, adding key methods and fields to the header struct. He also helped lay out some of the foundation with Jahan for both the client and server, constantly providing some form of base functionality.

7. Conclusion

Implementing TCP over UDP was a major challenge, but one filled with a lot of learnings. The project helped solidify the foundations and deeper workings of TCP than initially discussed in class. It provided a practical example of how networking works in the real world.

In addition to the difficulty of implementing the logic of the congestion control, this project really helped highlight the benefits of strong design and modularity within code, and how to actively debug and press on forward.

Overall the project was a challenging task, but one filled with a lot of experiential gain.