CS118: Project 2 Report

Leslie Lam, 804-302-387
Dominic Mortel, 904-287-174
Kevin Huynh, 704-283-105

June 5, 2016

1 Design of the TCP Header

We created a class to implement the TCP header described in the spec. The class contains 4 uint16_t variables: sequence number, acknowledgement number, window, and flags. This class was used to make the packaging and unpackaging of TCP headers easier. The two important functions of the class are encode and decode. The encode function generates a vector chars, which contains the TCP Header information in the correct order (big-endian). This was achieved by using some bit-wise operations to extract the first and last bytes of each uint16_t variable and pushing them onto the vector. The decode function takes a vector of chars and fills in the variables of the TCPHeader class. This is done by examining each pair of chars in the vector and combining them into one uint16_t variable.

2 Design of the Client

The client first creates a UDP socket using the IP address and the port number of the server. The socket options are set to add a timeout option of 500 milliseconds. Using a TCP Header class that we created and a vector buffer to hold the object, the client sends a UDP packet consisting of the TCP Header with the appropriate flags set for SYN using sendto. Then the client opens up a file named received.data to eventually receive data from the server. The client then goes into a while loop. Using gettimeofday, the client repeatedly sends SYN packets to the server once every 500 milliseconds until it receives a SYN/ACK. Next, the client uses recvfrom to receive packets from the server. The client keeps track of the next expected packet using the sequence number of incoming packets. It calculates the next expected sequence number using the previous sequence number and the size of the payload. The client uses modulo to implement wrap-around of the sequence numbers. After decoding the vector buffer to obtain the TCP Header and the payload, we use a series of if-else statements to categorize the packet using the flags of the TCP Header. If the packet is a SYN/ACK packet, the client sends an ACK to start the download process. If the packet is a regular ACK packet, the clients appends to the downloaded file only if the sequence number of the packet is the next expected packet. Thus, if the sequence number is not the next expected packet, it is discarded and not buffered. Next, the client sends an ACK packet to the server with the ACK number set to the next expected packet from the server. If the client receives a FIN packet, it sends a FIN/ACK packet and prepares to terminate the connection. Using gettimeofday, if the client does not receive a final ACK after sending the FIN/ACK after 500 milliseconds, it resends another FI N/ACK. Lastly, if the client receives an ACK packet with the correct sequence number, it will terminate the connection and close the file.

Some problems that we faced implementing the client was how to implement timeout for sending the initial SYN and the final FIN/ACK to start and terminate the connection. Another problem that we faced was how to correctly calculate the sequence number so that the file is only appended to when the sequence number is correct.

3 Design of the Server

The server first creates a UDP socket that listens to the given port number and all network addresses of the host. A recv timeout of 500 milliseconds is added via socket options, which causes all recv from calls to timeout after 500 milliseconds. The server has three states: connection set up state, packet transfer state, and closing connection state.

In the first state, the server waits for a SYN packet. After receiving a packet with the SYN flag set, the server responds with a SYN-ACK packet. When it receives the next ACK, the server will begin transferring the file.

At this point, the server is in packet transfer state. First, the server calculates the window, WND, from the minimum of the congestion window, receiver window, and (max sequence number + 1)/2. This window tells the server how many packets can be sent at a time. Then the server tries to send as many packets as possible, each with a payload size of 1024 bytes (for a total packet size of 1032 bytes). The server sends each packet at once and then waits for ACKs from the client. A timer is used to keep track of the latest received ACK. If the server does not receive an ACK for the oldest packet within 500 milliseconds, then timeout occurs.

When the server times out, it sets the slow start threshold (SSTHRESH) to half of the current congestion window, or 1024 bytes if half the congestion window is less than 1024 bytes. The congestion window is then reduced to 1024 bytes, and the server begins retransmitting from the last unacked packet.

After the server has finished transmitting the whole file, then the server then waits to receive all of the remaining ACKs. When all of the packets have been ACKed, the server sends a packet with the FIN flag set, signalling the beginning of the connection teardown. After sending the FIN, the server waits for the client's FIN/ACK response. If no FIN/ACK is received before the timeout, the server will retransmit the FIN. Finally, after receiving the FIN/ACK, the server will send one final ACK. After sending this ACK, the server then waits 3 RTO (1.5 seconds); if the server receives any packets during this time, it will check the flags to see if it is a FIN/ACK. If the packet is a FIN/ACK, the server will retransmit the final ACK. If the server does not receive anything for 3 RTO, then the server will close the socket and terminate.

4 Problems Encountered

We encountered several problems during this project.

1. We had trouble sending a basic message from the server to the client at first. However, this was fixed after updating our Vagrantfile.

2. We also struggled with how to implement the timeout functionality. At first we thought that the ACKs were not cumulative, so we were thinking about ways to set a timer for each different packet. We first thought that maybe we could have a queue of structs containing information about the packet number and a corresponding timeout time. However, this proved very difficult to implement. We later found out that the ACKs were cumulative, so we followed the approach outlined in the textbook, where only one timer is used. When a client receives a packet out of order, then it will simply retransmit the same ACK again and drop the erronous packet. The server then keeps a timer corresponding to the oldest unacked packet. If the timer expires before an ACK is received for that packet, then the server will go into timeout and retransmit that packet. If a correct ACK is received, then the server will restart the timer if there are any unacked packets left.

3. We had an issue with our transferred file being corrupted when the server had to retransmit packets. At first the issue was an off-by-one error; we had a variable that kept track of the location in the file of the oldest unacked packet. If the server timed out, then it would restart transmissions at that locatioin in the file. However, our variable was actually one byte ahead of the correct location, causing all retransmitted packets to be one byte ahead. After we fixed this issue, we realized that we had a problem with our sequence numbers. If a packet is lost, the client will continue to resend the same ACK until it gets a packet with the correct sequence number. However, our server window was being calculated wrong, and the sequence numbers would wrap around, such that the client would receive a packet with the correct sequence number, but not the correct payload. Thus the file would be missing a chunk of bytes in the middle. This was fixed by ensuring that our window including the number of unacked packets was smaller than (max sequence number + 1)/2.

4. We also had an issue where the server and client would enter an infinite loop when tearing down the connection. This was also due to another off-by-one error in our calculation of sequence numbers and ACKs. After fixing this error, the server and client terminated properly.

5 Testing

To test our implementation we sent several different types of packets from client to server:

- Basic TCP Header Packet: we sent over basic packet header to test that the UDP socket was sending and receiving properly. This also tested to see if our TCPHeader class would properly encode and decode byte streams read of the socket
- Files smaller than 1 MSS: next we tested sending a packet that contained a payload, in this case we used the README.md file in the directory. This tested that our code could handle a payload concatenated to a header and also that we can properly read in a file
- Files larger than 1 MSS: now we test files larger than one minimum segment size to determine if we can send multiple packets from the server to the client. At the same time we print out a status message per sent and received packet for both the client and the server to ensure that we are observering the proper sequence numbers and general behavior, such as sequence numbers wrapping around.

After sending the file, we tested using cout statements to see if the server was sending the FIN packet, receiving the FIN-ACK packet, sending the final ACK and then closing the socket. Then once that was done we used "diff" to compare the sent file to the receivied file for corruption.

The next step is to test the timeout:

1. We created a function that would be called on timeout and had it print out a message indicating that a timeout has occured.
2. After adding in timers that would reset after a succesful sent packet we test that no timeouts occur when there is no packet loss (timeout from either the recv call or our own timer) and no corruption in the file
3. Next we introduce X% packet loss and Y ms latency using options from the vagrant file to see if the basic timeout and the recv function will report any timeout occurring
4. Finally we use our fully implemented timeout function with X% packet loss and Y ms latency to test that we are observing the proper sequence number increase, the proper congestion window size increase and decrease on packet received and packet loss, the proper retransmission of lost packets (and that they have the "Retransmission") string printed out in the status, and lastly that the resulting file on the client side is not corrupted.

In general, we mainly used cout statements to determine if we were seeing the correct behavior, printing out values and deciding if those are the values we needed to see. For example we had file corruption, but this was due to the index we were using to read in the file not being set to the correct position on retransmission, and printing out the values helped narrow the problem down.

Overall, we determined that our server and client do transfer files properly. However, for a file of 4MB, it may take up to 20 minutes if there is significant packet loss. This is because if our SSTHRESH drops down to 1024, then the server will always be in Congestion Avoidance mode, making it difficult to transfer packets quickly. Some of the results of our tests are as follows:

- 5% packet loss: $\sim$2 minutes
- 10% packet loss: $\sim$6 minutes
- 15% packet loss, 4ms delay: $\sim$13 minutes

6 Contributions

Leslie Lam (804-302-387) and Dominic Mortel (904-287-174) wrote the web server. Kevin Huynh (704-283-105) wrote the web client. We all helped debug the client and server. Each wrote the corresponding sections of the report.