Project 2: Simple Window-based Reliable Data Transfer

Jeffrey Xu (404768745), Andrew Grove (304785991)

ABSTRACT

In this project, we used the UDP interface to implement a window-based reliable data transfer protocol not unlike TCP. We were unable to use high-level libraries in this project, so most of this project dealt with the creation of an additional header with the necessary information, i.e. ACK and sequence numbers and flags, and the creation of a selective repeat protocol to check said information. This selective repeat protocol was able to provide reliable data transfer over an arbitrarily unreliable connection by using ACK messages to ensure success.

High-Level Description

This project first used the UDP library in order to implement a simple client-server model. By using the UDP, although it does not confirm delivery, it is still able to send messages to and from the client and server. It does, however, due to the checksum already present in the UDP header, allow us to assume that a message, if sent, will have the correct values in them. This allowed us to create a simple header to attach to each message, which would be given to the UDP library so that it would be able to attach a UDP header to it. The message will then be sent through usual means until it reached the destination, whereupon the UDP library would remove the header, leaving the simple header attached to the message. At this point, all we had to do is inspect the simple header and break it down and parse the necessary information.

The header we developed is one that is quite simple. The header is 64 bits long: with the first 16 bits being reserved for the ACK, the next 16 bits reserved for the sequence number, the next 16 being reserved for the message's length, and the last 16 being reserved for any necessary flags, which for our purposes was ACK, SYN, and FIN. The 16-bit fields for the ACK and sequence number form unsigned ints able to represent up to 32767, beyond which it would reset back down to zero. In order to create this header, we utilized a character array, splitting up each 16-bit field into two characters of an 8-bit size, and storing them next to each other. In order to use flags, which were only one bit long, we had to use bitwise operators to adjust the corresponding character (in our case, byte 7 of the header) so that it would reflect these values appropriately.

Firstly, to implement the reliable data transfer, we needed the client and the server to conduct a handshake. We followed a procedure similar to that of TCP, utilizing a three-way handshake. The client would first send a message with our custom packet, where the SYN flag is 1. If the server is ready to respond, it will be waiting for any incoming connection with a packet like this, and will send back a message where the SYN flag and the ACK flag is 1. If the client receives this message, it will send back a message where the ACK flag is 1, and the ACK field will be equivalent to the sequence number that it had received. Additionally, the data part of this packet will hold the file path that the client desires. Each of the first two messages will be retransmitted if their corresponding ACK is not received within the RTO. The third message is not directly ACKed, but if the client receives another SYNACK it knows that its response must be retransmitted. Once the server receives this third message, it will see if this file path is valid. If it is not, it will send back a 404 message, which in our implementation is signified by a FIN message with a non-zero length. If it is, it will begin transmitting the file.

For small files of less than 1016 bytes long, i.e. those that can be sent in one packet, it is relatively simple. All we had to do was put the contents of that file into the data portion of our packet. Once the server sent this modified UDP packet, assuming that it arrives, the client simply has to open a file with the name received.data, and insert the data from the packet into said file. Afterward, the client simply has to send an ack to confirm that the client had received the file. If either the ACK or the message is lost while transmitting the server simply has to send this same packet again, once the timer times out.

Files more than 1016 bytes must be broken into multiple packets. The server attempts to read 1016 bytes from the file at a time, then stores that data into a packet and sends it out. If it reads less than 1016 bytes, it knows it has reached the end of the file. The client then receives these packets and organizes them by their sequence number, which prevents the data from being reordered if the packets arrive out of order. The client does not need to worry about how many packets there are or which ones should be sent; it merely replies with an ACK for every packet it receives. If the server does not receive an ACK within the RTO, it resends the packet. The client does not need to know whether its ACK has been received because if it is not, the server will simply resend the corresponding packet, giving the client another chance to send out the same ACK. Once all the packets have arrived, the client runs through the packets by sequence number and writes their data into the file received.data. Because every packet but the last one contains 1016 bytes of data, only the length of the last packet needs to be remembered. The client knows which packet is the last because it is the one with the highest sequence number that is not a FIN packet.

Speaking of FIN packets, the onus is on the server to close the connection because only the server knows how much data needs to be transmitted. Thus, once the file has been completely read and every every sent packet has been ACKed, the server transmits a FIN message and waits for a response from the client, retransmitting if none arrives within the RTO. When the client receives the FIN, it should have received every packet of the file because the server must have received every corresponding ACK. Thus, the client can immediately begin preparing for shutdown. The client replies with a packet with the FIN and ACK bits set and starts a timer equal to twice the RTO. If it did not wait, the FINACK could be lost and the server would be helpless because its retransmitted FINs would arrive at a closed socket and remain unACKed. The timer gives enough time for the server to time out and resend its FIN twice, so three FINACKs would have to fail in a row for the server to be left hanging with a half-closed connection. Unfortunately, this is not impossible, just improbable. While the rest of the protocol will work eventually so long as the loss rate is not 100%, this part is more likely to fail the worse the loss rate is. Once the connection is closed, the server returns to waiting for a handshake to start a new connection and the client consolidates the data into received.data as described in the previous paragraph.

To implement the sliding window, we used a queue containing pointers to all currently sent but not ACKed packets along with the time they were transmitted, sorted by the time since their last transmission, as well as a variable keeping track of which unACKed packet has the lowest sequence number. We used pointers to the packets so that packets could be created in each loop iteration without being destroyed when the loop restarted. Each time through the loop, the server listens for an ACK, and if one arrives, it searches through all the unACKed packets to remove the corresponding one from the queue and update where the beginning of the window is if necessary. The server can only send a packet if the current sequence number minus the beginning of the window leaves enough space in the window for that packet to be stored. After removing the ACKed packet, the server checks how old the top packet in the queue is. If this time exceeds the RTO, the top packet is popped off, resent, and pushed back into the bottom of the queue with a new transmission time.

Difficulties and Solutions

Some minor difficulties we had we had was with the construction of the header. Due to the limitations of the language, we were not able to access the bits through indexing. Additionally, because the packets are made up of characters, not unsigned, it became somewhat complicated to extract the necessary information and combine them so that it would become a 16-bit unsigned integer. To fix this, all we had to do however was cast the character into an unsigned character, shift it, then add the lower-order bits (which also have to be casted into unsigned), rather than only using bitwise operators.

Another problem we had was in the construction of a window. By regular means, waiting for an ack would stop the server from running, not using the window and instead acting like a stop-and-wait procedure. In order to remedy this problem, we used a similar approach to that of asynchronous IO. Instead of just blocking when the server is waiting for an ACK, instead it will continuously send messages until the poll signals that an incoming message is ready to be sent through or until the window is filled up. By doing this, regardless of when an ACK appears, the server will continue to send messages until the window is filled up, but as soon as an ACK does appear, the server will instead tend to that ACK, shift the window accordingly, and continue transmitting messages.

We also had some issues that involved continuing to read the file after reaching the end. This was diagnosed to be due to the main loop continuing to run as long as not all the packets had been ACKed. There was still room in the window for packets to be sent, so the program would read, fail its read, and continuously send empty packets. To fix this we set a flag to stop attempting to read the file once the end had been reached.

Lastly, we had some problems were the data field in our packets would be corrupted when reaching the final packet. This was due to the fact that the length of the packet was reset at the start of the loop. Thus, if it had a message shorter than the maximum message length, which only occurred on the last packet, but there was no room in the window for the message to be packaged into a packet, the loop would return to the top and forget the message's length. This was fixed by storing the message's length outside of the loop so that it would persist if the message could not be sent immediately and had to be saved for the next iteration.