Window-based Reliable Data Transfer in C++

Sean Custodio
503 895 980

Justin Wei
003 942 248

Implementation

We began our reliable data transfer implementation by creating a header file with constants and structs shared by both the server and client. The packet size, header size, ack size, and timeout are all defined as well as the file packet and ack packet structs. For the file packet struct, we decided to include the source port number, destination port number, packet sequence number, data size, and an array to hold the packet data(payload). In the ack packet struct, we include the source port number, destination port number, ack number, and the number of the next part of the file needed.

For our server implementation, we first take in a port number, window size , packet loss probability, and packet corruption probability as arguments. Then we create a socket and bind the socket to our local address. After binding, we listen for a request from a client. When a client request is found, we first set the timeout for all receive operations using the setsockopt function. Using the SO_RCVTIMEO option with the timeval struct, we set the timeout to be two seconds. Now that the timeout is set, we then receive the file request from the client. Upon successful reception of a request, we print out what message the client is requesting and proceed to call our parse request function. In the parse request function, we begin by opening the file and finding its size with the tellg function. If the file successfully opens, we print that we will begin sending to the client and call our send file function. If the file could not open, we print that the file was not found and send that to the client as well.

When sending the file, we chose to use the go back n protocol. In order to ensure reliable data transfer with our protocol, we keep track of the packet number we are sending, the last received ack number, and the next position in the file that needs to be sent. Then we use a while loop to keep creating packets until our file offset is greater than the file size. At each iteration of this loop, we reset our window offset and packets sent counter which initiates the start of a new frame to be sent under go back n. Next we have another while loop that makes sure our window position is less than the given congestion window and that we stop if all the file data has been sent. In this loop, we first set the appropriate data size needed for each packet. The data size for each packet will not be the max size if we are close to meeting our congestion window or we are near the end of the file data. After figuring out the data size, we proceed to set our packet header. Using the file packet struct, we set the source port number, destination port number, sequence number, and data size. Then we read the amount of data found earlier from the file and save in the packet data buffer. Once the packet is finalized, we send it to the client and print out the packet number we sent.

After sending a frame of packets, we use a for loop to wait for the same number of acks as packets sent. If we receive an ack correctly, we then simulate packet loss and

packet corruption. If randomly generated numbers are either less than our probability for loss or probability for corruption, we discard the ack and continue with the loop. If a later ack is received correctly, the ack loss does not affect us since the client is expecting a later packet. If the last ack of the frame is lost, we must retransmit from the position given by the last correctly received ack. If we do not simulate corruption or loss, we set our last acked counter to the ack received, change the file offset to the next expected position, and print out which ack we received. If an actual timeout occurs in the receive function, then we break out of the loop and continue sending packets from the packet number given by the last correctly received ack. After the file is completely sent, we print that we successfully sent the file.

For our client implementation, we take in the hostname, port number, filename, loss probability, and corruption probability as arguments. We then create a socket and set the socket timeout the same way as the server using the setsockopt function. Next we send a file request to the server and wait to receive a response that includes the file size or error message. If the file was not found, we print the message out to the client. If it was found successfully, we print that we are beginning to retrieve the file of n bytes and begin creating a new file by calling our get file function.

To begin getting the file, we first define a file offset to keep track of the file position and a counter for the next expected packet number. Similar to the server, we begin with a loop that makes sure that our offset is less than the total file size. Then we wait to receive a packet from the server. If the receive function times out, we just continue to wait until it succeeds. When we successfully receive a packet, we begin to simulate packet loss and packet corruption. If the random number generated is within our probability thresholds, then we print to the client that we discarded the packet. Since we discarded the packet, the server will timeout when waiting for an ack and retransmit the discarded packet. Now if we don't simulate corruption or loss, we then check if the packet we received is expected. The packet could be a duplicate due to simulated loss or corruption on the server side. If it is a duplicate or out of order packet, we discard it and print to the client that we discarded it. If it is the expected packet, we print out that we received the packet correctly. Now that we received the expected packet correctly, we set the ack header with the source port number, destination port number, next expected packet number, and the expected file position. If the packet we received previously was not the expected packet, we set the header with the same packet number we sent before. After the header is set, we send the ack to the server and print out to the client that we sent it as well. Next, if we received the expected packet correctly, we write its data out to the file. After the the client finishes retrieving the file, we print out that we finished downloading. But to account for the server possibly losing the last ack, we use a loop to keep receiving packets sent by the server and repeatedly send the last ack until it is sent successfully.

Difficulties

Simulating Packet Loss/Corruption:

At first we simulated loss and corruption for packets and acks in both the server and client. We realized that everything was being accounted for twice the amount needed. Now the client only simulates packet loss and corruption and the server only simulates ack loss and corruption. We also debated whether we should simulate loss or corruption for the file request sent by the client. But for testing purposes, it made sense to not account for that because if we make the probability 100%, we could never test for packet retransmission. The client would just repeatedly request to the server. We also had issues using random() here. The srand() seed was not generating random values fast enough. So many times the packet loss and corruption would happen in chunks. T

Implementing Go Back N:

For the first version of the implementation, we started out with a basic file transfer protocol that sent the whole file successfully at once without packetizing it. After this was finished, the design changed to sending the data in packets, but with the stop and wait protocol rather than go back n. The first version of this stop and wait protocol did work, but was not easy to change for go back n, so a whole new implementation was made. We had a lot of issues in making sure that the file that was sent to the client was the same as the original server. There was a lot of math involved in making sure that the bytes sent from the server would accomodate the cwnd. After this new implementation for stop and wait was successful, the necessary changes were made to make it go back n. We send a frame of packets to the client and wait for the frame to be acknowledged. The size of our frame is given by the congestion window. If any packets are unacked within the frame, a new frame is created starting from the last unacked packet.

Timeout for last unpacked packet:

When considering how to implement timeout for sockets, we chose between using the select function and the setsockopt function. We decided to use the setsockopt function so that we could use the SO_RCVTIMEO option and change the socket layer option directly. This actually worked well with the majority of our code, but we had issues when the server was losing the ack for the last packet. What would happen is the server would continually retransmit the last packet when the client already has the entire file. We fixed this by adding in a loop at the end of the client that continually acks until the server stops retransmitting.