UCLA CS 118 - Computer Network Fundamentals

Project 2: Window-based Reliable Data Transfer over UDP (Go-Back-N)

Ming Wei Lim	UID# 304147563	SEASnet: mingl
Caitlin Torsney	UID# 403905022	SEASnet: caitlin

Go-Back-N

For GBN, the sender can send out a certain number of packets within a window. The receiver must receive all packets in order. If it receives a packet that is different from the in order packet it was expecting, it will just resend the previous ACK. For the sender, the window can only be moved up if it received the ACK for the first packet in the window.
In our implementation, the sequence numbers and ACKs were in terms in bytes of the data. If the initial sequence number is 0 and the sender sends 1500 bytes, the receiver will ACK 1500 signifying that it has received bytes 0-1499 and it's expecting the next packet to begin a byte 1500.

Packet

In addition to holding the data to be sent, our packet also held several useful variables. First, it will hold a sequence number that will signify the first byte of the data. The length of the data in terms of how many elements of the array it takes up also is stored. This is used for fread and fwrite. The actual size of the data in bytes is also stored to help the receiver compute the ACK number. Finally, 3 switches are implemented to determine what kind of packet is being sent. These switches determine whether the packet is a filename, an ACK, or the final packet that will be sent.

Client

In the client, once the UDP connection is set up, an initial packet is constructed to send the requested filename over to the server. The filename switch is turned on in the packet to tell the server that it's receiving a filename. Next, the client enters an infinite loop where all the GBN protocol will be carried out. Because the network is usually extremely reliable nowadays, we mimicked data lost and corruption. The probability of packet loss or corruption can be set by entering a number between 0-100 in the beginning of the code. If a packet is determined to be lost, the client just completely ignores the received packet. If the packet is determined to be corrupted, a packet is formed that contains the ACK for the most recently received in order packet. If there is no packet corruption or loss, the client then checks if the sequence number of the received packet matches the expected sequence number. If not, then the received packet is ignored and the client sends an ACK for the most recently received in order packet. If the packet contains the expected sequence number, then the expected sequence number is incremented by the size of the data and it'll also serve as the ACK number for the response ACK the client will send back to acknowledge that it received the correct packet.

Server

In the server, once the UDP connection is set up, the server waits for a request from the client. The server is non-blocking by using select(). When it checks for requests from the client it does not wait infinitely and timeout mode can be used. Once a request for a file comes from the

client the server checks to see if it has that file and if so, it opens it and divides it into packets of size 512 bytes each. The server will then populate the window until it is full. Once the window is full the packets are sent to the client in normal GBN protocol. When a packet is sent a trigger time is specified. If the current time ever reaches the trigger time then timeout mode is entered and all unACKd packets in the window are resent. Each time an ACK is received from the client the window is updated and the trigger time is increased. For packet loss or corruption on the server side the server checks that all packets with a seqnum <= the ACK seqnum are removed from the window. The server also keeps track of the number of distinct ACKs that have been received and the sending of packets is terminated once the number of ACKS = the total number of packets used to send the file. Once all the file packets are sent a final tear down packet is sent and the request file resource is closed.

Messages

The messages printed by both the server and client are used to tell the user what's going on as the GBN protocol is executing. On the client side, a message is printed whenever a packet is received or sent. The details of the message will let the user know whether the packet received was corrupted, lost, or out of order. Sent packets will produce messages detailing the ACK number that is being sent out. Similarly, for the server, messages are printed out when a packet is sent or received. Received packets will produce messages printing the packet's sequence number, and the user will also be notified that the window is being moved up if the correct ACK is received. If an incorrect ACK is received, a message will be printed telling the user that timeout mode is being entered. For sent packets, a message will be printed showing the packet's sequence number. Special messages such as packets being resent due to a timeout can also be printed.

Timeouts

The timeout length for our server is 0.1 seconds. Each time a packet is sent from the server a trigger time is set to = (time of send + 0.1) seconds. And for each sequential pack that is sent the trigger time is updated to = (time of new send + 0.1) seconds. If the current time is every > the trigger time the timeout protocol of GBN is executed.

Difficulties

One of the main difficulties reached during this project was implementing the timer. At first, we used the time() function to calculate elapsed time. However, the timer was never triggered for this implementation, and it took a while to figure out why. We didn't realize that time() had a maximum resolution of seconds. We needed a finer resolution because our program performed way to fast to be timed in seconds. We changed our implementation to use gettimeofday() where we could specify the resolution to be microseconds. This turned out to be way more accurate than time() and the timer worked correctly. However, we then realized that using

microseconds was too fine of a resolution when transmitting large files. This is because microseconds capped at 1 million, so it would reset everytime it reached 1 million. We reverted back to using seconds and set the timer to less than a second so the program would not take forever to run.

Another difficulty we encountered was that recvfrom() would stall on the server side whenever the client didn't have anything to send back. To fix this, we implemented nonblocking mode for our socket using select() so recvfrom() would timeout if the client didn't send anything for a while.