CS 118: Computer Network Fundamentals

Professor Lu

Project 2

Roger Chen (UID 504043927)

Kailin Chang (UID 503999157)

**High-Level Description**

We defined our custom data structures and global constants in packet.h. The global constants we defined are:

const int MAX\_PACKET\_SIZE = 1000

const int HEADER\_SIZE = 8

const int PACKET\_TIMEOUT = 100 // this is in terms of ms = 0.1s

In addition, we created a custom struct Packet to represent a packet. Within each instance of packet, we have following fields that make up the header information:

bool type // returns true for message, false for ACK

bool last\_packet // returns true if packet is last packet

int seq\_sum // indicates sequence number of packet

int packet\_num // indicates packet number of packet

int data\_length // indicates length of the data in each packet

In addition to the header information, each packet also has an array called data that holds the information to be sent with each packet:

unsigned char data[MAX\_PACKET\_SIZE - HEADER\_SIZE]

The window-based protocol we implemented is Go-Back-N. We split our implementation into the sender and receiver side. One important thing to note is that even though the command window is given in terms of bytes, our command window is in terms of packets, and our command window is calculated by dividing the given command window (in bytes) by our MAX\_PACKET\_SIZE (in bytes) to get our command window size (in packets). On the server side, the general overview of our implementation is the following:

1. Set socket and populate server address
2. Bind server to socket
3. Get file request
4. Create initial ACK for file request: send ACK with sequence number 0 if the file is valid, -1 if the file is invalid
5. Split received file data into packets: packets are pushed onto a vector called packets
6. Send initial messages and push each message’s sent time onto a vector called sent\_times
7. Wait to receive ACKs and check timeouts
   1. You first check timeouts: if timeout, resend all packets in window
   2. Then, you check for received ACKs: if the packet number of the ACK is greater than or equal to the base, then we update our base to the received ACK’s packet number + 1 and send all packets that have not been previously sent but now fall in the updated window

On the receiver side, the general overview of our implementation is the following:

1. Set socket and populate server address
2. Send initial request for file
3. Received ACK for initial request from server
4. Receive and send ACKs for subsequent packets
   1. If packet is received in order, send ACK
   2. If packet is received out of order, send ACK for most recently received in-order packet
5. Send the ACK for last packet 10 times to ensure that server does not have the last ACK dropped
6. Write packet contents to file

To implement timeouts, we used a vector<timeval> to store the times of each packet being sent and compared the current time with the oldest time located at the front of the vector. If it was greater than our timeout threshold, we clear all the times in the vector and resend all packets in the interval of [base, next\_packet\_num – 1] and update this vector with the new updated times. Under normal conditions, the oldest time gets removed from the vector upon receipt of a successful cumulative ACK.

**Difficulties Faced And Solutions**

One of the first difficulties we encountered was regarding File I/O. We originally stored the data as a string, but realized calling sizeof()would not account for the size of the data stored in the string. We then changed it to a char[] , but then had trouble with binary files so we researched that data from binary files should be stored in unsigned char. Furthermore, we had an issue regarding whether or not we needed null-bytes. We implemented different approaches and concluded that our program did not need null-bytes to read packets or write to files.

Another difficulty we faced was how to design our command window. Since the command window was given in terms of bytes, we were not sure how to convert that into packets efficiently. At first, we tried to calculate the window size based on how many could fit in the window, which means the window size would vary because the packets are not always the same size. However, we realized that that method was inefficient since the window slides and that would involve a lot of possibly complicated calculations. Thus, we decided to set a fixed window size that is calculated by dividing the given command window size (in bytes) by the maximum packet size (in bytes). Even though that method could be inefficient because it does not always use up the entire command window, it is much simpler to implement when the window starts sliding.

The last difficulty we encountered was implementing timeouts. We originally used time(NULL) to keep track of the time and difftime() to determine if a packet times out, but realized that the accuracy is only to the magnitude of seconds. We decided to use gettimeofday() instead, which returns a struct timeval that contains both seconds and microseconds; this was suitable for our purposes because our timeout window is only 100 [ms].