COM SCI 118
Songwu Lu

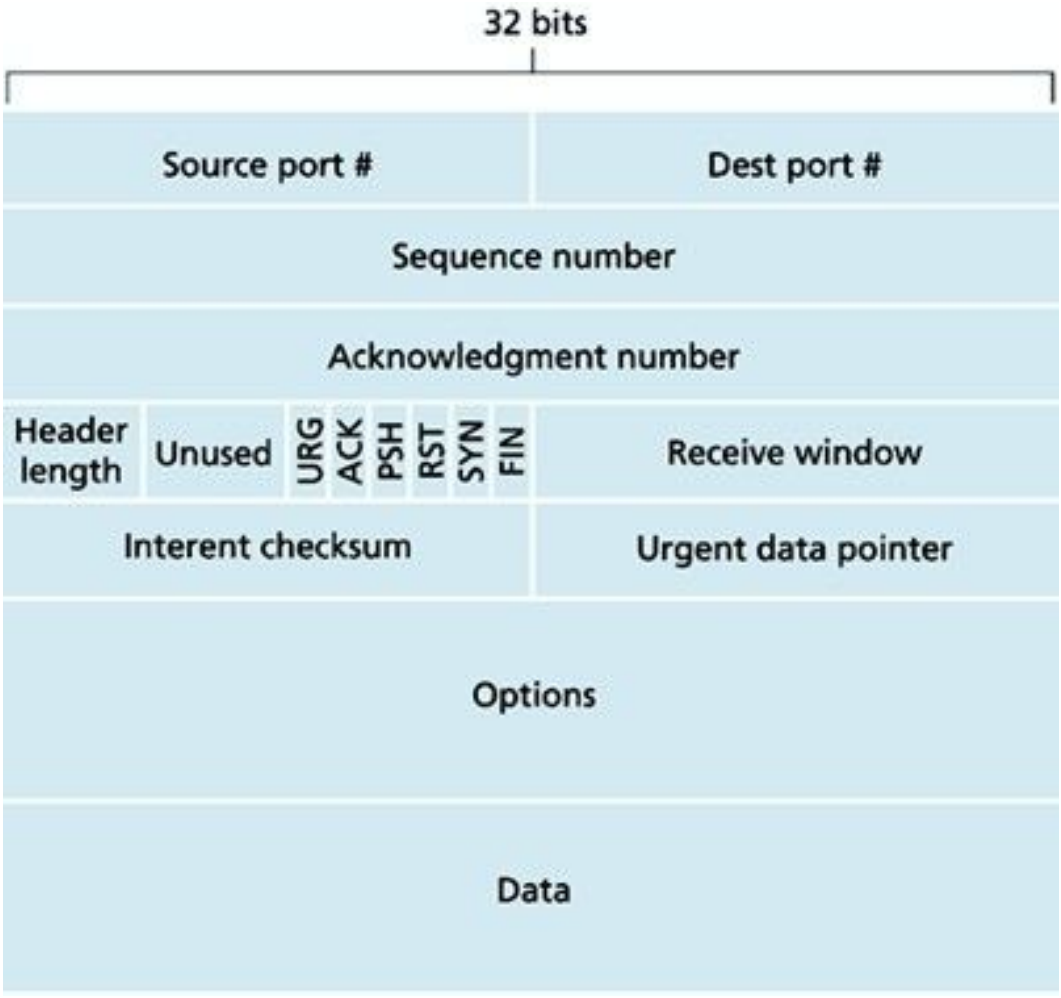
Project 2 Report

Jai Srivastav — 604605757
Jeffrey Qiu — 804808179

Implementation Description

In this project we implemented a simple congestion window-based protocol that was based on the Selective Repeat protocol as specified in the textbook. We had a server communicate with a client by means of the User Datagram Protocol (UDP) socket, which does not promise data delivery, and then enabled the server send the client application a file upon request.

We followed the recommendation outlined in the project description and approached the server and client development in incremental steps. We started with the assumption that there would be no packet loss, and established a connection with a three-way handshake. Our header format is 20 bytes and consists of all of the standard header fields as illustrated in the textbook. We created various helper functions located in *utilities.h* such as get4Bytes(const char buf[], const int offset) and setBit(char buf[], const int bit, const char val) that utilized various Header and Bit offsets that corresponded to each of the illustrated fields below in order to facilitate with populating a header prior to the sending of a packet or the extraction of various header fields for packet processing.



With connection established between the client and server we were then able to implement the Selective Repeat events and actions. When we received a packet with a sequence number within the range of [window_base, window_base+N-1] where N is our window size we would buffer the packet inside a cache. If the sequence number of the packet equaled that of the window_base then we would write the payload to a file named “received.data” followed by the payload of a consecutively numbered packet if it were previously received and buffered. The receive window would then be moved forward by the number of packets that had been written to the file. If the packet had a sequence number within the range of [window_base-N, window_base-1] then an ACK will be generated and sent to the server, despite it having already been acknowledged.

On the other hand, the server will continuously read and send chunks of a specified file until it reaches an EOF. When a file chunk is read, the next sequence number that becomes available will be packetized with the data chunk and sent to the client if the number is within the sender’s window, otherwise it will be buffered or retransmitted at a later time. Each sequence number that had been associated with a packet also has its own logical timer, and if the timer elapses time equal to or greater than the retransmission timeout value then the packet will be sent again in order to protect against lost packets. When the server receives an ACK from the client then the corresponding packet will be marked as received if it is in the window and if its sequence number is also equal to the server window_base then the window_base will be moved forward until the unacknowledged packet with the smallest sequence number. New packets that are untransmitted which lie within the new placement of the window will also be transmitted, and the process starts again once more.

It was somewhat challenging to handle various edge cases, chief among them being the handling of the final payload that would likely be less than the full size of a packet payload. Our initial naive approach was inelegant involved a flag to check whether the packet payload was less than the previous payload on each iteration of our “polling” while loop and whether a FIN message came immediately after. However, we then discovered that we could seamlessly integrate the handling of a smaller payload without a flag by means of comparing the number of bytes received from recvfrom().

Overall, most of the issues were resolved in a reasonable amount of time when we committed pencil to paper and drafted our application architecture prior to approaching development of the server and client in incremental steps.