# Project 2 Report
# Simple Window-Based Reliable Data Transfer

Hermmy Wang
704978214
Keyu Ji
704966744

## 1. Introduction

TCP and UDP are two protocols to implement the transport layer of a network packet. These protocols are the reason why multiple applications can use the same network connection simultaneously. The purpose of this project is to implement a TCP connection with UDP in C programming language. While UDP has smaller packet size and does not require creating a connection before sending out data, it does not guarantee in-order data delivery, does not recover packet loss, and has no congestion control. On the other hand, although TCP has more overhead such as establishing three-way handshake with the server before sending data than UDP, it guarantees reliable data transfer and offers congestion control via algorithms like slow start, congestion avoidance, and fast retransmission.

## 2. Implementation description

### a. Window-based protocol

In this project, we are required to write a server application and a client application.

The server application opens UDP socket and saves the client's files (up to 100MB) in their arriving order. The saved file needs to handle the case where a signal interrupts the transmission. Also, the connection times out after 10 seconds (10000ms) no matter whether it has received any file successfully from the connection. Upon receiving the request from the client, the server should respond to the client with SYN and ACK flag set.

The client application opens UDP socket, sends files to the server, and also implements congestion control. Upon the connection is initiated with 3-way handshake, the client should first send a packet with SYN, a random sequence number, and a 0 acknowledgement number; then it waits the server's response; lastly, it sends another packet with ACK flag to start sending the real data. The client times out after 10 seconds when it receives no packets from the server. After the data transmission is completed, the termination steps are sending FIN flag, waiting for ACK flag from the server, waiting 2 seconds for the server's FIN flags while responding with ACK flag upon each server's FIN.

The congestion control allows the client to send data of the initial window size before waiting for the first acknowledgement. After the window size decreases below the threshold for the first time, increment it with 512 bytes (slow start). If the window size is above the threshold, increment it with 512*512/window size. In the case of timeout, the threshold is set up the maximum value between half the window size and 1024; the window size is reset to 512. Slow start is performed after each retransmission. Note that ACK from the server indicates the sequence number of the next packet from the client. After receiving 3 duplicate ACKs in a row, the threshold is set up the maximum value between half the window size and 1024; the window size is reset to the threshold+1536. After retransmission, window size is set to the threshold and enter congestion avoidance. If there is another duplicate ACK arrives, increment the window size by 512 bytes. Then send a datagram if allowed by the window size. Upon a new ACK arrived, reset the window size to the threshold.

**b. Header format**

We have designed our own format of the protocol header for this project. The header consists of a sequence number, an acknowledgement number, an acknowledgement flag, a SYN flag, a FIN flag, and the data size. The flags are characters, and the other parameters are of short type. Since the project requires the header length to be strictly 12 bytes, we pad the rest bytes with zeros.

**c. Messages**

The messages are printed out to the standard output in the following format:

Receive a packet:

```
RECV ⟨SeqNum⟩ ⟨AckNum⟩ ⟨cwnd⟩ ⟨ssthresh⟩ [ACK] [SYN] [FIN]
```

Send a packet:

```
SEND ⟨SeqNum⟩ ⟨AckNum⟩ ⟨cwnd⟩ ⟨ssthresh⟩ [ACK] [SYN] [FIN] [DUP]
```

Since the server does not need to implement congestion control, the field <cwnd> and <ssthresh> will be zeros for the server.

**d. Timeout**

A timer is added on each send and un-ACKed packet. Once the timer times out after 10 seconds, we initiate retransmission on this lost packet.

## 3. Difficulties

a. Data structure for the packet

We wanted to design the header format as an array of data. Each field consists of one byte. All the fields form an array of 6 bytes of valid data, while the rest is padded with zeros. However, later we realize this method cannot work out because we need more than one byte to store numbers. Therefore, our final data structure for the packet is a struct in which six fields, a sequence number, an acknowledgement number, an acknowledgement flag, a SYN flag, a FIN flag, and the data size, are wrapped together in a data structure called “header_proto”.

b. Congestion Control
   Implementing congestion control requires the client to send multiple packets before receiving any response from the server. We have to control the loop and initiate reading from the server only after sending the window-sized data.

c. Timer:
   To implement timeout, we choose the function in the <sys/select.h> library:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds, struct timeval *timeout);
  struct timeval timeout;
  timeout.tv_sec = 10;
  timeout.tv_usec = 0; // wait for 10 seconds
  int received = 0;
  fd_set inSet;
  FD_ZERO(&inSet);
  FD_SET(sockfd, &inSet);
  received = select(sockfd+1, &inSet, NULL, NULL, &timeout);
```

   The function will return a value greater than 0 if the time out does not occur.