# CS118 Project 2 Report

Chayanis Techalertumpai, Galen Wong

404798097, 104939937

## 1 Introduction

This report outlines the implementation of TCP and difficulties our group faced in this project. This project aims to deepen our understanding of the underlying protocols in TCP as well as how TCP deals with packet loss.

## 2 Implementation

We implement some functions of reliable data transfer protocol to using UCP in C and C++.
To compile the program: `$ make`
To run the program:
Client side: `$ ./client <HOSTNAME-OR-IP> <PORT> <FILE_NAME>`
Server side: `$ ./server <PORT>`

### 2.1 Header format

We created a struct type called `header` to define a packet header. The struct header contains

- `int16_t seqnum`, sequence number
- `int16_t acknum`, ack number
- `char flags`, character sequence for ACK, SYN, and FIN flags. Since `char` type is 8 bits but we only need 3 bits for our flag(s), we leave the first 5 bits unused.
- `char pad[7]`, an array of characters that are initialized to be all zeros. This provides padding to make a header 12 bytes.

To obtain a character sequence of flags, we assign macros to ACK, SYN, and FIN and `OR` them together.

## 2.2 Messages

Server and client log to `stdout` for every packet received or sent with the following format:

- `RECV <Seqnum> <Acknum> <cwnd> <ssthresh> <ACK> <SYN> <FIN>`
- `SEND <Seqnum> <Acknum> <cwnd> <ssthresh> <ACK> <SYN> <FIN> <DUP>`

The `SEND` message has an additional `DUP` bit which indicates if this packet is a result of retransmission.

## 2.3 Connection class

We create a `Connection` class which deals with handshake, receiving and transmitting packets, and handwave on the server side. The reason for this is to provide another level of abstraction which makes it easier for us to maintain connections in server. Each time a SYN packet arrives on the server side, the server creates a `Connection` object, proceeds with the handshake protocol and takes appropriate actions. Once the handwave is done, the server deletes this `Connection` object and moves on. In other words, the server program only manages connections and takes the data read by `Connection` and writes it into appropriate file.

## 2.4 Timeouts

There are two timeouts in this program.

### 2.4.1 10s timeout

The 10s timeout occurs when:

- the server establishes a connection but receives no data from the client after 10 seconds since the connection establishment
- the server establishes a connection but receives no data from the client after 10 seconds since the last successful data packet transmit
- client receives no packets from server for more than 10 seconds

When this timeout occurs, the relevant side exits the program with an exit code 1 and an error message `Error: 10 seconds timeout`.

### 2.4.2 RTO timeout

RTO is fixed at 0.5 seconds. The RTO timeout occurs when client has not received an appropriate ACK from server for more than 0.5 seconds. The timeout triggers packet retransmission and appropriate congestion control actions.

## 2.5 Timer

`Timer` is used in both `Client` and `Connection` in keeping track of the timeouts for each packet. `Client` has two timers: `time10` for the 10s timeout and `timer` for the RTO timeout. Each `Connection` has `time10` timer (and `timer` which is used only in handwave).
When a client sends a packet, client's `time10` and `timer` is activated while the file has not been exhausted or the size of `cwnd` is non-zero. These two conditions are specified to avoid a situation where the timer is unnecessarily triggered when the file is already exhausted. Client's `time10` is stopped when it receives any packet from server. Its `timer` is stopped when the corresponding ACK of the sent packet is received.
This follows the TA's suggestion to keep track of only one timer for RTO and constantly activate and deactivate it as a way of optimizing the program. Keeping track of one timer per each packet results in overhead which can reduce program's performance.

## 2.6 `cwnd` buffer

`cwnd` buffer is implemented by a vector of `<struct header, string>` where we store the header and body of a packet. Every time a client sends a packet, it inserts the packet into `cwnd`. Whenever an ACK is received, the client checks if the ACK is in order by comparing it against the first packet in `cwnd`. If yes, the relevant packet is removed from `cwnd` and slow start or congestion avoidance is performed. Otherwise, the number of duplicate ACKs is incremented.

## 2.7 Slow Start, Congestion Avoidance/ Slow Start

- Initially, `cwnd` is set to be 512 and `ssthresh` is set to be 5120.
- After handshake, the client starts sending packets in which the number of packets is controlled by `cwnd`.
- After each ACK is received, the client uses Slow Start if `cwnd` > `ssthresh` by incrementing `cwnd` by 1 packet (512 bytes) and Congestion Avoidance if `cwnd` ≤ `ssthresh` by incrementing `cwnd` by 1/`cwnd` packet.
- A packet loss is detected when the number of duplicated ACKs is incremented to 3 or when the RTO timeout is reached.
- When packet loss occurs, client sets `ssthresh = max(cwnd/2, 1024)`, set `cwnd` = 512, and retransmit data after the last ACK byte. Slow Start or Congestion Avoidance is performed after retransmission.
- `cwnd` is incremented by 1 packet every time additional duplicate ACK is received.
- Cumulative ACK is used during packet loss.

### 2.8 Server-side Buffering with `rwnd`

Instead of Go-Back-N with server side buffer as size 1, we choose to implement a full on buffer to reduce the number of retransmits on the client side. Theoretically speaking, the `rwnd` has size $\infty$ therefore window size is only bounded by `cwnd`. We maintain a vector/list of packets. Upon receiving a packet, we buffer it up. Then, for each element in the list of buffered packet

1. We check if the packet has the expected sequence number.
2. If yes, we return the data to Server to write to file, increment expected sequence number, remove the packet from the buffer, go back to step 1 for checking the rest of the array
3. If no, we simply buffer it up and does not return any data.

This seems like a dangerous algorithm, this can potentially goes up to $O(n^2)$ time if everytime we remove the last element in the list. We could have maintained a more complex data structure, such as a hash map that maps SEQ number to packet, or a list sorted according to SEQ number. However, since `cwnd` (which bounds the window size) is bounded by 20 packets, this operation is safely bounded. Therefore, a simpler algorithm helps since managing other states is already difficult enough. There is an hidden bug in this approach. We will get into that in section 3.1.

## 3 Difficulties

### 3.1 Problem of Lost ACK

We imagine that `cwnd` has 5 packets: [1, 2, 3, 4, 5]. Say that server has received and buffered 4 of them: [2, 3, 4, 5]. It should send 4 duplicate `ACK1` to client. And client should retransmit packet 1 upon 3rd duplicate ACK. If server receives it, it will send back `ACK6` and buffer will be emptied. Imagine that `ACK6` is lost. Then, client will retransmit packet 1 upon RTO timeout. However, server will buffer it up also causing packet 1 to stay in the buffer forever, until the ACK number eventually wraps around. Now, we have 2 packet 1's in the buffer and of course our program will be confused. Initially, we think that there is no way around this situation. Since we cannot directly compare the sequence number in the `s1 < s2` manner (sequence number can wrap around), this problem has to be left unsolved.

Solution: We realize that there is a maximum `cwnd` limit, which is 10240. The maximum sequence number is 25600, which is sufficient for us to determine if sequence number comes before or after. We wrote a helper function in the following manner.

```
bool is_s1LTs2(int16_t s1, int16_t s2) {
    if (s1 < s2) return true;
    if (s2 < 10240) {
            int temp = 25600 - 10240 + s2;
            return s1 > temp && s1 <= 25600;
    }
    return false;
}
```

Now we can choose to ignore sequence numbers. Therefore, our implementation can work for large files with loss.

## 3.2 Problem of Frozen Timer

Imagine a scenario when the file has been exhausted, but `cwnd` is not emptied yet because we have not received the ACK for the sent packet. When an ACK is received, we will disable the timer for RTO. However, at this point, we no longer have packet to add to `cwnd`. Therefore, timer for RTO is never activated, and no packet ever gets transmitted. The states of both the server and the client are now frozen. Eventually, the 10s timeout arrives and kill the connection. This is bad for reliable transmission.

Solution: We change the RTO timer start condition. The timer should be activiated whenever there is still data in the `cwnd` buffer. Then, this solves the above mentioned error.