# CS118
# Project 2 Report

# *Simple Window-Based Reliable Data Transfer*

**Abdullah Albanyan - 304479543**
**Alvin Corhodzic - 804415450**

## 1. Implementation Description

- The implementation is split into three main classes, **Packet, Server,** and **Client.**
- A Packet is created by specifying a header, which is specified as follows, as well as an optional payload:

```
struct PacketHeader {
  uint32_t seqno = 0;
  uint32_t ackno = 0;
  uint16_t flags = 0;
};
```

- Both Server and Client have the methods sendPacket() and receivePacket().
- sendPacket() encodes the packet into a byte array and sends it to the specified port and hostname.
- receivePacket() waits for a packet, with an optional parameter specifying how long to wait before timing out, where it will just return -1. Typically, this function is always called with a timeout of 500ms.
- All constants and class definitions are stored in rdt.h.
- We implement a connection handshake by letting the client send a SYN, the server sends a SYNACK, then the client sends an ACK with the filename included. The server then sends the first cwnd chunks of the file with an ACK for the filename. If the ACK is not received by the client, it will retransmit the filename.
- On the server, we have a function called sendFileChunk(). This function reads the next MAX_PKT_SIZE bytes from the specified file sent by the client, creates a packet, adds that packet to the current window, and sends it immediately.
- Since this is an SR protocol, we make use of individual ACKing. Each packet sent increments the sequence number by the size (in bytes) of their payload. The ACKs that are sent in response will have the same ACKno as the SEQno of the packet they are ACKing.
- When a packet is acked, we check if its ACKno = baseseqno of the window. If so, we delete all ACKed packets from beginning of window to first unACKed packet, and update baseseqno.
- Since this is an SR protocol, we maintain a per-packet timeout. This is implemented by storing a “timestamp_to_die”, or timeout_time for each packet, then setting the timeout of receivePacket() to the timeout of the packet which will timeout the soonest during the event loop.This ensures that we can retransmit packets as soon as they timeout, while using only a single thread per client.
- On the client, we ensure that packets are written in-order and only once to file by maintaining a queue of sequence numbers of the last MAX_SEQNO/MAX_PKT_SIZE packets. If a received packet’s seqno matches one of those seqnos, it will not be written to file (but still ACKed). Out of order packets are handled by maintaining a receive

window on the client-side. If a received packet's seqno is equal to the baseseqno of the receive window, the receive window is advanced and all consecutive packets are written to file immediately.

- We implement disconnection handshake by having the server send an empty FIN once the file has finished transfer. The client responds with a FINACK, then initiates a timed wait (double the normal timeout time of 500ms) before closing. The server responds to the FINACK with an ACK, and immediately closes, cleaning itself up. If the client receives this ACK before the timed wait ends, it will immediately close, cleaning itself up.
- We implement congestion control using standard TCP congestion control protocols (i.e. we store a congestionstate which can vary between SLOW_START, CONGESTION AVOIDANCE, and FAST_RECOVERY, and update cwnd, ssthresh, and congestionstate based on various events like dupacks or timeouts.

**2. Difficulties Faced**

- The main difficulty we faced was reasoning about sequence numbers. Several bugs were experienced due to improperly incrementing or storing sequence numbers.
- In addition, correctly implementing the logic behind the receive and sender windows was also tricky, partially because they involved comparing sequence numbers.
- Some difficulties were faced with buffer overflow issues causing data corruption in received packets.