# COM SCI 118 Spring 2019
# Computer Network Fundamentals

*Project 2: Simple Window-Based Reliable Data Transfer*

Jesse Catalan (204785152)
Ricardo Kuchimpos (704827423)
University of California, Los Angeles

# Simple Window-Based Reliable Data Transfer


## Abstract

We implemented a simple window-based RDT protocol using BSD sockets over UDP. In particular, we handle packet loss and assume that packets experience no reordering or corruption. This can be accomplished by using a sliding window and maintaining the state of the congestion window and Slow Start threshold. Once implemented, we were able to successfully transmit binary files from the client to the server, even in the presence of packet loss.


## 1. Background

### 1.1. TCP

The transmission control protocol (TCP) is a connection-oriented transport protocol operating in the transport layer of the internet protocol stack. Its promises for guaranteed delivery of accurate data make it useful in applications where we want no data losses or corruptions to occur. This makes it a better candidate in sharing data over the Internet when compared to its counterpart, UDP.

To understand the benefits of TCP, one must first review the properties of UDP. UDP is a simple, connectionless transport protocol. When a client wishes to send data to a server, the client and server do not establish a connection. Rather, the server listens for data on a specified port, and the client sends data to that port. Upon receipt, the server is only able to detect bit errors, but does nothing about it. Upon finishing transmission, the client closes its side of the connection and the server continues to listen for data over its port. And that's the extent of UDP's promises. There is no guarantee on packet order, packet delivery, or packet retransmission. For applications where we want secure delivery, TCP builds upon UDP's structure.

To provide delivery guarantees, TCP establishes a connection with the server to allow for bidirectional data flow. To do so, the client initiates a three-way handshake. This handshake is comprised of the client's SYN message, the server's SYNACK response, and the client's ACK confirming the server's SYNACK. This bidirectional communication introduces server acknowledgements, which are used to inform the client of successfully sent packets. When the client data is split into packets, each packet is given a sequence number to inform the server of what data is being sent. When the server confirms that the data sent was the one it was expecting, it sends an acknowledgement number with the next byte of data expected. This solves two issues with UDP. First, and of importance in this project, in the case of packet loss, the server will detect that the client sent a packet that it was not expecting next, an implication that the packet that the server was expecting was lost. To remedy this, the server sends a duplicate acknowledgment back to the client, which informs the client that the server is expecting a specific packet. Upon timeout of that packet, the client can then resend the lost packet. The second issue TCP corrects is the reordering of packets. When the server receives an out-of-order packet, it can buffer that packet until it receives the expected packet. Upon receipt of the expected packet, the server can then acknowledge both packets by using a cumulative acknowledgement.

Another way TCP builds upon UDP is its flow control and congestion control. The former prevents the loss of packets when the server's buffer is full, while the latter attempts to prevent packet loss, delays, and other costs of congestion by dynamically reducing the number of packets in transmission.

### 1.2. TCP Congestion Control

Network congestion occurs when there is too much data being transmitted over the network at once by multiple users. This results in both packet loss and packet delays. TCP handles this by having each user "test" the network's bandwidth by sending packets at

an increasing rate. When there's loss TCP congestion control protocol forces users to slow down until there is no packet loss. By doing this, TCP can ensure that users send at approximately the same rate and the network's full bandwidth is utilized.

When a client wishes to deliver data to the server, the network is able to move to the equal share and full bandwidth utilization (equilibrium) point through the AIMD (additive increase, multiplicative decrease) rule. This rule entails increasing the number of packets allowed to be in transmission after every successfully acknowledged packet and halving that number when there is packet loss [1].

## 2. Design and Implementation

### 2.1. Slow Start

When initiating connection between the client and server, there are two values we set to keep track of congestion. The first, the congestion window (`cwnd`), is the value we update upon receiving ACKs. The other, the slow start threshold (ssthresh), is the value used to determine when to switch from the slow start algorithm to congestion avoidance. The `cwnd` is always set to transmit a single packet of data at the beginning of any connection. In our implementation, ssthresh was set to 5120 bytes, or equivalently, since each packet has a maximum payload size of 512 bytes, 10 packets.

Each successfully sent packet (indicated by a valid ACK) would increase `cwnd` by one packet. As we begin to send more packets in parallel, the `cwnd` increases at a exponential rate, doubling after onee RTO if there is no loss.

### 2.2. Congestion Avoidance

Congestion avoidance occurred when `cwnd` reached the value of ssthresh. In this case, we increase `cwnd` by `(512 * 512) / cwnd`, as given in the spec. We continued to do this until loss was discovered, at which point we set ssthresh to the max of the current `cwnd` and 1024 bytes, or equivalently, two packets. We then reset `cwnd` to 512 bytes, or one packet. We then transfer back to the slow start algorithm.

### 2.3. Header Format

In implementing our packet class, we were given the restriction of a 12 byte header length. To do so, our packet class had a member variable consisting of a struct representing our header. This struct had a char representing all three of our flags (SYN, ACK, FIN) and two unsigned shorts representing our sequence number and ack number and a gap filling the remaining 6 bytes (since structs automatically offset the char to align with the unsigned shorts).

The rest of our packet class contained the payload and an int to represent the payload size. Because this resulted in a packet greater than the maximum packet size, we did not send our int when assembling the packet into a buffer. This ensured that we only sent the maximum of 524 bytes.

### 2.4. Messages

Messages were created from instances of our packet class. In order to send an instance of a C++ `Packet` object, it had to be transmitted over the connection as a byte stream. This can be accomplished by our method `Packet::AssemblePacketBuffer()`. The packet consists of the header and the payload, which can contain up to 512 bytes of data.

### 2.5. Timeouts

Packet loss was handled by detecting timeouts and re-sending the lost packet. The approach that we took to detect the timeouts was to use a single timer that essentially monitored the packet with the lowest unacknowledged sequence number. Other approaches that we considered were to attach a timer to each packet, or to maintain a global timer and a dictionary of time offsets for each packet, but these would probably introduce too much of a performance overhead.

The single timer that we used was then reset whenever an ACK confirmed any previously

unacknowledged data before the expiration of the timer. However, if the timer expired before a cumulative ACK that acknowledged the reception of the oldest outgoing packet, the code would then retransmit that packet.

### 2.6. Client Window

The client which is sending the file is in charge of managing the congestion window `cwnd` and the Slow Start threshold `ssthresh`. The client can then ensure to never have more outgoing packets than the amount allowed by the congestion window. The algorithm for managing the window state is described in sections 2.1 and 2.2.

The client also keeps track of a variable called `send_base`, which keeps track of the lowest unacknowledged byte number. This base then slides forward upon reception of a new ACK.

## 3. Challenges

From a software engineering point of view, the most difficult part of the project was having a good design for the code. Such a design would aid in scaling the program and also in future maintenance of the source code. C++ allows object-oriented design, which we did not take advantage of in our source code. As a result, we encountered many instances of repeated code.

Another difficulty was that the protocol implementation must maintain several states, so it was very frequent to encounter errors where we failed to update or incorrectly updated one of these state variables. To give one specific example, the program initially hung whenever the sequence number wrapped around once it hit its maximum value, so the code had to account for this behavior.

Memory management was also a source of difficulty. When testing on a Lubuntu machine, we encountered an error that was due to memory leaks. On our Mac OS X trials, this issue did not appear so we were not aware of the problem until later. Fortunately the solution was as simple as adding a destructor to free any memory that was allocated via the `new` keyword.

Testing the program was another challenge. Since our code was not very modular, it was difficult to test individual components of the program. As a result, our debugging consisted mainly of printing out the values of variables to the standard error file, and conducting many trials with different file sizes and packet loss rates.

In addition to retransmission timeouts, our code also detected socket timeouts, when either the client or the server received no incoming packets from the other for more than 10 seconds (by checking the difference between two `clock_t` variables). The client handles this by running a timer and checking the elapsed time in every iteration of the event loop. The server handles this by setting a socket option to timeout after 10 seconds, which will cause the `recvfrom()` call to set the error number to `EAGAIN`.

## 4. Conclusion

Working on this project allowed us to get an insight into reliable data transfer protocols like TCP. Even on an unreliable link, the client and server can coordinate in such a way that ensures that data is sent and received in its entirety.

## 5. References

[1] Kurose, James; Ross, Keith; "*Computer Networking: A Top Down Approach*"