# CS 5220: Project Technical Report

Simeon Wuthier

*Department of Computer Science*
*University of Colorado, Colorado Springs*
swuthier@uccs.edu

## 1 Compilation / Execute Guide

I selected C++ as my programming language, all algorithms required for the project are fully implemented and functional. The adjustable packet drop rate percentage has been implemented in both UDP protocols on the server. After deleting the PDF on the client then re-requesting it, all three protocols return a successful transmission of the PDF file. Additional test PDFs have been included in the project files. Below are the instructions on how to execute the code files.

### 1.1 Server-side

To compile and execute part 1 (TCP), use the following code.

```
g++ -std=c++11 TCP_Server.cpp -o TCP_Server
./TCP_Server
```

To compile and execute part 2 (UDP) with a frame loss rate of 30% and using protocol 2 (selective repeat), use the following code. The frame loss rate can be an integer between $[0, 100]$ and the protocol must be in $[1, 2]$.

```
g++ -std=c++11 UDP_Server.cpp -o UDP_Server -lpthread
./UDP_Server 30 2
```

**Alternatively, run *./run_part1.sh* to execute part 1 (TCP), and *./run_part2.sh* to execute part 2 ($UDP_1$ and $UDP_2$).**

The included server-side files are as follows.

- **run_part1.sh** — Compiles and executes *TCP_Server.cpp*.
- **TCP_Server.cpp** — TCP basic file transfer protocol.
- **run_part2.sh** — Compiles and executes *UDP_Server.cpp*.
- **UDP_Server.cpp** — UDP file transfer in *stop-and-wait* and *selective repeat*.

### 1.2 Client-side

To compile and execute part 1 (TCP) to download *SIGCOMM10-DataCenterTCP-2.pdf*, use the following code.

```
g++ -std=c++11 TCP_Client.cpp -o TCP_Client
rm -rf SIGCOMM10-DataCenterTCP-2.pdf
./TCP_Client windom.uccs.edu SIGCOMM10-DataCenterTCP-2.pdf
```

To compile and execute part 2 (UDP) to download *SIGCOMM10-DataCenterTCP-2.pdf* using protocol 2 (selective repeat), use the following code.

```
g++ -std=c++11 UDP_Client.cpp -o UDP_Client -lpthread
rm -rf SIGCOMM10-DataCenterTCP-2.pdf
./UDP_Client windom.uccs.edu SIGCOMM10-DataCenterTCP-2.pdf 2
```

**Alternatively, run *./run_part1.sh* to execute part 1, and *./run_part2.sh* to execute part 2.**

The included client-side files are as follows.

- **run_part1.sh** — Compiles and executes *TCP_Client.cpp*.
- **TCP_Client.cpp** — TCP basic file transfer protocol.
- **run_part2.sh** — Compiles and executes *UDP_Client.cpp*.
- **UDP_Client.cpp** — UDP file transfer in *stop-and-wait* and *selective repeat*.

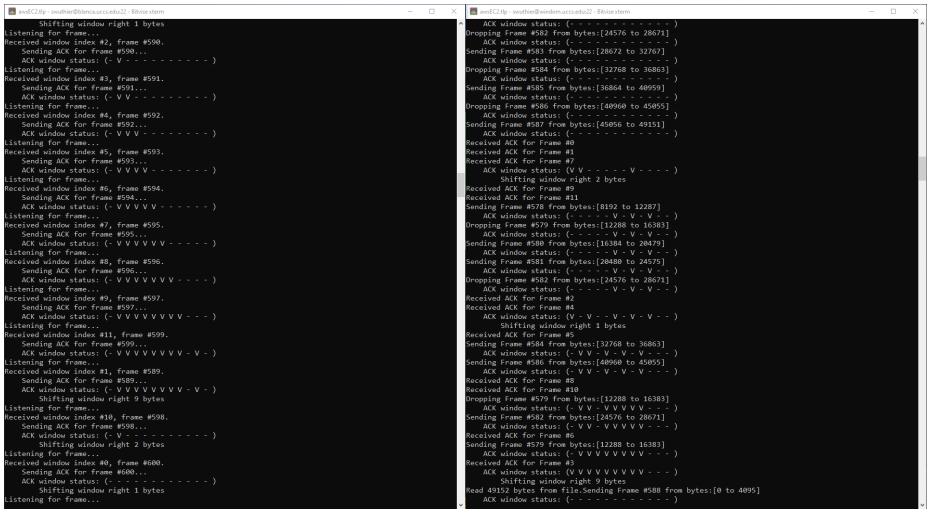
## 2 Self-testing Result

This section discusses the verification steps for stop-and-wait and selective repeat. In both situations, the file transmits and the PDF can be opened, but the protocol-level verification requires additional steps to ensure the sliding window is properly implemented in selective repeat.

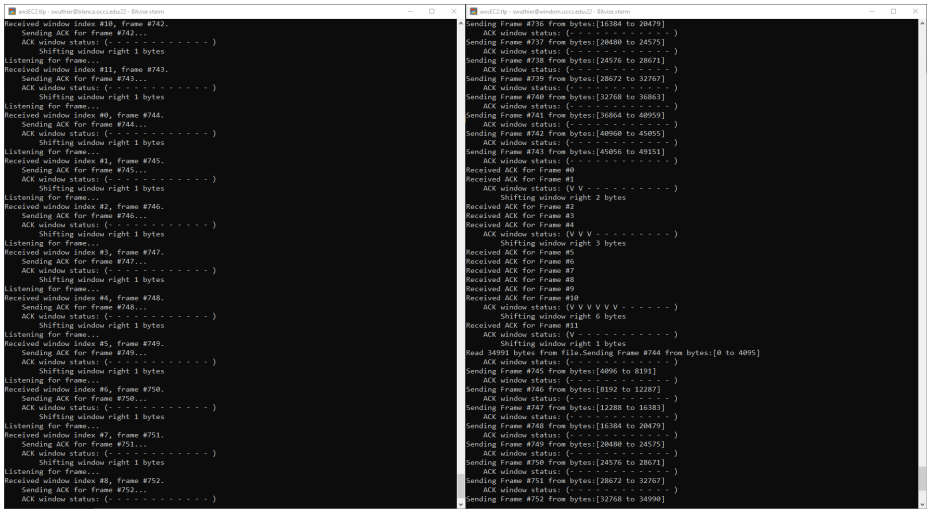
### 2.1 Features / Design Comments

The code is well-documented through comments. I implement both parts by building the framework as the baseline for the protocol. The framework includes the following functions.

- IsFrameDropped — Takes a loss probability, and returns *true* or *false* depending on if the random number is less than the loss probability.
- ComputeCrcChecksum — Given a char* array and length, add all the bytes together, then take the last byte from the sum, this is the non-cryptographic hash that is used for packet payloads.
- SerializeAck — Creates a two-byte ack by packing the negation flag (ACK/NACK), then the sequence number.
- DeserializeAck — Given a char* array, extract the negation flag and sequence number.
- SerializeFrame — A frame consists of an end of transmission (EOT) flag, sequence number, payload size, payload, and checksum. This function packs all the data into a single char* array, ready for transmission.




(a) Using a drop rate of 30% the transmissions are more non-deterministic. The left screen is the client on *blanca.uccs.edu*, while the right screen is the server on *windom.uccs.edu*.




(b) Using a drop rate of 0% the transmissions are more uniform. The left screen is the client on *blanca.uccs.edu*, while the right screen is the server on *windom.uccs.edu*.

Fig. 1: The console screen for selective repeat in UDP. For every ACK transmission the program prints the *windowReceivedLog* Boolean array, where "−" means the message is pending, and "*V*" means the message has been received. When the window's leftmost $N$ packets are received, the window automatically shifts until the rightmost packet returns to "−". Window shifts are displayed on both the client and the server. Using *blanca.uccs.edu* for the client and *windom.uccs.edu* for the server, the transmissions function at maximum speed, and the PDF is readable for every protocol.

- DeserializeFrame — Given a serialized frame, extract all data values. Returns *true* if the checksum matches the cyclic redundancy check (CRC) of the frame, otherwise, it returns *false* signalizing a negated ACK needs to be sent.
- AsyncListenForAcksThenRespond — runs on a separate thread, and is responsible for receiving ACKs and frames, and flipping the corresponding window upon successful data transmission. Before writing to the window (*WindowReceivedLog*), the program must set a mutual exclusion (mutex) lock, then unlock the data after setting the data. This is to ensure that both threads are able to maintain a single data structure at once.
- Fatal — Gracefully terminate the program if the socket is unable to get started, or the client is unable to connect. Once the server has started, it cannot be stopped by the client.

The server runs an infinite loop to continue accepting connections in the future, while the client terminates as soon as it receives the full file contents (denoted by the EOT flag within the frame). The flow of the protocol is split into two phases.

- Phase 1: File name transmission. The client sends the file name to the server every 100ms. Upon receiving the file name on the server, the server will send an ACK if the file exists, and a NACK if the file does not exist. If the client receives a NACK then it will gracefully terminate with an "*ERROR: 404, file not found*" error. If an ACK is received by the client, then it can proceed to the next step.
- Phase 2: File transmission. If the protocol is 1, then stop-and-wait will be used, otherwise, selective repeat will be used.

Selective repeat is the only protocol that utilizes AsyncListenForAcksThenRespond's multi-threaded behavior. Selective repeat was programmed to be parallel because the receiver function can work alongside the more complex protocol-level code, and just update as soon as AsyncListenForAcksThenRespond updates *WindowReceivedLog*. For the phase 1, and phase 2: stop-and-wait protocol, I use the following code to force a *recvfrom* timeout, so that I can assume that a packet has been dropped, and re-send it.

```
// Timeout after FRAMETIMEOUT milliseconds
struct timeval readTimeout;
readTimeout.tv_sec = 0;
// Convert milliseconds to microseconds
readTimeout.tv_usec = FRAMETIMEOUT * 1000;
setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&readTimeout,
    sizeof readTimeout);
```

By default, the frame timeout is 100ms. For frame timeouts of 0ms, the *setsockopt* returns to the default of no timeout, where *recvfrom* waits indefinitely. The port used is 2060. For stop-and-wait, the verification is straightforward since the networking is linear, where a frame and acknowledgment alternate on both the

client and server. For dropped packets, the frame is repeated when the timeout of 100ms is reached. For selective repeat, however, the logging is more verbose since the window is displayed while ACKs are received. Due to multi-threading, there is a race condition between printing the window and updating the window. On the client, there is a notification that frames are received, and an ACK/NACK is sent depending on the frame's success. On the server, there are additional messages for the window size and what byte numbers are being sent with regards to the frame number. ACKs and NACKs are displayed as well.

**Packet Dropping** The following code snippet is used to determine if a packet should be dropped or not.

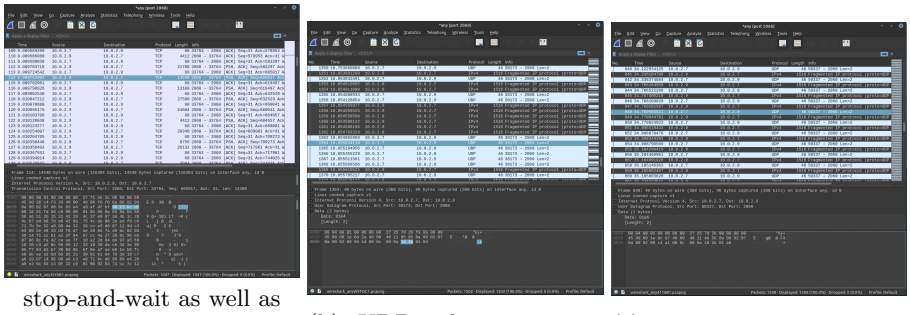
```
#include <cstdlib>
// lossProbability = 0% then no frames are dropped
// lossProbability = 100% then all frames are dropped
bool isFrameDropped(int lossProbability) {
    int r = 1 + rand() % 100; // 1 to 100
    return r <= lossProbability;
}
```

If IsFrameDropped(X) returns *true*, then the server will withhold the *sendto* function from being called. This is included in selective repeat *and* stop-and-wait.

### 2.2 Trace of Flow Control for Lost Frames

Fig. 1a shows the selective repeat protocol in action when the drop rate is set to a 30% lossy network channel. The window visualization (i.e. what frames are pending transmission and what frames have been received by the buffer) are printed to the console. As displayed, there are four types of messages that are displayed on the client which are responsible for representing the state of the node. Upon receiving a window, the node will print the window index (i.e. index within the scope of the window where 0 is the start of the window), and the frame number (i.e. which frame has been received with respect to the entire transmission, where 0 is the initial 4096 bytes of the file). The client has a similar behavior but instead of sending a frame, it will also print "Dropping Frame X from bytes [Y to Z]", which helps to better understand which packets are being dropped so that the user can expect a NACK. In the real world this is unrealistic since there would be no way to determine when a frame is dropped without a NACK, however, for the sake of experimentation this helps to debug and provide more verbose output. Fig. 1b shows the same behavior but with a drop rate of 0%. In this situation, the PDF file is sent showed a 93.32% decrease in transmission time.

Using Wireshark on the client with the filter "*port 2060*", I retrieve three packet capture (PCAP) files, one for TCP, one for UDP stop-and-wait, and one



stop-and-wait as well as

(a) TCP file transfer using the classical protocol.

(b) UDP selective repeat file transfer using a window size of 10 and a packet drop rate of 30%. There is a greater variation because dropped traffic causes NACK transmissions.

(c) UDP stop-and-wait file transfer using a packet drop rate of 30%. From this, it is clear that the traffic is uniform in that the ACK and data messages alternate.

Fig. 2: Transferring the provided "*SIGCOMM10-DataCenterTCP-2.pdf*" over varying protocols. The frame size is 4096 bytes with a 10-byte header, consisting of an EOT flag, the frame size in bytes, the payload, and the four-byte CRC hashing algorithm. ACK messages are 2 bytes consisting of the Boolean negation flag and the sequence number (referring to the frame index).

for UDP selective repeat[1]. Fig. 2 shows each PCAP file where it is easier to differentiate between frames and ACK/NACK transmissions. For ACK/NACK, the byte size is 2 (where the first byte is the negation flag and the second byte is the sequence number). Selective repeat shows greater variation in the packet/frame transmissions due to the window size of 10 allowing different frames to be transmitted simultaneously.

[1] For ACK transmissions, Wireshark detects this as "IPv4". This is due to the fact that frames have no headers and are simply two bytes, [*negate*, *seqnum*].