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, and fully implement all algorithms. I also implement the adjustable packet drop rate percentage 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. 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.

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

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

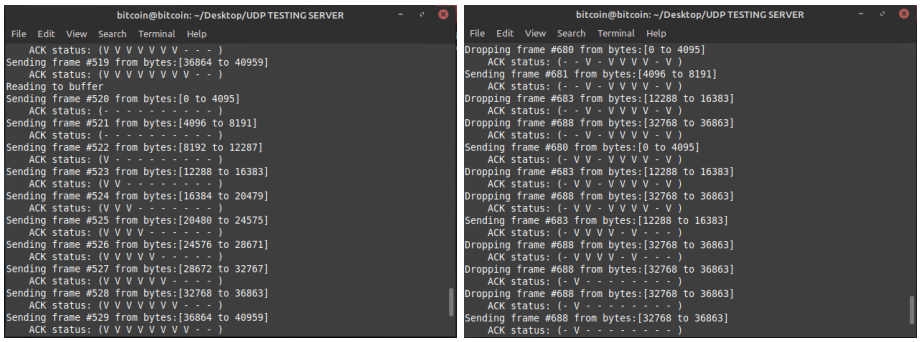
To compile and execute part 2 (UDP) to download *SIGCOMM10-DataCenterTCP-2.pdf* using protocol 2 (selective repeat), use the following code.

```
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*.



(a) Using a drop rate of 0% the transmissions are more uniform.

(b) Using a drop rate of 30% the transmissions are more non-deterministic.

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 rightmost packet is received, the window automatically shifts until the rightmost packet is "–". Note that experiments were executed on two virtual machines with a shared NAT Network, therefore delays are negligible. However, using *blanca.uccs.edu* for the client and *windom.uccs.edu* for the server, the transmissions still work at maximum speed, and the PDF is readable for every protocol.

2 Self-testing Result

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. This 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.
- FastHash — 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 EOT flag, sequence number, payload size, payload, and checksum. This function packs all the data into a single char* array, ready for transmission.
- DeserializeFrame — Given a serialized frame, extract all data values. Returns *true* if the checksum matches the hash of the frame, otherwise, it returns *false* signalizing a negated ACK needs to be sent.
- HandleAckMessages — 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.

Selective repeat is the only protocol that utilizes HandleAckMessages'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 HandleAckMessages updates *WindowReceivedLog*. For the 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.

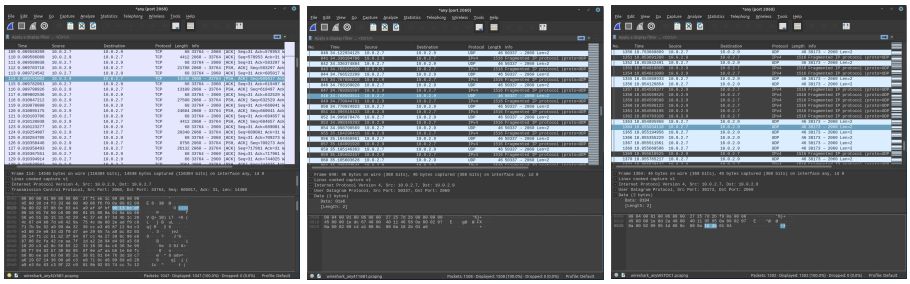
```
// Timeout after FRAMETIMEOUT milliseconds
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = FRAMETIMEOUT * 1000; // Convert milliseconds to microseconds
setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof tv);
```

By default, the frame timeout is 100ms.

2.2 Trace of Flow Control for Lost Frames

The port used is 2060. 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 for UDP selective repeat. Fig. 2 shows each PCAP.

For selective repeat, 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 shown in Fig. 1.



(a) TCP file transfer using the classical protocol.

(b) 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.

(c) 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.

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 end of transmission (EOT) flag, the frame size in bytes, the payload, and a byte hashed checksum (where the hash algorithm is $([\sum_i \lambda_i] \mod 2^{16})$ for each frame, λ_i). Ack messages are 2 bytes consisting of the Boolean negation flag and the sequence number (referring to the frame index).