SFU CMPT 371 Mini Project 2 Write up

Reid Lockhart
Andy Wang

Part 1. Protocol Specifications

Our connection-oriented protocol is built on a UDP socket. Each packet has a header and a payload body. The header fields are detailed below

Sequence number (4 bytes)	ACK (4 bytes)	Flags (1 byte)	Receiver window (rwnd) (2 bytes)
Payload length (2 bytes)	Checksum (2 bytes)	Data	

The first phase of a connection is a 3 way handshake initiated by the sender. First, the sender sends a packet with the SYN flag set. The receiver sees this and responds with a package having both the SYN and ACK flags set. Finally, the sender sends another packet with the ACK flag set to notify that it has acknowledged the server's response. The connection is now open. To close the connection, a similar protocol is followed. When the sender is finished transmitting its data, it sends a packet with the FIN flag set. The server receives this and sends two packets, first is an ACK so the receiver knows it got the FIN message. The second is its own package with the FIN flag set. The receiver sees this and sends back a final ACK. Now, the connection is closed. During this process, the sockets have a timeout and they will retransmit the packet if the timeout occurs. For example, the client is finished sending its data but its FIN message gets lost or discarded. It doesn't hear anything back, the timeout occurs, and so the FIN message is sent again.

Our protocol uses Go-Back-N to ensure reliability. The sender keeps a set window size and a timeout. If the entire window hasn't been ACKed when the timeout runs, it re-sends the

current window. The receiver's job is to send an ACK for each packet it receives so the sender can adjust the window and keep track of whether it needs to resend the window. For congestion control, our protocol uses the AIMD algorithm to boost performance while minimizing losses. The sender window is set to the minimum of cwnd and client_rwnd. We cannot go over client_rwnd or this would risk overflowing the receiver's buffer. Cwnd starts at 1, and increases by one every successful ACK. Once a loss is detected, it is halved and then the increase process begins again. Once Cwnd reaches the point that a packet was lost last time, it increases slowly, only by ¼ per ACK. This is so that we can hopefully keep the size large enough to enable a lot of packages to be in-flight at once, without the number getting too large, which would result in losses which can be expensive for go-back-n since the entire unACKed window needs to be retransmitted.

Each packet features a simple checksum, which is the one's complement of every two byte increment of the packet using 1's complement. This provides a quick but somewhat unreliable way of seeing if any bit flips or corruption occurred during transport.

To ensure the sender isn't sending too many packets, the connection ACK from the receiver includes rwnd, which is the buffer space it has to store packets. The sender will set its window size to the minimum of rwnd and cwnd so that the buffer won't overflow.

As part of the packet header, the receiver sends rwnd, which is the buffer space it has to receive into. This is taken into account by the sender who adjusts the maximum window size so that

Part 2. Analyze Traffic

To analyze the performance of our protocol, we started by capturing the packet trace for a transmission using the wireshark Adapter for loopback traffic capture. To prove that our protocol is connection oriented, we can view the three-way handshake taking place in the wireshark trace.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	UDP	47	60406 → 8080 Len=15
2	0.014457	127.0.0.1	127.0.0.1	UDP	47	8080 → 60406 Len=15
3	0.014658	127.0.0.1	127.0.0.1	UDP	47	60406 → 8080 Len=15
4	0.015448	127.0.0.1	127.0.0.1	UDP	111	60406 → 8080 Len=79

At the very beginning, starting at Time 0.000000, port 60406 sends a SYN packet to port 8080 (the server). The server then sends a SYN-ACK to acknowledge the SYN, with an ACK number equal to the received SYN + 1. After the client receives the SYN-ACK, it sends a final ACK to complete the connection establishment before transmitting data. Once the server receives this final ACK, it confirms that the connection is fully established. This proves that our protocol is connection-oriented because it establishes a logical connection using a three-way handshake (SYN → SYN-ACK → ACK) before any data is exchanged.

4	0.015448	127.0.0.1	127.0.0.1	UDP	111	60406 → 8080 Len=79
5	0.015660	127.0.0.1	127.0.0.1	UDP	47	8080 → 60406 Len=15
6	0.016005	127.0.0.1	127.0.0.1	UDP	111	60406 → 8080 Len=79
7	0.016206	127.0.0.1	127.0.0.1	UDP	47	8080 → 60406 Len=15

After the three-way handshake, the client begins transmitting data packet by packet, and the server returns an ACK for each packet it receives.

Flow Control:

To ensure flow control, the server advertises its current receive window to the client, and the client adjusts the amount of data it sends based on this value. The server processes incoming packets only if the current buffer size is below the pre-set limit and returns the remaining buffer space as rwnd; otherwise, the packet is discarded.

In the Wireshark capture, we can observe that the server's ACK packets include an advertised window (rwnd) value that decreases as the server's buffer fills and increases when the processing thread drains the buffer. This confirms the receiver-driven flow control mechanism. The client never sends more than min(cwnd, rwnd) outstanding packets, which prevents buffer overflow and aligns with the behavior defined in the reliable data transfer (rdt) design model.

571	12.347892	127.0.0.1	127.0.0.1	UDP	111	57968 → 8080 Len=79
572	13.351681	127.0.0.1	127.0.0.1	UDP	111	57968 → 8080 Len=79
573	13.367448	127.0.0.1	127.0.0.1	UDP	47	8080 → 57968 Len=15

In the screenshot above, packets No. 571 and 572 show the client sending the same packet (seq = 250) twice. This occurs because earlier, when processing the packet with seq = 225, the server's buffer was full and the packet was discarded, prompting the client to retransmit.

Loss/error Simulation

We simulate errors by occasionally dropping packets using a random function, which allows us to test the retransmission logic. When packets are lost, the server continues to return ACKs for the last correctly received sequence number, effectively re-requesting the missing packets. Out-of-order packets do not advance expected_seq, and the server sends the last valid ACK instead. This enables us to test retransmissions, duplicate ACK behavior, out-of-order handling, and timeout-based recovery in a controlled environment, as required when implementing rdt protocols on top of UDP.

```
Client.py:
if random.randint(1,100) > 2:
      sock.sendto(pkt, SERVER_ADDR)
      sent = messages[next_seq].decode('utf-8')

next_seq += 1
```

```
Server.py:
if seq == expected_seq:
       expected_seq += 1
       CURRENT_BUFFER_SIZE += 1
       # process packets
       # …..
else:
     print(f"WARNING: Out of order. Expected: {expected_seq}, Received: {seq})")

# return previous expected sequence number
ack_pkt = create_packet(0, expected_seq - 1, ACK_FLAG, (BUFFER_SIZE -
     CURRENT_BUFFER_SIZE), b"")
```

In the screenshot below, the client sends a large number of packets in rapid succession. This occurs because, at packet No. 330, the server was expecting seq = 146 but instead received seq = 147, 148, …, 152. As a result, starting at No. 340, logged by wireshark, the client retransmits all packets that the server had previously dropped, starting with seq = 146.

340	7.191444	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
341	7.191504	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
342	7.191532	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
343	7.191558	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
344	7.191583	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
345	7.191609	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
346	7.191635	127.0.0.1	127.0.0.1	UDP	111 57968 → 8080 Len=79
347	7.206708	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
348	7.206811	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
349	7.206859	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
350	7.206904	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
351	7.206949	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
352	7.206993	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15
353	7.207037	127.0.0.1	127.0.0.1	UDP	47 8080 → 57968 Len=15

Congestion Control:

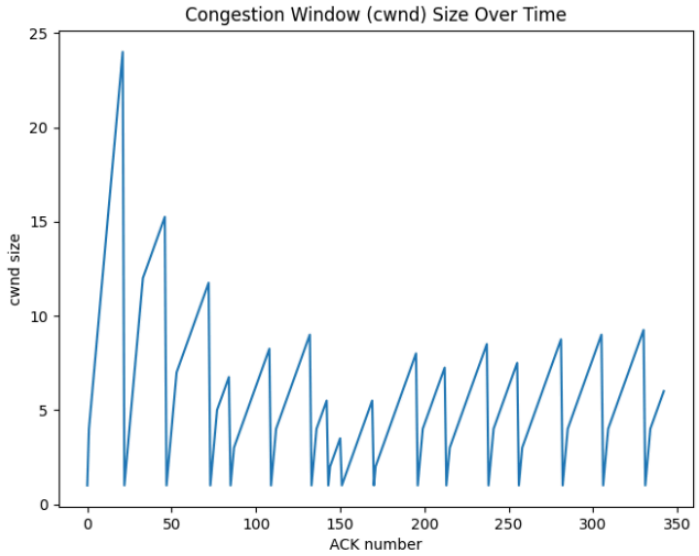
Our protocol also integrates congestion control on the client side. Initially, we set cwnd = 4 and threshold = 32. If cwnd < threshold, we increase cwnd by 1 (slow start); otherwise, we increase it by 0.25 (congestion avoidance). When a timeout occurs indicating congestion or loss we reduce the threshold to half of the current cwnd and reset cwnd to 1. This resets the window to a safe value and restarts slow start. In the cwnd and rwnd graphs, we can observe the expected behavior: exponential growth during slow start, linear growth during congestion avoidance, a sharp drop to 1 after a timeout, and then a restart of slow start.

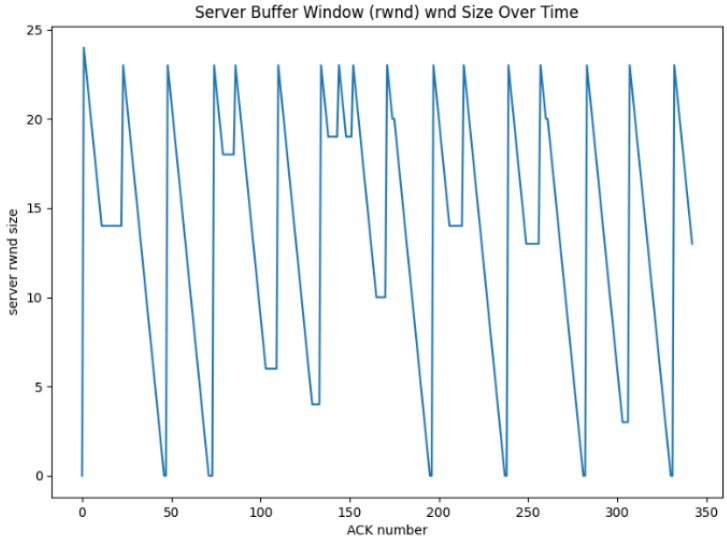
```
sock.settimeout(TIMEOUT)
 try:
    data, addr = sock.recvfrom(1024)
 except socket.timeout:
    print("-----\nTimeout waiting for ACK; resending window\n-----")
    # reset cwnd and set threshold to what it was before failure

    threshold = max(int(cwnd / 2), 1)
    cwnd = 1

    for r in range(send_base, next_seq):
       packet = create_packet(r, 0, DATA_FLAG, CLIENT_RWND, messages[r])
       sock.sendto(packet, SERVER_ADDR)
    continue
```

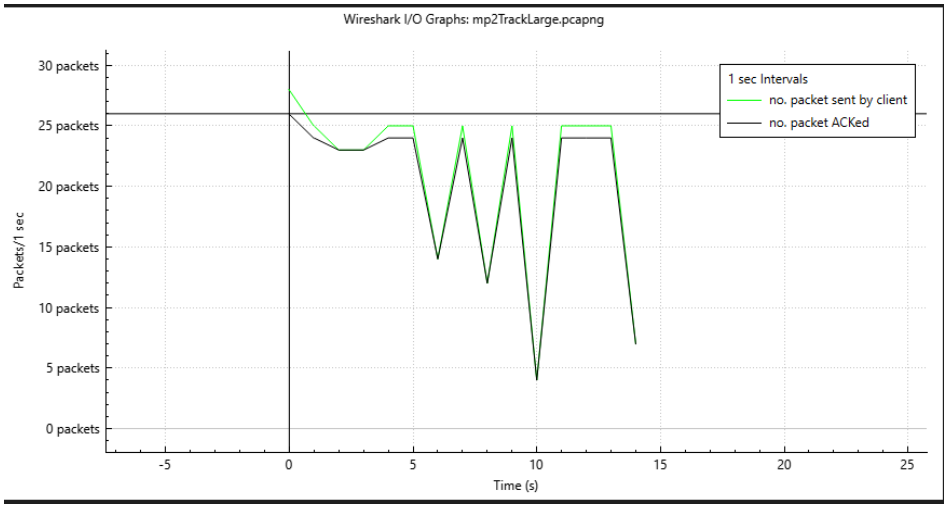
Acceptable Performance

We evaluated acceptable performance by observing the rate at which data packets were delivered over time. The graphs show a stable throughput pattern, with rapid increases during slow start, a more gradual slope during congestion avoidance, and brief dips during retransmission phases. The transfer completes successfully with minimal stalls, demonstrating

that the protocol adapts well to simulated loss, avoids buffer overflows through flow control, and maintains steady progress, thereby achieving acceptable performance.







After all the data is transmitted, and receives ACK for every packet, the client sends a FIN packet to the server to indicate that no more data will be transmitted. Upon receiving this FIN, the server responds with an ACK acknowledging the client's FIN, and then sends its own FIN to begin closing the connection from the server side.The client waits for this server-side FIN, and once it arrives, the client sends a final ACK acknowledging it.

The server then resets its internal state, including expected_seq = 0, connected = False, client_addr = None….

```
631 14.398311    127.0.0.1    127.0.0.1    UDP    47 57968 → 8080 Len=15
632 14.398866    127.0.0.1    127.0.0.1    UDP    47 8080 → 57968 Len=15
633 14.398894    127.0.0.1    127.0.0.1    UDP    47 8080 → 57968 Len=15
634 14.399150    127.0.0.1    127.0.0.1    UDP    47 57968 → 8080 Len=15
```

In wireshark, it also shows the 4 packets sent between client and server to end the connection .This orderly FIN → ACK → FIN → ACK exchange mirrors TCP's four-way termination and ensures a clean and reliable shutdown of the session.

Overall, our protocol successfully establishes a connection, reliably transfers data, and gracefully closes the session using a TCP-style 3-way handshake and 4-way termination. Through flow control, congestion control, packet loss simulation, and retransmission handling, it ensures correctness and stability over an unreliable UDP channel. The captured Wireshark traces and performance graphs confirm that the protocol operates reliably and efficiently.