CMSC481 Project 1: Reliable Transport Protocol

Fall 2025

Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Due: November 20, 2025, at 11:59pm (UMBC time)

1 A Word of Warning

Please make sure to carefully read and follow all submission instructions (Section 8) before submitting your project. Failure to do so may result in a partial or zero score. This project is due by **November 20, 2025, at 11:59pm (UMBC time)**.

2 Project Overview

In this project you will be building a reliable transport protocol, Reliable UDP, or rUDP, on top of standard UDP. Your rUDP protocol must provide reliable delivery of UDP datagrams in the face of packet loss, reordering, and corruption. As we discussed in lecture previously, UDP does not normally offer reliable transport, and here we're going to fix that.

You are going to implement a sender (`rSender.py`) that follows the rUDP protocol described below. Your code is written in Python 3 and will be tested using the provided autograder. The receiver implementation (`rReceiver.py`) is already complete and provided to you.

We made three videos to help you learn about this project and get started. You can watch them here: https://www.youtube.com/@E-WiNs-UMBC/playlists. You're welcome to ask questions or leave comments there. Please **stay polite and don't share answers directly.**

3 Understanding the rUDP Protocol

3.1 Why We Need rUDP

UDP (User Datagram Protocol) is a simple transport protocol that provides no guarantees about packet delivery. Packets can be lost, duplicated, reordered, or corrupted. rUDP adds a reliability layer on top of UDP, similar to what TCP provides, but with a simpler implementation suitable for learning. The key reliability mechanisms in rUDP are:

- **Sequence numbers**: Each packet is numbered to detect loss and reordering
- **Acknowledgments (ACKs)**: Receiver confirms which packets have arrived
- **Checksums**: Detect corrupted packets using CRC32

- **Retransmission**: Resend packets that haven't been acknowledged
- **Sliding window**: Control how many packets can be "in flight" at once

3.2 Packet Structure and Size Calculations

Every rUDP packet consists of a 16-byte header followed by optional data:

```
struct Packet {
    uint32_t type;      // 4 bytes: Packet type (0-3)
    uint32_t seq_num;   // 4 bytes: Sequence number
    uint32_t length;    // 4 bytes: Length of data payload
    uint32_t checksum;  // 4 bytes: CRC32 checksum of data
    byte data[];        // Variable: Data payload
}
```

The packet size limits come from network constraints:

- Ethernet Maximum Transmission Unit (MTU): 1500 bytes
- Minus IP header: 1500 - 20 = 1480 bytes
- Minus UDP header: 1480 - 8 = 1472 bytes (max rUDP packet size)
- Minus rUDP header: 1472 - 16 = 1456 bytes (max data payload)

This is why you'll see 1472 as the maximum packet size and 1456 as the maximum data chunk size throughout the code.

3.3 Protocol Flow Explained

The protocol follows the sequence shown in Figure 1:

1. **Connection Establishment:** The sender initiates a connection by sending a START packet with a random sequence number (e.g., 3928107). This random number helps distinguish different connections. The receiver responds with an ACK containing the same sequence number, confirming the connection is established.
2. **Data Transfer:** The sender breaks the file into chunks of up to 1456 bytes each. Each chunk is sent as a DATA packet with sequence numbers starting from 0 and incrementing by 1. The sender can have multiple packets "in flight" up to the window size.
3. **Acknowledgment:** The receiver uses **cumulative ACKs**. An ACK with seq_num=5 means "I have received all packets 0-4 and am expecting packet 5 next." This is different from individual ACKs where each packet is acknowledged separately.
4. **Connection Termination:** After all data is sent, the sender sends an END packet with the same random sequence number used in START. The receiver acknowledges with an ACK, and the connection closes cleanly.

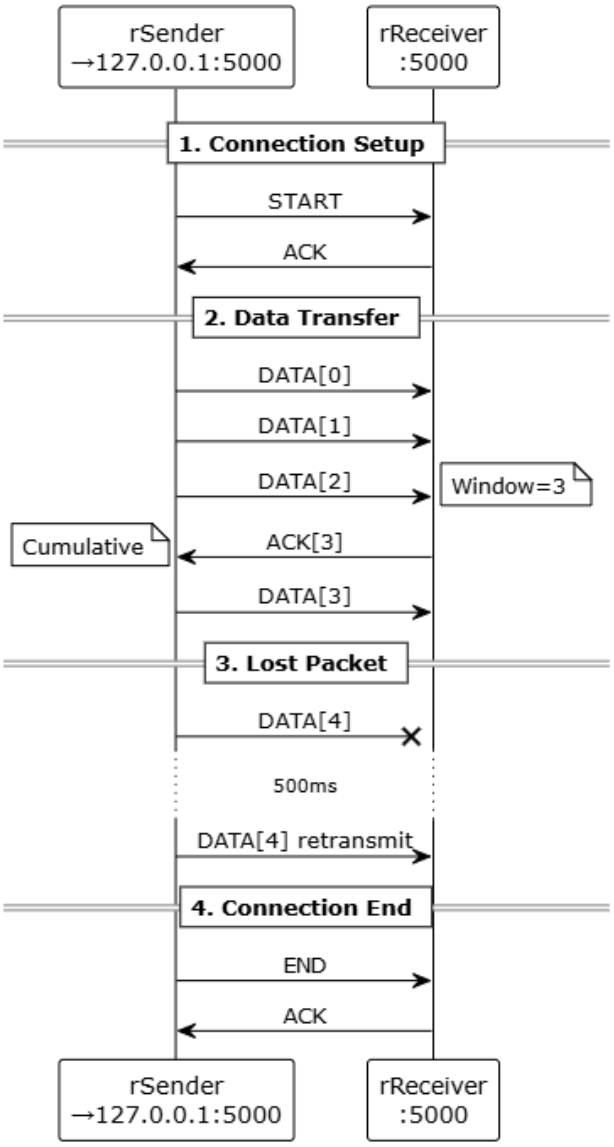

Figure 1: rUDP Protocol Flow

3.4 Understanding Cumulative ACKs

Cumulative acknowledgments are a key concept in rUDP. Instead of acknowledging each packet individually, the receiver acknowledges the highest in-order packet received. For example:

- Sender sends packets 0, 1, 2, 3, 4
- If receiver gets packets 0, 1, 2, 4 (packet 3 is lost)
- Receiver sends ACK=3 (meaning "I have 0,1,2 and need 3 next")
- Even though packet 4 arrived, it cannot be acknowledged until 3 arrives
- When packet 3 finally arrives, receiver sends ACK=5 (acknowledging both 3 and 4)

4 rSender Implementation

rSender will read an input file and send it to the rReceiver using UDP sockets. rSender will chunk the input file into the proper size and append a CRC32 checksum. The seqNum will increment by one for each packet that is sent.

You will use a sliding window system to help achieve reliable data transfer. The size of the window will be specified when rSender is invoked. rSender must accept cumulative ACK packets from rReceiver.

After transferring the entire file, you should send an END message to mark the end of the connection. rSender must ensure reliable data transfer in the face of lost packets, corrupted packets, reordered ACKs, duplication of packets, and delay in ACK arrivals.

rSender will have a 500 ms retransmission timer for the oldest unACKed packet in its window. If the timer expires then rSender will retransmit all packets in the current window. Whenever the sender window progresses you will reset the timer for the new first unACKed packet in the window.

5 Implementation Checkpoints

Your implementation is divided into 6 checkpoints (0-5), each building upon the previous ones. **You will complete specific sections in the `rSender.py` template file.**

5.1 Checkpoint 0: Basic Setup and Project Report (20 points)

Verifies that your project directory is correctly set up with all required files. The autograder checks that `rSender.py`, `packet.py`, and other necessary files exist and can be imported. For all checkpoints, please use your results to form a project report (more details in Section 8).

Common Mistake: Running the autograder from the wrong directory. You must run the autograder from within your project directory. Your working directory should look like this:

```
./
    autograder.py
    packet.py
    rReceiver.py
    rSender.py      # Your implementation
    input/          # Test files directory
```

5.2 Checkpoint 1: Connection Establishment (20 points)

Implement the `perform_handshake()` method to establish and terminate connections. Send the handshake packet (START or END) and wait for an ACK with matching sequence number. Retry up to 10 times if no ACK is received, using socket timeout to detect lost packets. Return True if successful, False after 10 failed attempts.

5.3 Checkpoint 2 & 3: Sliding Window (40 points)

Implement the sliding window protocol with cumulative ACKs in the marked section of `transfer_file()`. When receiving an ACK, check if it advances the window (ack.seq_num > left). If yes, slide the window forward by updating left, right, and window variables. Send any newly exposed packets when the window slides, and detect when all packets have been acknowledged.

5.4 Checkpoint 4: Packet Loss Recovery (20 points)

Add timeout and retransmission logic to handle packet loss. When the 500ms timeout expires, retransmit all packets in the current window and reset the timeout timer. Continue waiting for ACKs after retransmission.

5.5 Checkpoint 5: RTT Estimation (20 points, Extra Credit)

Implement RTT measurement and estimation for demonstrating convergence. Calculate sample RTT when receiving ACK for the first packet in window. Update estimated RTT using exponential weighted moving average with $\alpha = 0.125$:

$$\text{EstimatedRTT} = (1 - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$$

Log RTT measurements to show how your estimation converges over time. Your implementation should demonstrate that the estimated RTT stabilizes as more samples are collected.

6 Testing Your Implementation

Use the provided autograder to test your implementation:

```
# Test all checkpoints (0 through 5)
python3 autograder.py

# Test only checkpoint 3
python3 autograder.py 3

# Test checkpoint 0 (basic setup)
python3 autograder.py 0
```

The autograder will automatically run your sender against the provided receiver with various network conditions to verify correct behavior. **Your grade will be based on the autograder results and the TA's manual evaluation of your submitted code and project report.**

Note for autograder: MacOS may have security mechanisms that can cause errors when creating files. If Testcase 1 fails but all other testcases pass, it will not affect your points for that checkpoint. On Windows or Linux, the testcases should run correctly.

7 Reading and Understanding Logs

The log files are crucial for debugging your implementation. Each line in the log represents one packet sent or received, with four fields:

```
<type> <seq_num> <length> <checksum>
```

7.1 Log Field Meanings

- **type**: Packet type as integer (0=START, 1=END, 2=DATA, 3=ACK)
- **seq_num**: Sequence number of the packet
- **length**: Length of data payload in bytes (0 for control packets)
- **checksum**: CRC32 checksum of the data (0 for non-DATA packets)

7.2 Example Log Analysis

Here's an example sender log with explanations:

```
0 3928107 0 0         # START packet, random seq 3928107
3 3928107 0 0         # Received ACK for START
2 0 1456 2154266692   # Send DATA packet 0, full size (1456 bytes)
2 1 1456 3367462139   # Send DATA packet 1
2 2 1456 1893930610   # Send DATA packet 2
2 3 1456 2667534801   # Send DATA packet 3
2 4 1456 1234567890   # Send DATA packet 4 (window full, size=5)
3 1 0 0               # Received ACK=1 (packet 0 received)
2 5 1456 9876543210   # Window slides, send packet 5
3 2 0 0               # Received ACK=2 (packet 1 received)
2 6 1456 1111111111   # Window slides, send packet 6
3 5 0 0               # Received ACK=5 (packets 2,3,4 received)
2 7 784 2222222222    # Send packet 7 (partial packet, 784 bytes)
2 8 1456 3333333333   # Send packet 8
2 9 1456 4444444444   # Send packet 9
3 10 0 0              # Received ACK=10 (all data received)
1 3928107 0 0         # Send END packet
3 3928107 0 0         # Received ACK for END
```

Key observations from this log:

- The START and END packets use the same random sequence number (3928107)
- DATA packets have incrementing sequence numbers starting from 0
- Most DATA packets are full size (1456 bytes), except packet 7 which is the last chunk
- ACK=5 acknowledges multiple packets at once (cumulative ACK in action)
- The window sliding is visible as new packets are sent after ACKs arrive

7.3 Identifying Common Issues in Logs

Repeated sequence numbers in sender log: Indicates retransmission due to timeout

```
2 3 1456 2667534801  # Original transmission
2 3 1456 2667534801  # Retransmission after timeout
```

Duplicate ACKs in sender log: Receiver is signaling packet loss

```
3 3 0 0  # Receiver needs packet 3
3 3 0 0  # Still needs packet 3 (got packet 4 or 5 out of order)
3 3 0 0  # Still needs packet 3 (got packet 6 out of order)
```

Non-sequential ACKs: Shows successful cumulative acknowledgment

```
3 2 0 0  # Have packets 0,1
3 7 0 0  # Have packets 0-6 (packets 2-6 just arrived)
```

8 Submission Instructions

8.1 Deliverables

You must submit the following two files to Blackboard:

1. **rSender.py** - Your complete implementation
2. **LastName_FirstInitial_CMSC481_Project1.pdf** - Your project report

Do not submit any other files (e.g., `packet.py`, `rReceiver.py`, test files, or log files). Your report must document your implementation process and results:

- **Format:** Times New Roman or Arial, 12 pt, double space, PDF only
- **Length:** Maximum 1 page per checkpoint (6 pages maximum for Checkpoints 0-5)
- **Naming:** `LastName_FirstInitial_CMSC481_Project1.pdf` (e.g., `Smith_J_CMSC481_Project1.pdf`)

Content for each checkpoint:

- How many tests did you pass?
- What worked and what didn't work?
- Challenges encountered and how you addressed them
- Sources referenced (e.g., textbook, online resources, AI tools)
- Screenshot showing the checkpoint passed in the autograder

8.2 Grading

Component	Points
Checkpoint 0: Basic Setup and Project Report	20
Checkpoint 1: Connection Establishment	20
Checkpoint 2: Sliding Window Part 1	20
Checkpoint 3: Sliding Window Part 2	20
Checkpoint 4: Packet Loss Recovery	20
Checkpoint 5: RTT Estimation (Extra Credit)	20
Total	100 (120 with EC)

Late Policy: Late assignments and project will be accepted with the following penalties applied to the earned score:

– Up to 12 hours late: -10%

– 12–24 hours late: -20%

– 24–48 hours late: -40%

– More than 48 hours late: no credit

Academic Integrity: This is an individual project. You may discuss concepts with classmates, but all the code must be your own. Do not share code or look at others' implementations. If you use ChatGPT, similar AI tools, or reference code or solutions found online, you must fully disclose how they were used. For more details, see `https://academicconduct.umbc.edu/`.

9 Running Individual Programs (Reference Only)

While you will primarily use the autograder for testing, the individual programs can be run as follows:

Receiver:

```
python3 rReceiver.py <port> <window-size> <output-file> [--log FILE]
```

Sender:

```
python3 rSender.py <receiver-IP> <receiver-port> <window-size>
                   <input-file> [--log FILE] [--rtt] [--loss-recovery]
```

The receiver can simulate network conditions with `--drop N` (drop every Nth packet) and `--delay MS` (add delay to ACKs).