# CPAN 226 - Lab 2: Reliable Transport over UDP

**Starter Repository:** https://github.com/lozass/CPAN226_Lab_2.git

## 1. Overview

In this lab, you will work with a simple Client-Server file transfer application. The provided code uses standard UDP (User Datagram Protocol). While UDP is fast, it provides **no guarantees** regarding packet order or delivery.

You will run your traffic through a "Network Proxy" (Relay) that simulates a chaotic network. Your goal is to upgrade the Client and Server to handle these network imperfections and ensure the file (old_lady.jpg) arrives perfectly intact.

### The Challenge

The lab is divided into two parts (50% each):

1. **Part 1:** Handle Packet Loss (Reliability).
2. **Part 2:** Handle Out-of-Order Delivery (Sequencing).

## 2. Getting Started

### Step 1: Get the Starter Code

You must clone the instructor's repository to get the baseline code and the test image.

```
git clone https://github.com/lozass/CPAN226_Lab_2.git
cd CPAN226_Lab_2
```

You should see:

- server.py (The Receiver - *You will modify this*)
- client.py (The Sender - *You will modify this*)
- relay.py (The Network Simulator - *Do not modify*)
- old_lady.jpg (The test image)

### Step 2: Initialize Your Submission Repo

You must submit your work via your own GitHub repository.

1. Create a **new public repository** on your GitHub account (e.g., CPAN226_Lab2_YourName).
2. Copy the client.py and server.py files you just downloaded into your new repository folder.
3. **Important:** Add the following header to the top of both client.py and server.py:

   ```
   # This program was modified by [Your Name] / [Your Student ID]
   ```

# 3. Observations: The "Working" vs "Broken" Network

Before writing code, you must observe how UDP behaves in ideal vs. chaotic conditions.

## Test A: Baseline Test (Direct Connection)

First, verify that the starter code works perfectly when there is no network interference.

1. **Terminal 1 (Server):** Start the server on port 12001.
   ```
   python server.py --port 12001 --output received_direct.jpg
   ```

2. **Terminal 2 (Client):** Send the image **directly to the Server** (Port 12001).

   ```
   # Note: We are targeting 12001 (Server), NOT 12000 (Relay)
   python client.py --target_port 12001 --file old_lady.jpg
   ```

3. **Result:** Open received_direct.jpg. It should be a perfect copy of old_lady.jpg. This proves the basic logic works.

   Take a screenshot with the two image files (original and received side-by-side, with visible file names and the local system/clock time and date).

## Test B: The "Broken" Network (Relay Simulation)

Now, we introduce the Relay to simulate network jitter.

1. **Terminal 1 (Server):** Restart the server on port 12001.

   ```
   python server.py --port 12001 --output received_relay.jpg
   ```

2. **Terminal 2 (Relay):** Configure the relay to introduce **20% reordering** but **0% packet loss**.

   ```
   python relay.py --bind_port 12000 --server_port 12001 --loss 0.0 --reorder 0.2
   ```

3. **Terminal 3 (Client):** Send the image to the **Relay** (Port 12000).

```
# Note: We are targeting 12000 (Relay) this time
python client.py --target_port 12000 --file old_lady.jpg
```

4. **Result:** Open received_relay.jpg.
   - **Observation:** You will see a corrupted image (color shifts, grey bars, or pixel scrambling).
   - *Why?* The bytes arrived in the wrong order. The image viewer tried to render the shuffled bytes, destroying the image.
5. **Take a Screenshot:** Capture this corrupted image. **Ensure your system clock (Date/Time) is visible in the screenshot.**

---

# 4. Implementation Guide

Your task is to modify client.py and server.py to fix the issues observed in Test B.

## Part 1: Fixing Packet Loss

*Standard RDT (Reliable Data Transfer) implementation.*

- **Goal:** Ensure the file transfers correctly even if relay.py is run with --loss 0.3.
- **Logic:** Implement **Stop-and-Wait** or **Go-Back-N**.
  1. Client sends packet.
  2. Server sends ACK.
  3. Client waits for ACK. If timeout -> Retransmit.

## Part 2: Fixing Out-of-Order Delivery

*This fixes the corrupted image problem.*

To fix the image, the Server must not write packets to disk immediately as they arrive. Instead, it must **Sequence** them and **Reorder** them.

### Step A: The Header (Client & Server)

You cannot reorder packets if you don't know their original order. You must add a **Sequence Number** to every packet.

- **Current Packet:** [ DATA ]
- **New Packet:** [ Sequence # | DATA ]

**Hint:** Use Python's struct library to pack the integer sequence number before the data.

```
# Client Side Hint
import struct
header = struct.pack('!I', sequence_number) # 4 bytes for Integer
packet = header + file_chunk
```

### Step B: The Reordering Logic (Server)

The Server needs a smarter logic than "receive and write." It needs a **Buffer**.

**Logic / Pseudocode:**

```
expected_seq_num = 0
buffer = {} # Dictionary to store out-of-order packets: { seq_num : data }

while True:
    packet = receive_packet()
    seq_num, data = unpack_header(packet)

    if seq_num == expected_seq_num:
        # 1. This is the packet we needed next! Write it.
        write_to_file(data)
        expected_seq_num += 1

        # 2. VITAL STEP: Check if the *next* packet is already waiting in our buffer
        while expected_seq_num in buffer:
            buffered_data = buffer.pop(expected_seq_num)
            write_to_file(buffered_data)
            expected_seq_num += 1

    elif seq_num > expected_seq_num:
        # Packet arrived too early (out of order). Store it for later.
        buffer[seq_num] = data

    else:
```

```
    # Packet is old (duplicate). Ignore it.
    pass
```

# 5. Submission Requirements

You must submit the link to your GitHub repository containing:

1. **Modified Code:** client.py and server.py
   - **MUST** include the header: # This program was modified by [Name] / [ID]

2. **Lab Report (PDF or Readme):**
   - **Screenshot 1:** The two jpg files (old_lady.jpg and the received version) from the direct client to server transfer (without the relay).
   - **Screenshot 2:** The corrupted received_relay.jpg (from Test B).
   - **Screenshot 3:** The clean received.jpg after your code fix, successfully transferred through the relay.
   - **Screenshot 4:** The result of the Final check.
   - **Requirement:** Both screenshots must include the **System Date and Time** in the taskbar/menu bar to prove they were taken by you.
   - A brief explanation (1 paragraph) of how your Buffer logic works.

## Final Check

Your solution is correct if you can run the following command and get a perfect image:

```
# The Ultimate Test
python relay.py --loss 0.3 --reorder 0.2
```

*(This tests both Part 1 and Part 2 simultaneously!)*

# 6. Grading

Your code will be tested for correcting Part 1 and Part 2 errors.
Part 1 – 40%
- Code works and test is successful (corrects packet loss) – 40%

Part 2 – 40%
- Code works and test is successful (corrects packet ordering) – 40%

Correct and quality Documentation
- Adequate screenshots, code on repo following the documentation requirements – 20%.