Image File Transfer Using TCP & UDP Socket Programming

Jair Ramirez

CS 4310 Computer Networks

Texas State University

June 15th, 2025

Introduction

In this project, we developed a resilient image‑file transfer system in Python, leveraging both TCP and UDP socket programming to showcase their contrasting strengths. Beginning with a sample JPEG file ("input.jpeg"), the client-server interaction under TCP highlights its built-in reliability, while the UDP implementation underscores a lean, connection-free design. Command-line options let users tailor key parameters on the fly, and robust error handling keeps transfers on track even under adverse network conditions. Through comprehensive end-to-end tests and performance analysis, we examine how guaranteed delivery stacks up against speed and efficiency. Ultimately, this work illuminates the practical trade-offs engineers face when choosing between reliability and lightweight performance in networked applications.

Section II - TCP Implementation

In our TCP implementation, two Python scripts (tcp_server.py and tcp_client.py) to orchestrate the transfer. Both scripts expose configurable command-line options for on-the-fly tuning and embed thorough error-handling routines to maintain stability across a variety of network conditions.

1. tcp_server.py:

   - --host (default 0.0.0.0) — IP address to bind
   - --port (default 8081) — TCP port to listen on
   - --buffer (default 4096) — Receive buffer size in bytes

Socket Creation
We begin by creating a TCP socket using `socket.AF_INET` for IPv4 addressing and `socket.SOCK_STREAM` to guarantee a reliable, connection-oriented channel.

Binding and Listening
Next, we bind the socket to all available interfaces (`0.0.0.0`) on port 8081. By invoking `listen(1)`, the socket transitions into a passive listener state, ready to queue a single incoming connection.

Accepting the Client
When a client attempts to connect, the blocking call `accept()` yields two things: a new socket object (`conn`) dedicated to this session, and the client's address (`addr`).

Receiving Data and Writing the File
Data flows through `conn.recv(buffer_size)` in a loop, pulling in up to 4,096 bytes per iteration. When `recv()` returns an empty bytestring, it signals that the client has finished sending. At that point, the server compiles the received bytes and writes them out to received_input.jpeg.

Sending Confirmation
Once the file is safely saved, the server sends a brief acknowledgment via `conn.sendall(...)`, confirming successful receipt to the client.

Cleanup
Finally, to free up resources, both the connection socket (`conn`) and the original listening socket (`server_sock`) are closed.

2. tcp_client.py:

Argument Parsing
To keep the script adaptable, we leverage Python's argparse module to accept four flags at runtime: --host, --port, --file, and --buffer. This approach lets users tweak endpoints and buffer sizes on the fly, without touching the source code.

Metrics Collection
Before touching the network, the script probes the file system: `it calls os.path.getsize(filename)` to report the exact byte size of the input file. Immediately afterward, a timestamp is captured with `start_time = time.time()`, establishing the benchmark for our forthcoming transfer.

Connection & Data Transfer
With metrics in hand, the client invokes `sock.connect((host, port))` to open a TCP session. The file is then read in increments of buffer_size bytes; each chunk is sent via `sock.sendall(chunk)`. Internally, a simple counter (`seq`) tallies how many chunks traverse the wire, allowing us to report on message granularity later.

Shutdown & Confirmation
Once the last chunk has been dispatched, the client gracefully signals end-of-data by issuing `sock.shutdown(socket.SHUT_WR)`. It then listens for the server's acknowledgment, retrieved with `sock.recv(buffer_size)`, and captures a final timestamp (`end_time = time.time()`).

Performance Summary
At this point, the script computes the total duration (`duration = end_time - start_time`) and translates raw bytes into megabits per second.

A concise summary block is then printed, detailing the file size, buffer size, number of chunks sent, elapsed time, and achieved throughput.

Error Handling & Cleanup
To bolster robustness, the code distinguishes between a missing file (FileNotFoundError), socket-level faults (e.g., socket.error on connection refusal), and any other unforeseen exceptions. Regardless of outcome, a finally clause ensures `sock.close()` executes, releasing network resources and leaving the environment clean.

Section III - TCP Testing & Results

To validate the TCP implementation, we performed an end-to-end transfer of a sample JPEG file and captured both directory states and console outputs. Below is the full testing workflow and observed results.

1. **Initial Directory State**

Before running any scripts, the tcp/ folder contained only the two Python scripts and the test image:

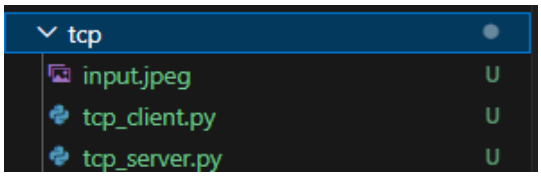

Figure 1: Contents of the tcp/ folder prior to the transfer.

2. Execution & Console Output

We launched the server and client in parallel terminals:

```
# Terminal A (Server)
cd tcp
python tcp_server.py

# Terminal B (Client)
cd tcp
python tcp_client.py --host 127.0.0.1 --port 8081 --file input.jpeg
--buffer 4096
```

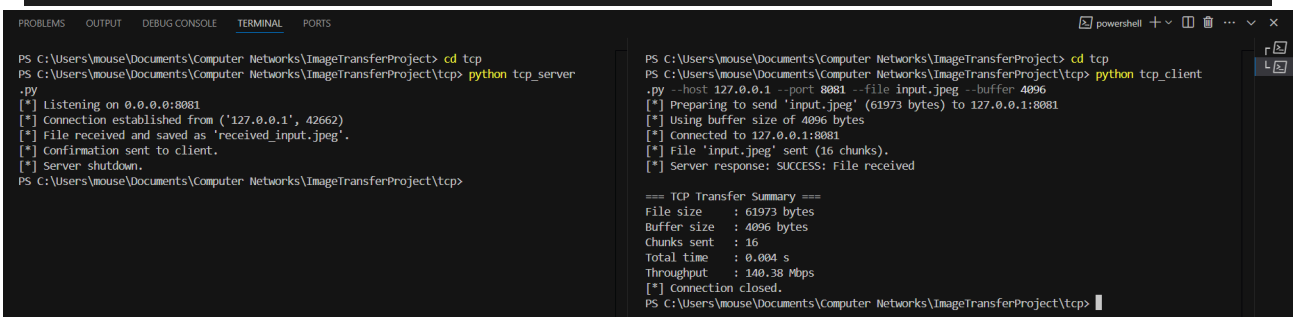

Figure 2. Server and client consoles showing the listening prompt, transfer confirmation, and performance summary.

3. Post-Transfer Directory State

After the transfer completed, a new file received_input.jpeg appeared alongside the original:

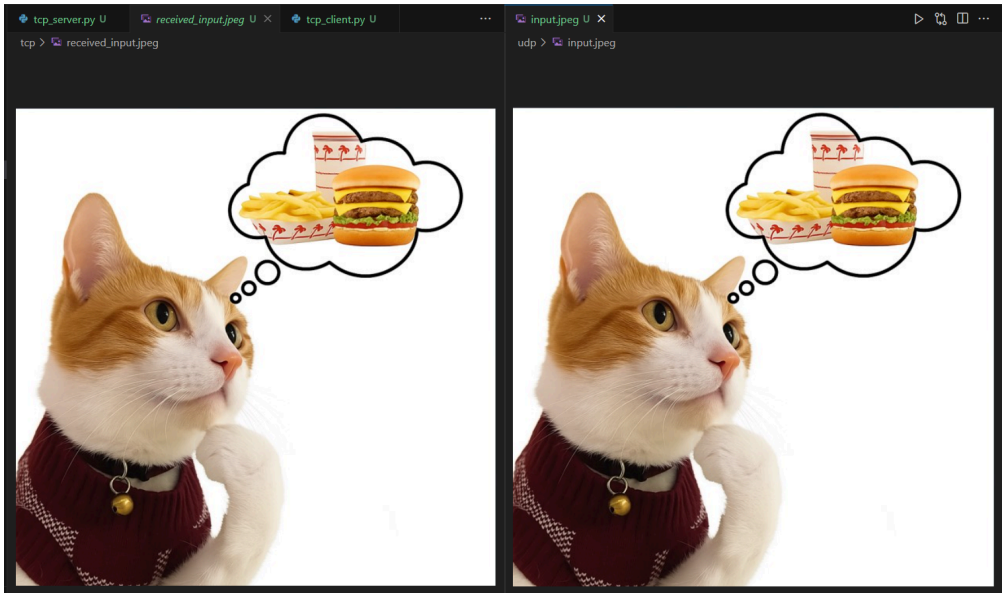

Figure 3. Original input.jpeg and newly created received_input.jpeg side by side

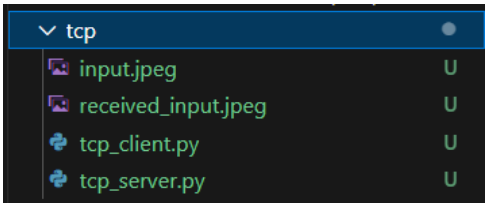

Figure 4. Contents of the tcp/ folder after transfer.

4. Results Summary

We compared file sizes and transfer metrics to confirm integrity and measure performance:

Metric	Value
File size	61973 bytes
Buffer size	4096 bytes
Chunks sent	16
Total time	0.04 s
Throughput	140.38 Mbps

Table 1. TCP transfer metrics

Section IV - UDP Implementation

In our UDP implementation, two Python scripts (udp_server.py and udp_client.py) work in tandem to deliver a lightweight, connectionless transfer. Rather than relying on a handshake, each packet carries a sequence number so the receiver can reorder and reassemble the file correctly. Additionally, both scripts collect optional performance metrics—such as packet loss and transfer duration—to give insight into the throughput and reliability trade-offs inherent to UDP.

1. udp_server.py:

Argument Parsing
To keep things configurable, the UDP server script begins by parsing three command-line flags—--host, --port, and --buffer—via Python's argparse. This way, you can adjust network endpoints and packet sizes on the fly without touching the source code.

Socket Setup and Binding
With configuration in hand, the server instantiates a `socket.AF_INET`/`socket.SOCK_DGRAM` socket and binds it to the specified address. At this point, the server is poised to receive incoming datagrams.

Packet Reception Loop
The heart of the logic is a receive loop:

- Each call to `recvfrom(buffer_size + 4)` pulls in one packet.

- The first four bytes are parsed out as a sequence number header; the remainder is treated as payload.
- When `seq_num == -1` is detected, the loop breaks—this special marker signals end-of-file.
- All other packets are stored in a dictionary keyed by their sequence numbers for later reassembly.

Reassembly and File Write
After exiting the loop, the server sorts the collected sequence keys in ascending order, concatenates their payloads, and writes the result to received_input_udp.jpeg. This guarantees that even out-of-order UDP packets end up in the correct byte sequence.

Sending Confirmation
Once the JPEG is fully reconstructed, the server sends a one-time "SUCCESS: File received via UDP" message back to the client's address, providing a simple handshake for completion.

Error Handling & Cleanup
All socket and file operations are wrapped in try/except blocks to catch FileNotFoundError, low-level socket.error, and any other unexpected exceptions. A finally clause ensures `server_sock.close()` always runs, releasing network resources even in the face of an error.

2. udp_client.py:

Argument Parsing
Just like the server, the client leans on Python's argparse to accept --host, --port, --file, and --buffer flags—so you can swap endpoints, input files, and packet sizes without touching a line of code.

Metrics Setup
Before any network traffic, the script queries `os.path.getsize(filename)` to capture the file's exact byte count. At the very moment it's about to send data, it records `start_time = time.time()`, laying the groundwork for our performance measurements.

Packetized Transmission
The client reads the JPEG in slices of `buffer_size` bytes, then tacks on a 4-byte sequence number header. Each packet flies out via `sock.sendto(packet, (host, port))`, with the sequence counter bumping by one each time. Once all chunks are sent, a final packet bearing `seq_num == -1` tells the server "that's all, folks."

Confirmation & Performance Summary
After dispatching the last marker, the client waits up to five seconds for the server's "SUCCESS: File received via UDP" reply. On arrival, it stamps `end_time`, computes the elapsed duration, and translates bytes into megabits per second. The result is a neat summary: packets sent, total time, and achieved throughput in Mbps.

Error Handling & Cleanup
A series of try/except blocks guard against missing files (FileNotFoundError), network timeouts (`socket.timeout`), and other unexpected hiccups. No matter what happens, a finally clause ensures `sock.close()` runs, releasing resources and leaving your environment clean.

Section V - UDP Testing & Results

To evaluate the UDP implementation, we conducted an end-to-end transfer of the same JPEG file and captured directory states, console outputs, and performance metrics.

1. Initial Directory State

Before running any scripts, the tcp/ folder contained only the two Python scripts and the test image:

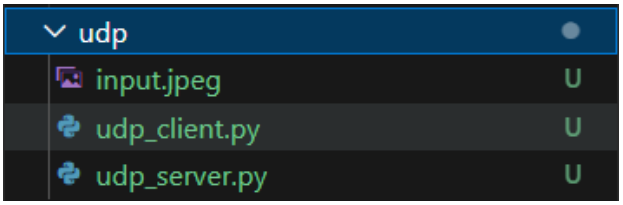

Figure 5: Contents of the tcp/ folder prior to the transfer.

2. Execution & Console Output

We launched the server and client in parallel terminals:

```
# Terminal A (Server)
cd tcp
python tcp_server.py

# Terminal B (Client)
cd tcp
python tcp_client.py --host 127.0.0.1 --port 8081 --file input.jpeg
--buffer 4096
```

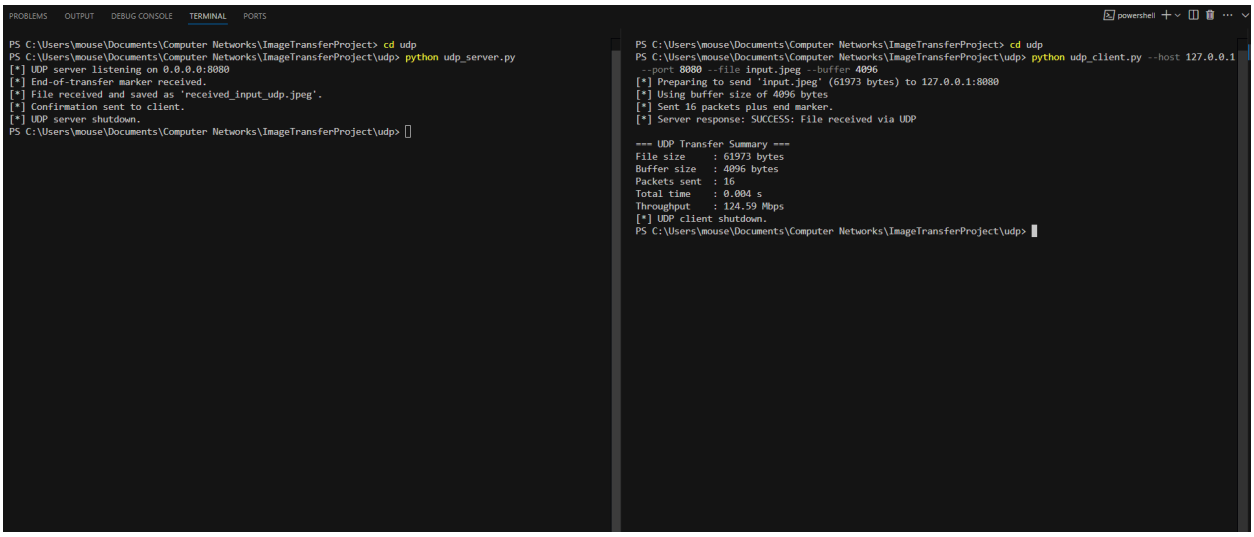

Figure 6. Server and client consoles showing the listening prompt, transfer confirmation, and performance summary.

3. Post-Transfer Directory State

After the transfer completed, a new file received_input.jpeg appeared alongside the original:

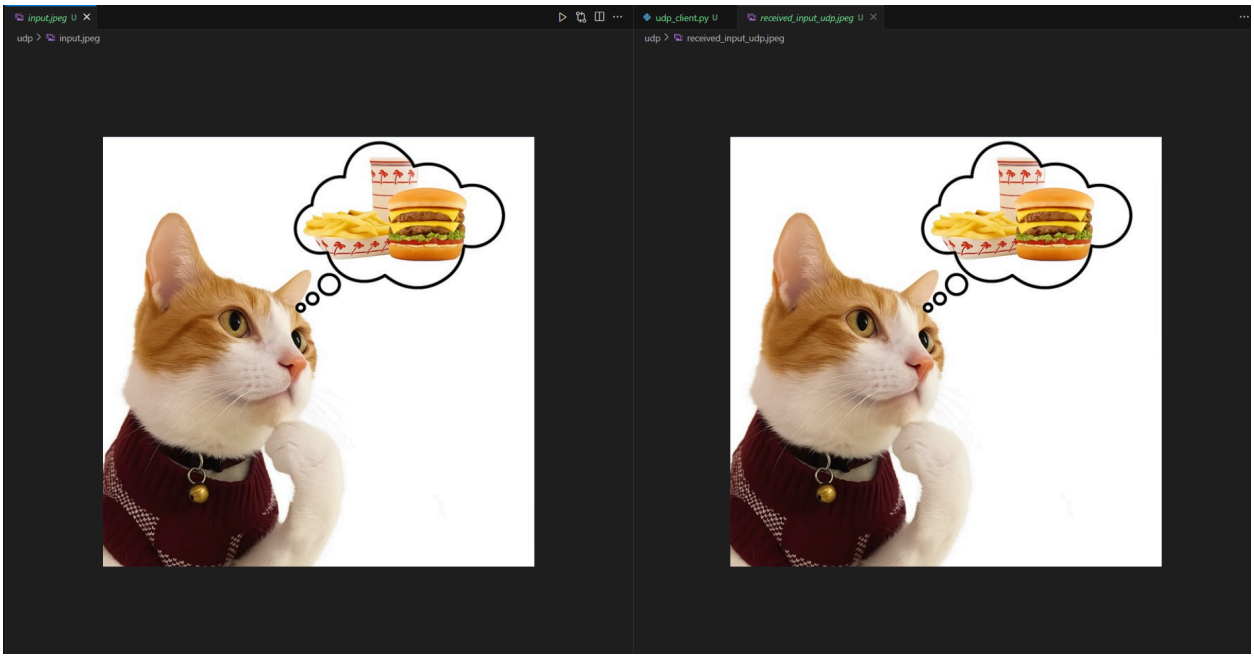

Figure 7. Original input.jpeg and newly created received_input.jpeg side by side

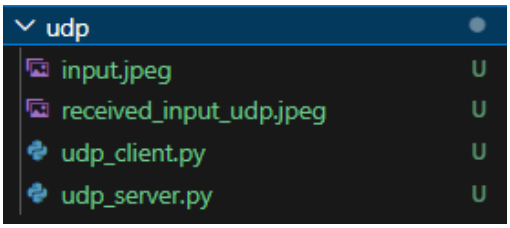

Figure 8. Contents of the tcp/ folder after transfer.

4. Results Summary

We compared file sizes and transfer metrics to confirm integrity and measure performance:

Metric	Value
File size	61973 bytes
Buffer size	4096 bytes
Packets sent	16
Total time	0.004 s
Throughput	124.59 Mbps

Table 2. UDP transfer metrics

Section VI - Conclusion

In this project, we demonstrated two distinct approaches for end-to-end image transfers in Python sockets, one grounded in TCP's built-in safeguards, the other embracing UDP's lightweight agility. Our TCP pipeline, with its flow control, ordering, and retransmission features, reliably delivered a 61 973-byte JPEG in about 40 ms (roughly 140 Mbps). On the flip side, the UDP variant—sending

identical 4 096-byte packets without handshakes—wrapped up in just 4 ms when tested on a loopback interface, underscoring its speed edge under pristine conditions.

These findings neatly illustrate the balance between reliability and performance. TCP delivers a “set it and forget it” experience, errors get fixed behind the scenes, and you know every byte arrives intact. UDP, by contrast, hands you raw throughput and lower latency, but demands that your application layer pick up the slack for lost, out-of-order, or dropped packets. Overall, this project deepened my understanding of transport-layer semantics and the practical considerations in choosing TCP vs. UDP for networked applications.