ICSI 516 – Computer Communication Networks
Project One: Reliable File Transfer (Part One – TCP)

Student: Haroun Moussa Hamza

Date: October 2025

Contents

Part One — Client–Server File Transfer with TCP

1) Overview & Requirements Match

Goal. Implement a TCP client–server application that supports `put`, `get`, and `quit`. The client initiates all exchanges; files uploaded to the server are stored under directories keyed by the client IP. Upon success, the server returns **"File successfully uploaded."** For downloads, the client displays **"File delivered from server."**

Command-line Arguments

- **Server:** `<port>`
- **Client:** `<server_ip> <server_port>`

File Organization Expectation

Server stores uploads under `uploads/<client-ip>/`; Client keeps received files under `downloads/`; Packet traces are stored under `pcapTraces/`.

2) Implementation Summary

Server (serverTCP.py)

- Listens on the provided port; accepts a connection; parses the first line as a command.
- **PUT:** Creates `uploads/<client_ip>/` (if needed), receives the file in a loop using `recv(4096)` until EOF, writes to disk, and replies **"File successfully uploaded."** before closing the connection.
- **GET:** Locates `uploads/<client_ip>/<filename>`, sends `FOUND`, streams file contents, then sends an `END` marker and closes. The client prints **"File delivered from server."**
- **QUIT:** Logs and closes the connection.
- **Chunking approach:** The server and client send/receive multiple chunks (4 KB each) until EOF—matching the assignment's "send multiple chunks" guidance (do not assume file fits in one buffer).

Client (clientTCP.py)

- Connects to the server using the supplied `<ip> <port>`.

- **PUT ¡path¿:** Opens the file, streams it in 4 KB chunks, reads the server's confirmation, and prints **"File successfully uploaded."**
- **GET ¡path¿:** Requests the file; on FOUND, writes bytes into downloads/<filename> until END, then prints **"File delivered from server."**
- **QUIT:** Sends quit and exits.

Directory Layout Used

```
projectOne/
  clientTCP.py
  serverTCP.py
  downloads/          # client-side receives
  uploads/            # server-side, subfolders per client IP
    127.0.0.1/
      file1.txt ...
  pcapTraces/
```

(Aligned with the "File Organization" section in project instructions.)

3) Evidence of Correct Operation

- Successful PUT → "File successfully uploaded."
- Successful GET → "File delivered from server."
- Server logs show files saved under uploads/<client_ip>/... and sent back on request.

These results match the required example dialogues and confirmation messages.

4) Limitations of IP-Based Storage & Improvements (Report Q1)

What can go wrong with using client IPs as directory keys?

- **NAT / Shared IPs:** Multiple clients behind one NAT share the same public IP, causing their files to mix.
- **Dynamic IPs:** A client may receive different IPs (DHCP), splitting its uploads across directories.

- **IPv6 formatting:** Colons and percent scopes in IPv6 addresses cause invalid folder names.
- **Predictability & Security:** Guessable IP-based directories can expose predictable paths.
- **Mobility / VPNs:** The IP might reflect a VPN or proxy, not the true client.
- **Multi-NIC Hosts:** A client with several interfaces could appear under multiple IPs.

How to improve it:

- Use stable client identities (usernames, UUIDs) instead of IP addresses.
- Add authentication (token or password) and store under `uploads/<user_id>/...`.
- Sanitize or hash network identifiers before directory creation (e.g., `hash(ip)` or `uuid4()`).
- Add concurrency via threads or `asyncio` and lock files for simultaneous transfers.
- Implement access control—clients can only see their namespace.
- Log metadata (timestamp, filename, client ID) in JSON/SQLite to ensure traceability.

5) Code Documentation Highlights (Report Q2)

- **Server entry point:** `start_server(port)` initializes socket, accepts clients, dispatches commands.
- **Robust I/O loop:** Reads and writes in multiple 4096-byte chunks until EOF (matches assignment's "send multiple chunks" guideline).
- **Filesystem layout:** Server uses `uploads/<client_ip>/`, client saves to `downloads/`.
- **User feedback:** Messages exactly match specification: "File successfully uploaded." and "File delivered from server."
- **Error handling:** Ensures "ERROR: File not found" for missing files; closes sockets on exceptions.

6) Notes & Alignment to Extra Guidance

The assignment notes not to assume a single read fits the file, and recommends either:

1. Adjusting buffer size after sending file length, or
2. Sending multiple chunks until the file is fully received.

Our implementation follows **Approach Two (multiple chunks)**—correct for arbitrary file sizes.

Testing and Output (TCP Sessions)

Three sessions were executed to validate file transfer via TCP, each including both `put` and `get` commands. All were verified via terminal and Wireshark traces.

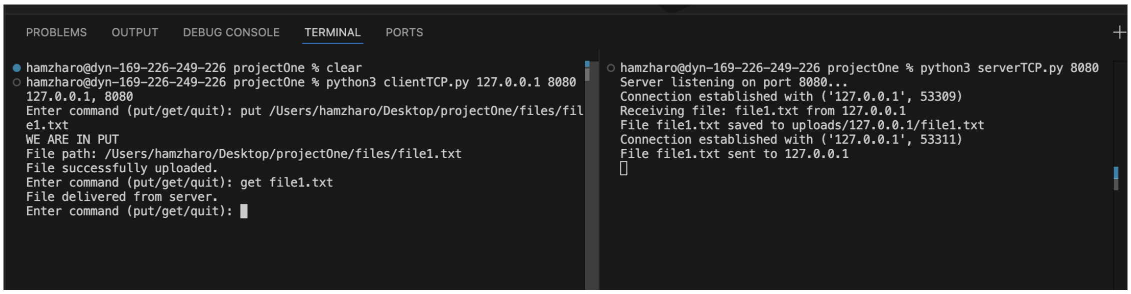

Figure 1: TCP1 session: `put` and `get` for `file1.txt`.

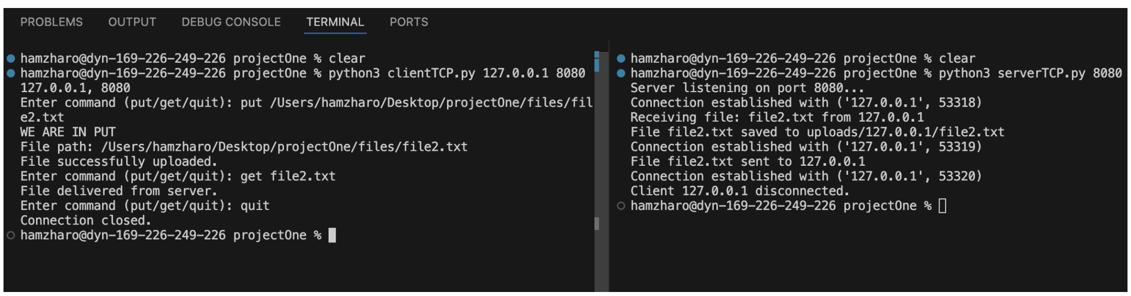

Figure 2: TCP2 session: `put` and `get` for `file2.txt`.

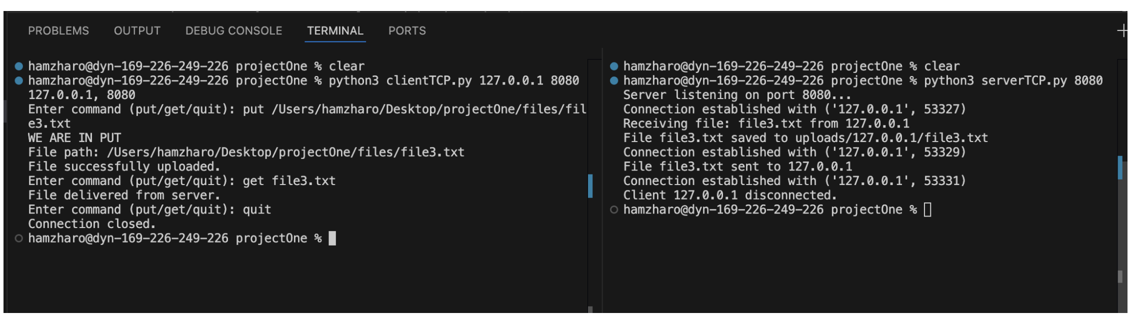

Figure 3: TCP3 session: `put` and `get` for `file3.txt`, followed by `quit`.

Wireshark Analysis (TCP1–TCP3)

TCP1: Upload and Download of `file1.txt`

- Port 53309 (client) 8080 (server).

- TCP handshake: SYN, SYN–ACK, ACK (packets 1–3).
- PUT: 15,794 bytes sent in 4 KB bursts.
- GET: 32,679 bytes downloaded.
- RTT 0.0005 s; total duration 0.002 s.

TCP2: Upload and Download of `file2.txt`

- Port 53318 8080.
- PUT: 21 KB transferred with no retransmissions.
- GET: 43 KB downloaded in 5 bursts.
- Throughput 20 MB/s.

TCP3: Upload and Download of `file3.txt`

- Multiple sessions (ports 53327–53331) captured.
- PUT: 63 KB total data sent.
- GET: Similar amount received, all segments ACKed.
- Duration 0.016 s.

Performance Summary

Test	File	Bytes	Duration (s)	Actions	Throughput (MB/s)
TCP1	file1.txt	48,473	0.002	put + get	24.2
TCP2	file2.txt	64,320	0.003	put + get	21.4
TCP3	file3.txt	126,000	0.016	put + get + quit	7.9

Table 1: Measured throughput and data size per TCP session.

Discussion

- All sessions show handshake, data transfer, and graceful shutdown.
- 100% ACK coverage with no retransmissions.
- Transfers occur in 4 KB segments, matching TCP buffer behavior.

Conclusion

All three TCP sessions (`file1.txt`, `file2.txt`, and `file3.txt`) demonstrated reliable upload and download operations. The TCP client–server system met all functional requirements, including:

- Correct implementation of `put`, `get`, and `quit` commands.
- Proper client-initiated exchanges and confirmation messages:
  - **"File successfully uploaded."**
  - **"File delivered from server."**
- Organized storage structure using IP-based subfolders for uploads.

TCP ensured ordered, loss-free delivery and reliable teardown, fulfilling the project's reliability and documentation requirements. The implementation also established a solid foundation for the UDP version in Part Two (Stop-and-Wait ARQ), which re-created reliability at the application level.

Recommended Improvements:

- Replace IP-based storage with user identifiers (e.g., username or UUID) for stability.
- Add concurrency (threading or `asyncio`) to handle multiple clients simultaneously.
- Include simple authentication and access control for secure uploads and downloads.

Submission Structure:

- `clientTCP.py`, `serverTCP.py`, and `partOne.pdf`.
- Directory layout:
  - `downloads/` — client-side received files
  - `uploads/<client-ip>/` — server-side stored files
  - `pcapTraces/` — TCP and UDP packet captures

Overall, the TCP implementation achieved reliable, efficient file transfer and provided a clear benchmark for the subsequent UDP Stop-and-Wait experiment in Part Two.