FTP-Lite: QUIC-based File Transfer Protocol

Course: CS544 - Computer Networks Project
Professor: Brian Mitchell
Student Name: Sarthak Vinod Shrungare
University ID:14744552

Overview

FTP-Lite is a custom stateful file-transfer protocol implemented using QUIC, designed to deliver secure, reliable, and efficient file uploads from clients to a central server. It utilizes modern transport technology (QUIC and TLS) along with application layer design principles. It uses deterministic finite automata (DFA), explicitly structured PDUs, and a stateful session. This project has both an aspect of practical protocol design, along with an aspect of robustness (e.g., asynchronously managing multiple clients, finding server dynamically through UDP broadcast, automated protocol testing through pytest. It serves as a realistic prototype of how lightweight file transfer systems can be designed and implemented with modern, secure, and extensible architecture.

Project Structure

```
FTP-Lite/
├── server.py
├── client.py
├── protocol.py
├── requirements.txt
├── cert.pem
├── key.pem
├── run_server.sh
├── run.sh
├── Makefile
├── tests/
│   └── test_ftplite.py
├── Result/
├── myfile.txt
├── fileA.txt
└── fileB.txt
```

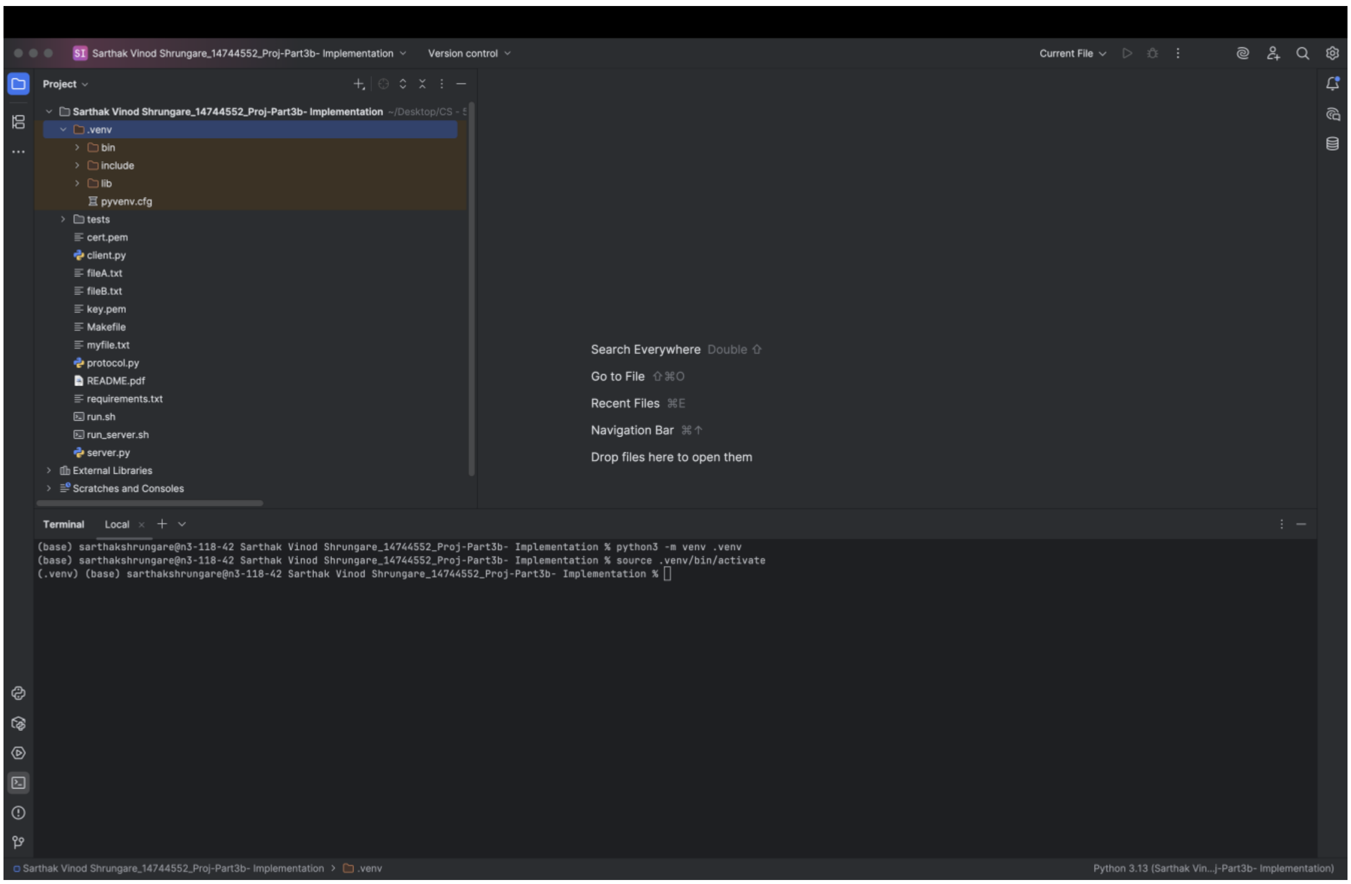
Installation & Setup

1. Prerequisites:

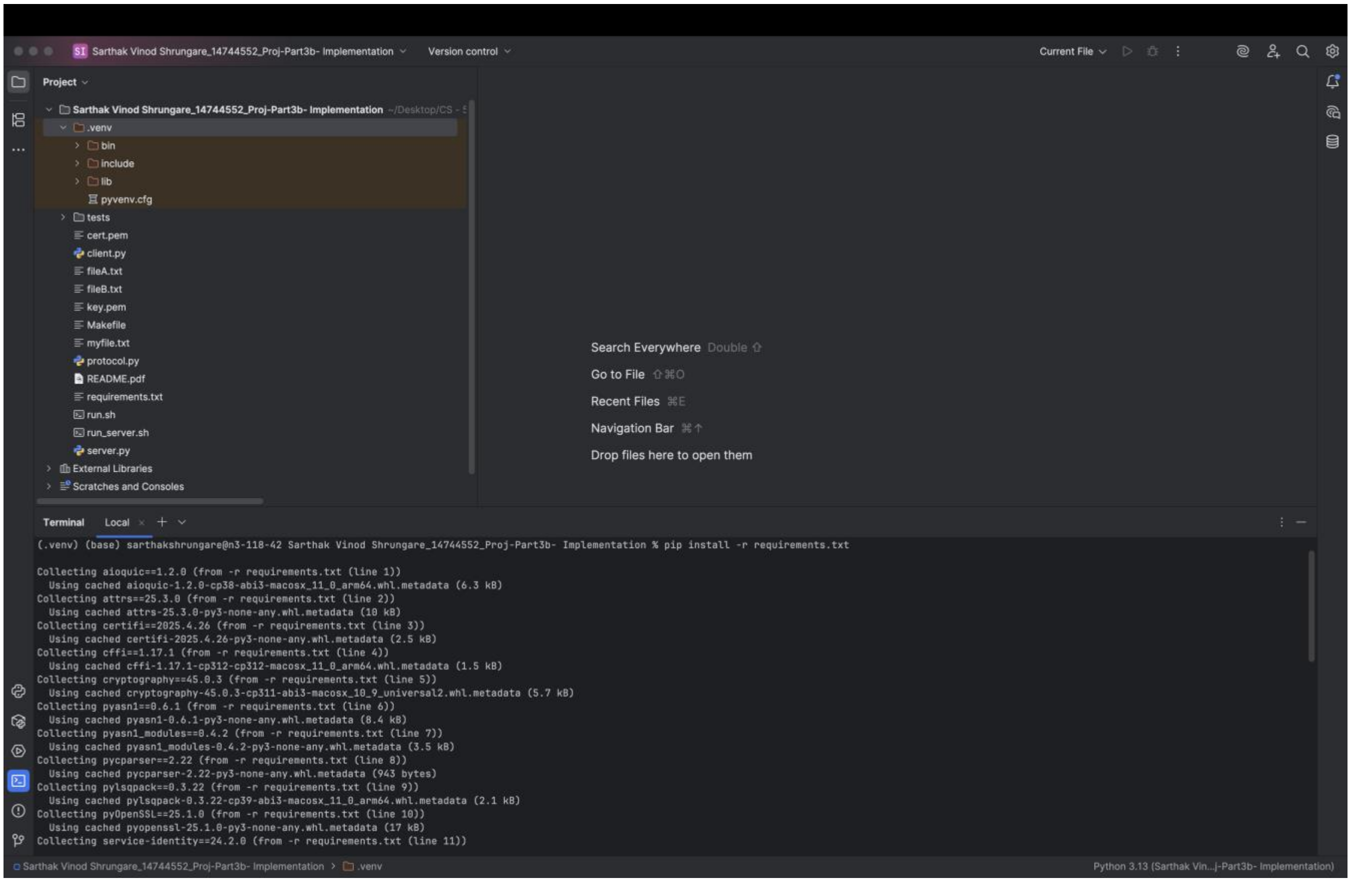
- Python 3.13+
- pip package manager
- OpenSSL for generating TLS certificates

2. Setup Steps:

- python3 -m venv .venv
- source .venv/bin/activate



- pip install -r requirements.txt



3. TLS certificates: cert.pem, key.pem are pre-generated and included in the project.

Platform Compatibility

This project was developed and tested on macOS. All scripts (*.sh), Makefile targets, and Python commands are fully compatible with Linux-based systems.

Protocol Description

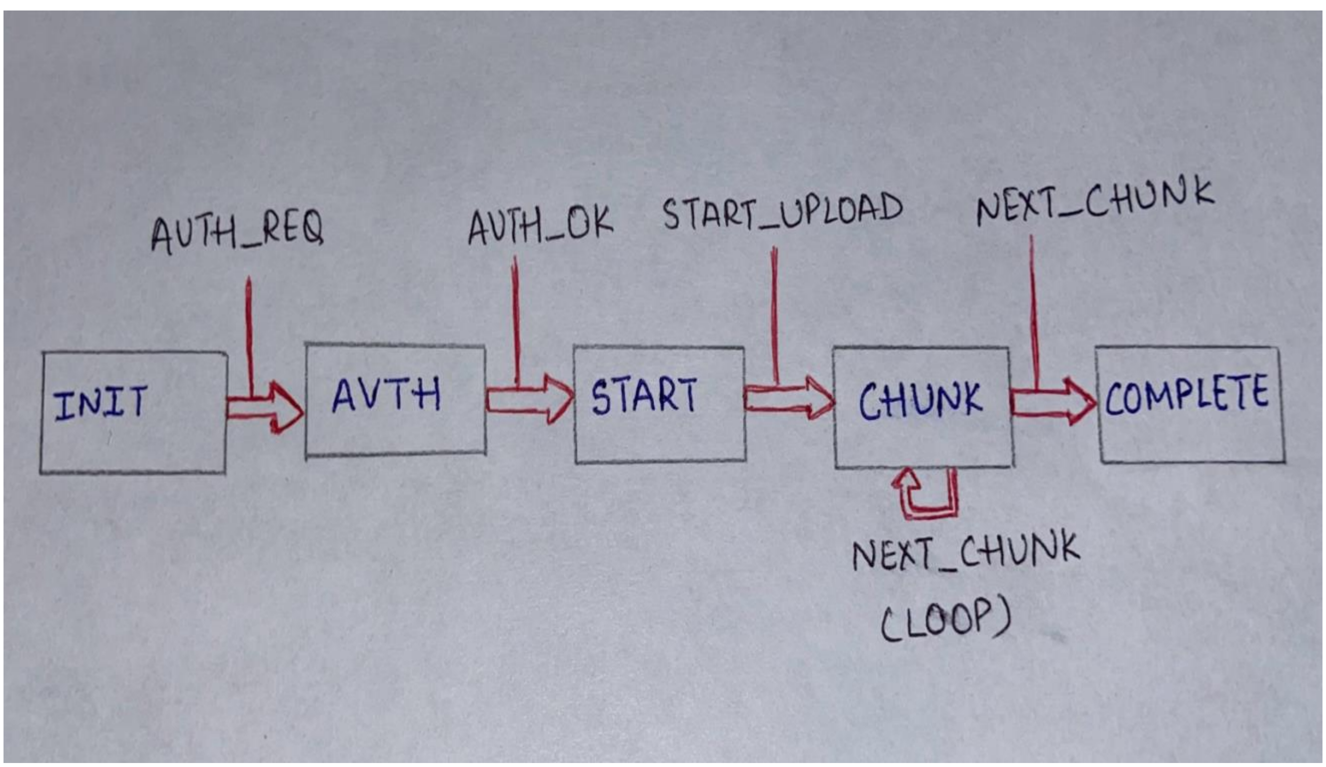
1. Service Description:

- FTP-Lite is a reliable client-server protocol that allows users to upload files to a server. It uses QUIC for transport and supports authentication, concurrency, and security using TLS.

2. DFA Protocol States:

- INIT → AUTH → START → SEGMENT → COMPLETE
- Each state governs how messages are sent and ensures that transitions occur in a validated and secure manner.

3. DFA Diagram:



4. Message Definitions (PDU's):

- Messages are defined with structured binary layouts using big-endian formatting. Each PDU includes a header with version, type, and length fields, followed by message-specific data. Examples include:

- **INIT**: Starts the protocol session and declares version compatibility.
- **AUTH**: Sends username and password for client authentication.
- **START**: Notifies the server about the incoming file name and size.
- **SEGMENT**: Transfers the file content in binary chunks.
- **COMPLETE**: Marks the end of file transmission and triggers upload finalization.

PDU Type	Field	Size (bytes)	Description
Common Header	Version	1	Protocol version (0x01)
Common Header	Type	1	Message type (INIT, AUTH, etc.)
Common Header	Length	2	Length of payload in bytes (big-endian)
INIT	(No payload)	0	Client initiates the protocol session
AUTH	Username	Variable	UTF-8 encoded username
AUT	Password	Variable	UTF-8 encoded password
STAR	Filename	Variable	Name of the file to upload
START	File Size	8	Size of the file (big-endian unsigned int)
SEGMENT	Segment Number	4	Segment index for chunk (big-endian)
SEGMEN	Chunk Data	Variable	Binary data of the file chunk
ACK	Segment Number	4	Server acknowledges received segment
COMPLETE	(No payload)	0	Signifies end of file transmission
ERROR	Error Message	Variable	UTF-8 error message describing failure

5. Extensibility:

- Protocol versioning supported with a version field (currently 0x01).
- New message types or optional fields can be added while maintaining backward compatibility.

6. Authentication:

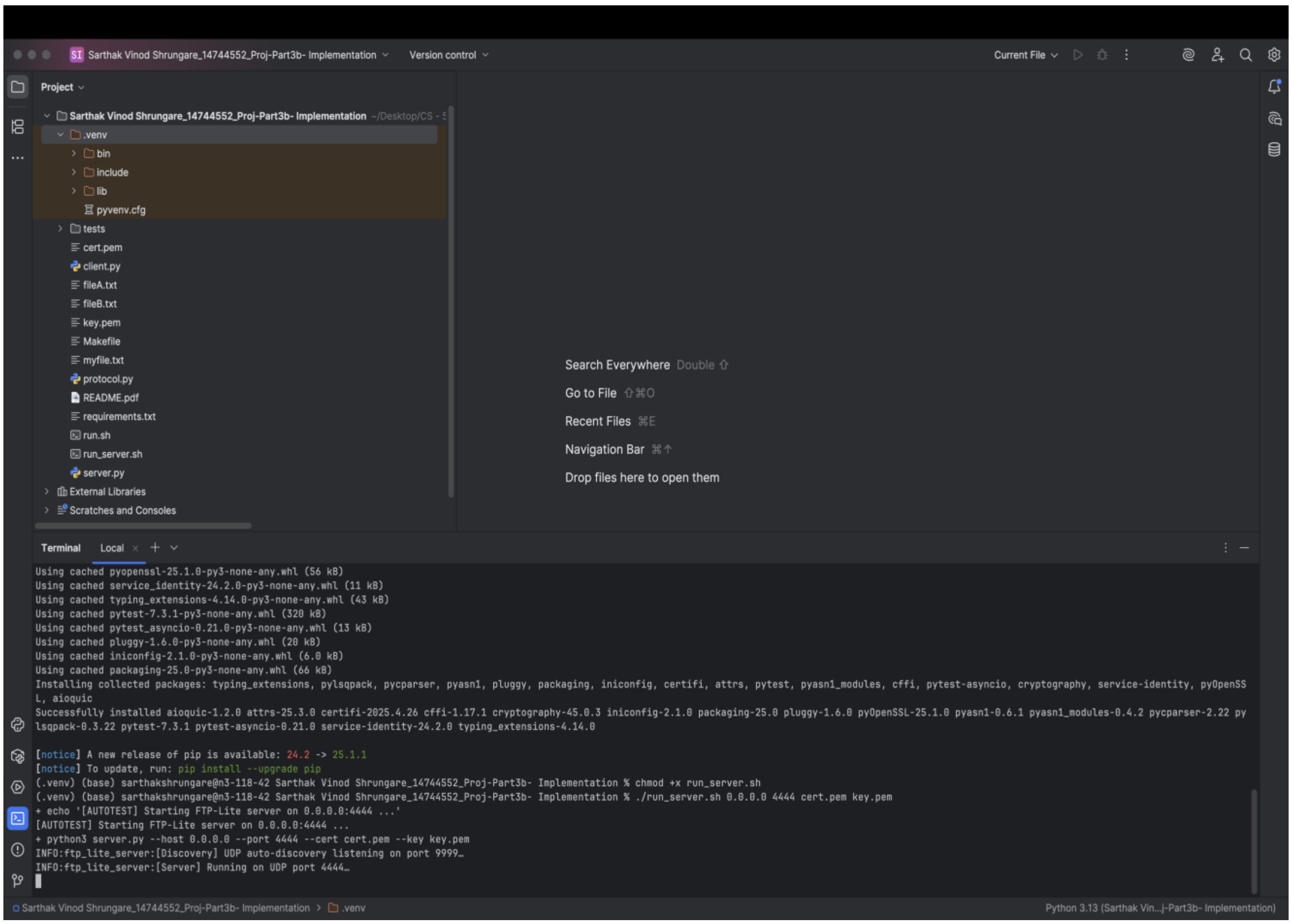
- Uses password-based login (e.g., user: bob, password: admin).
- Unauthenticated users are denied access with AUTH-ERR.
- In production, authentication should use hashed credentials and secure storage

.

HOW TO RUN:

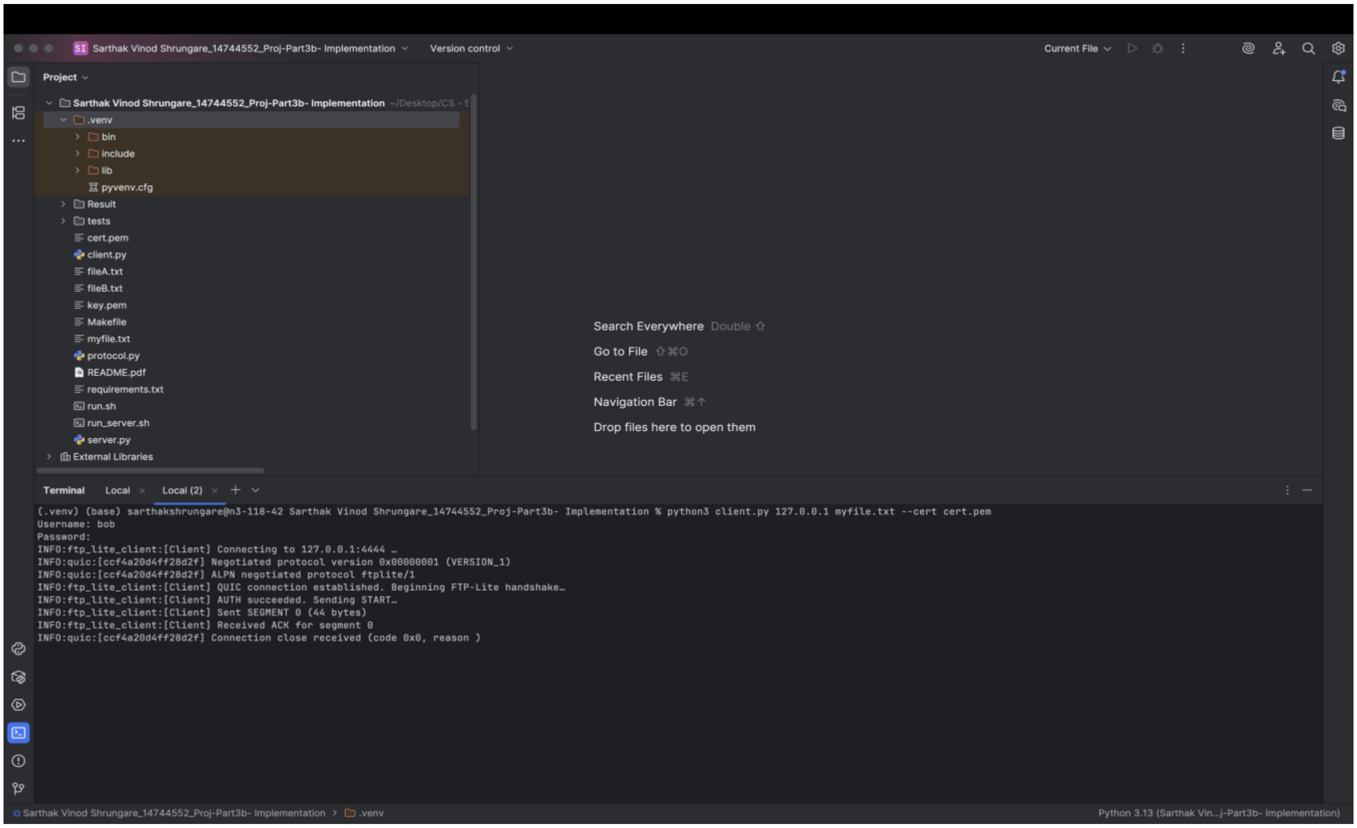
1. **Starting the Server:**

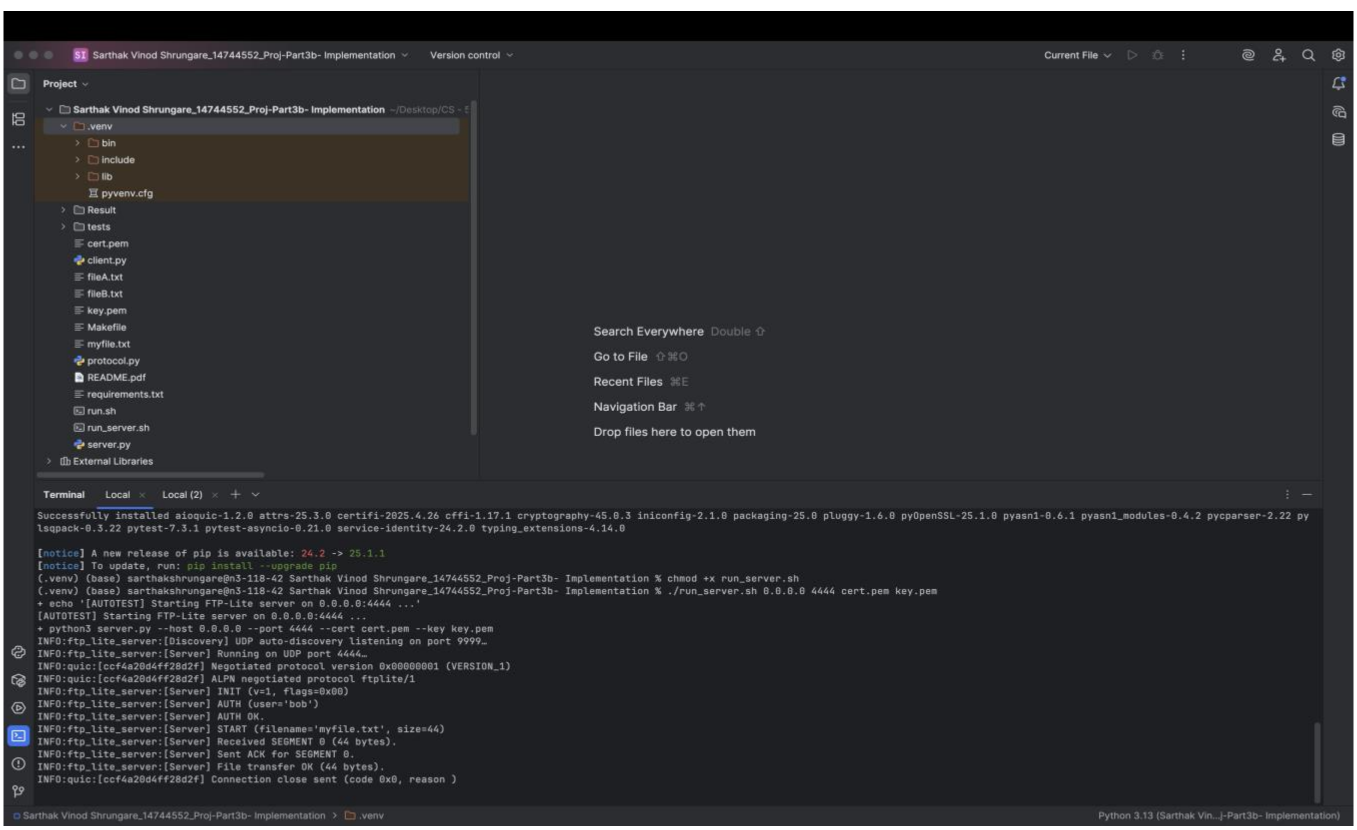
- chmod +x run_server.sh
- ./run_server.sh 0.0.0.0 4444 cert.pem key.pem



2. **Interactive Mode:**

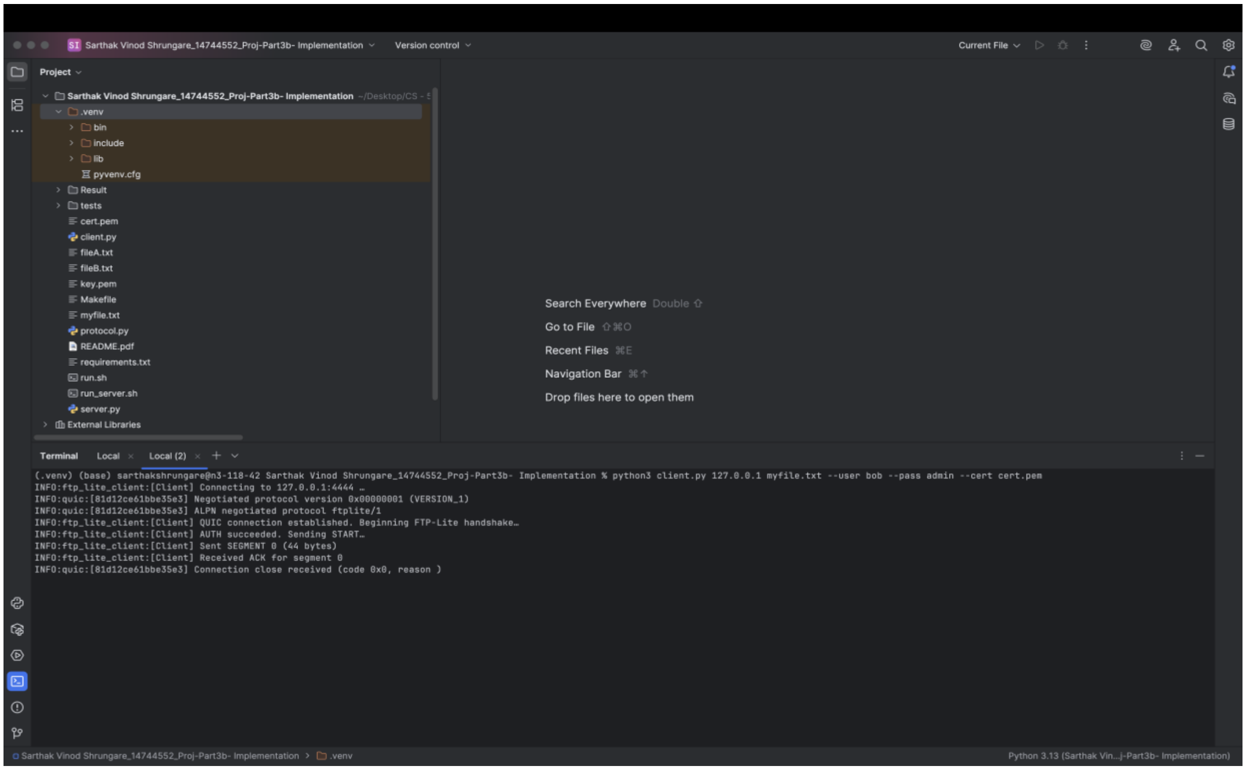
- python3 client.py 127.0.0.1 myfile.txt --cert cert.pem

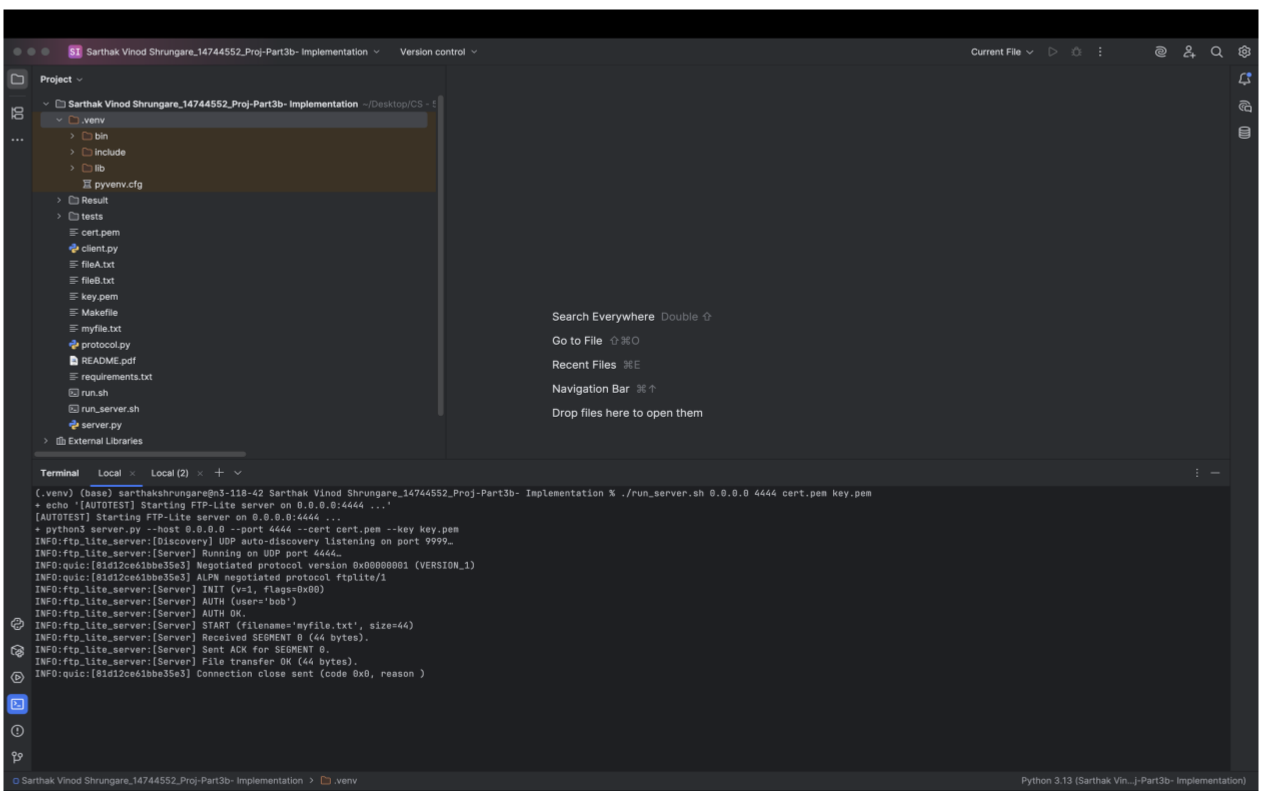




3. Running the Client:

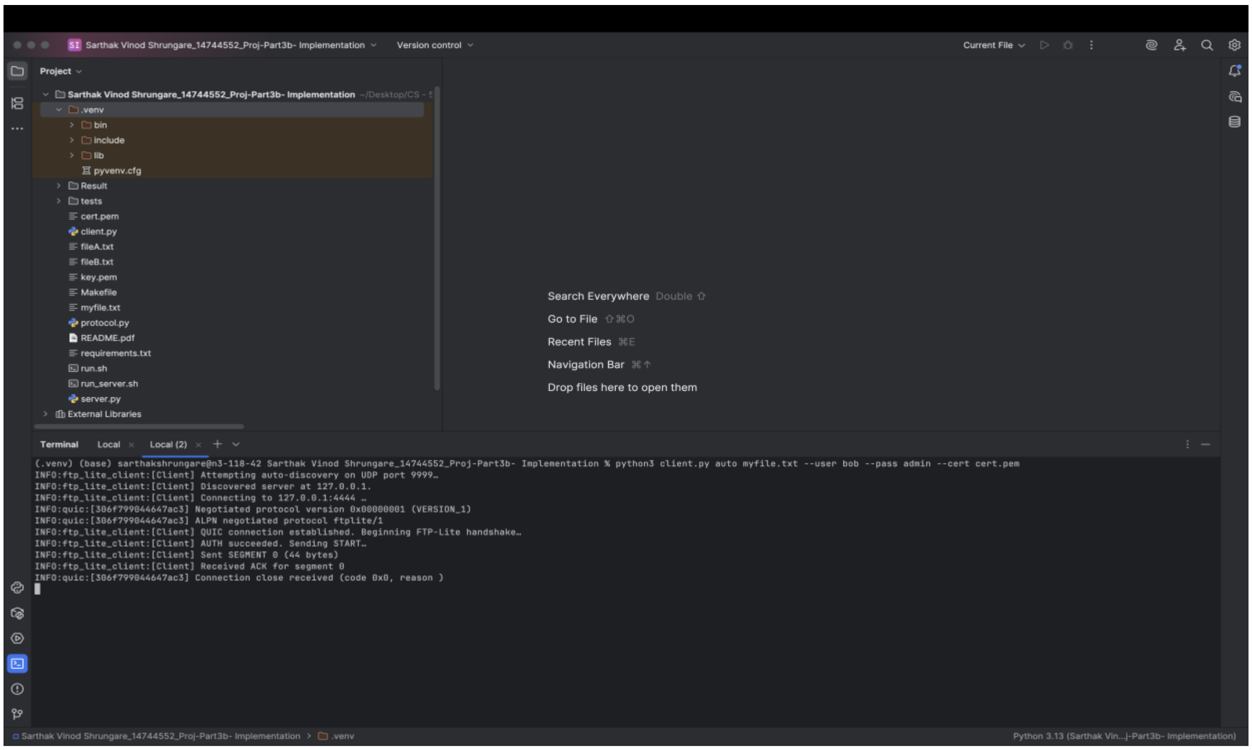
- python3 client.py 127.0.0.1 myfile.txt --user bob --pass admin --cert cert.pem

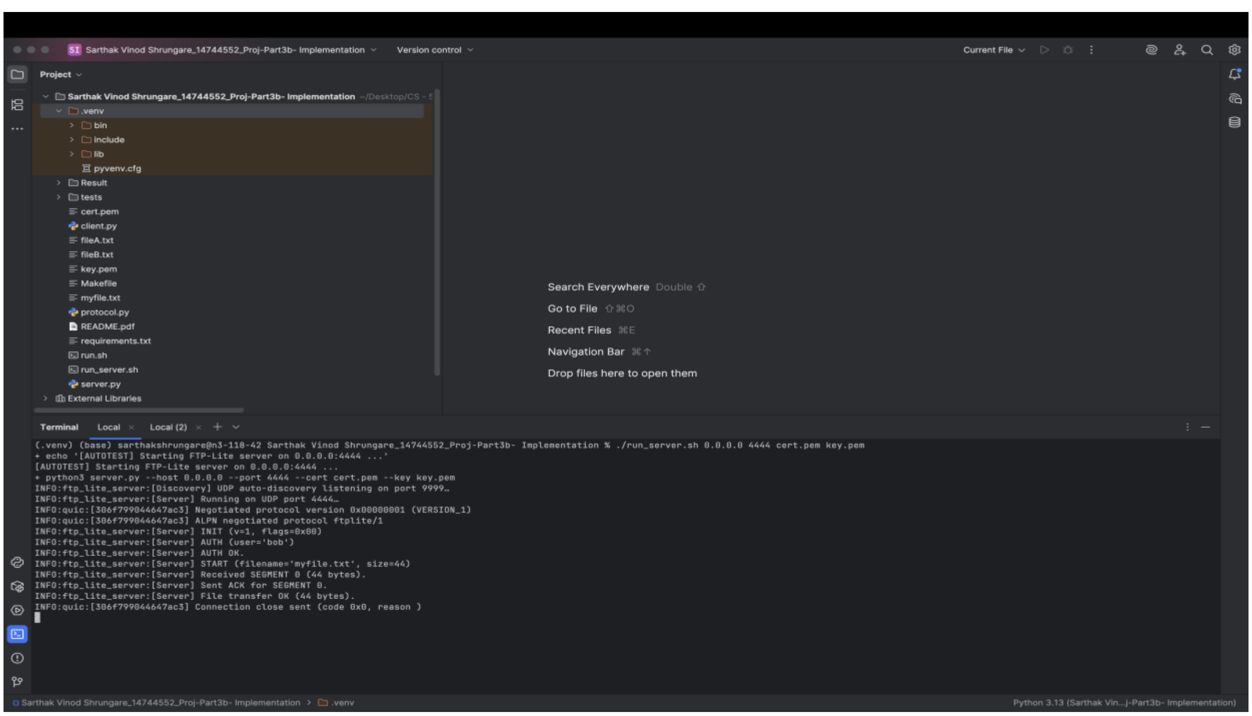




4. Auto Discovery:

- python3 client.py auto myfile.txt --user bob --pass admin --cert cert.pem



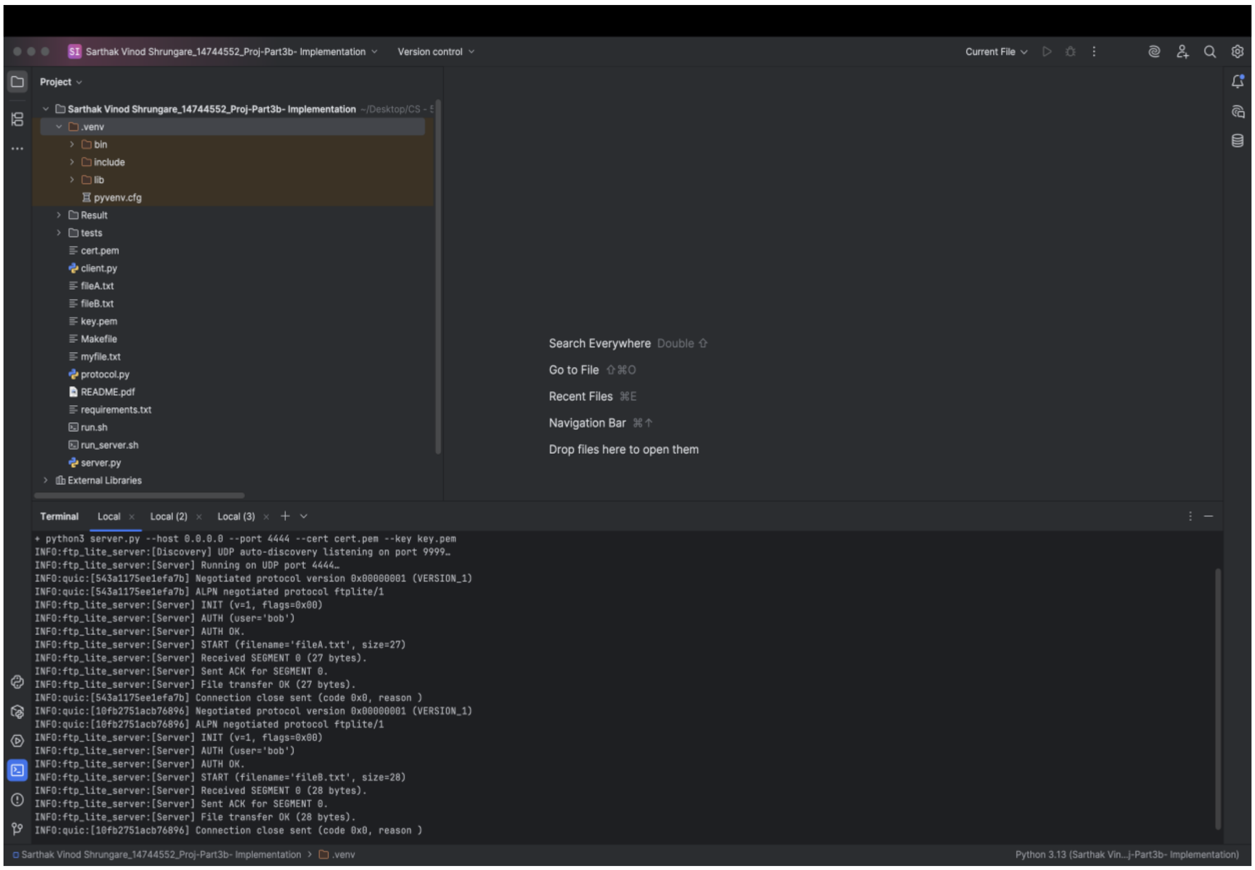


Testing Concurrency

Use 3 separate terminals:

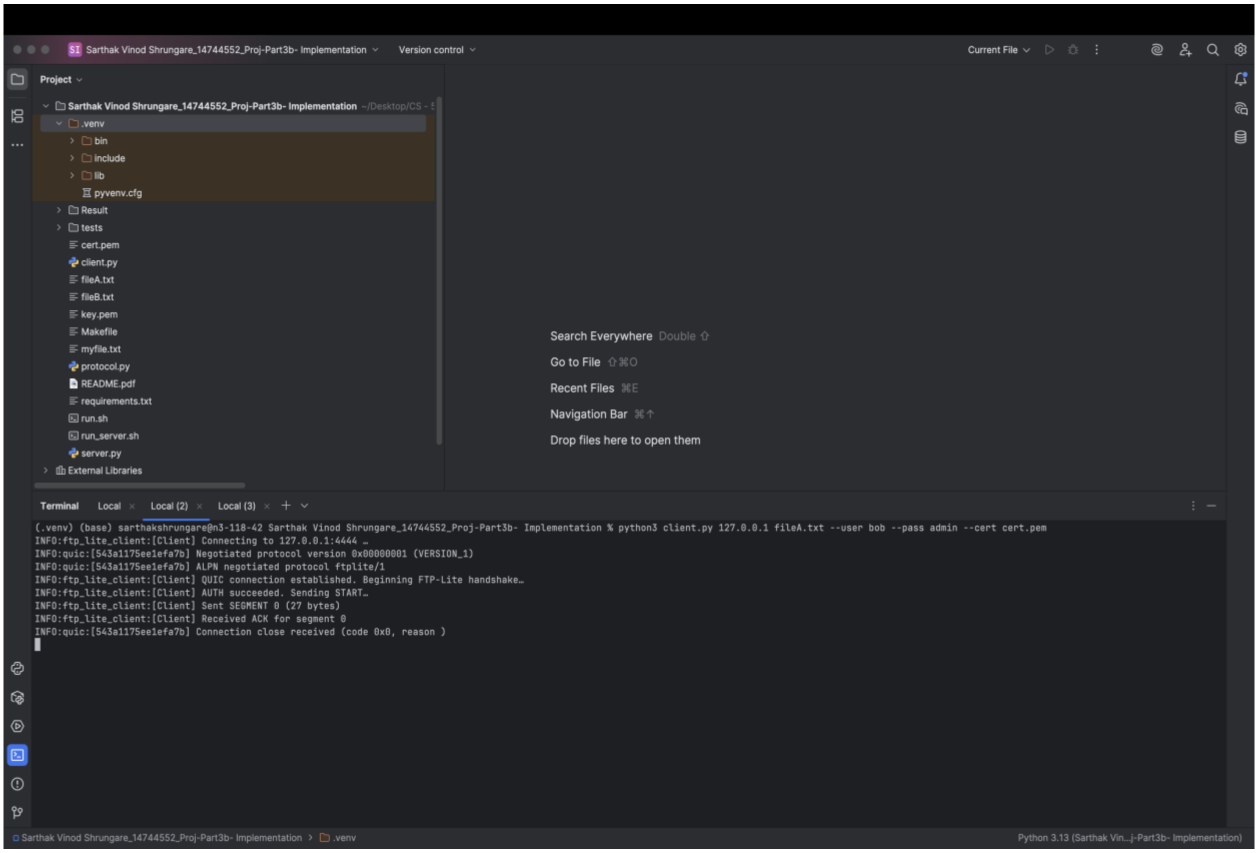
- **Terminal 1** (Server):

  ./run_server.sh 0.0.0.0 4444 cert.pem key.pem



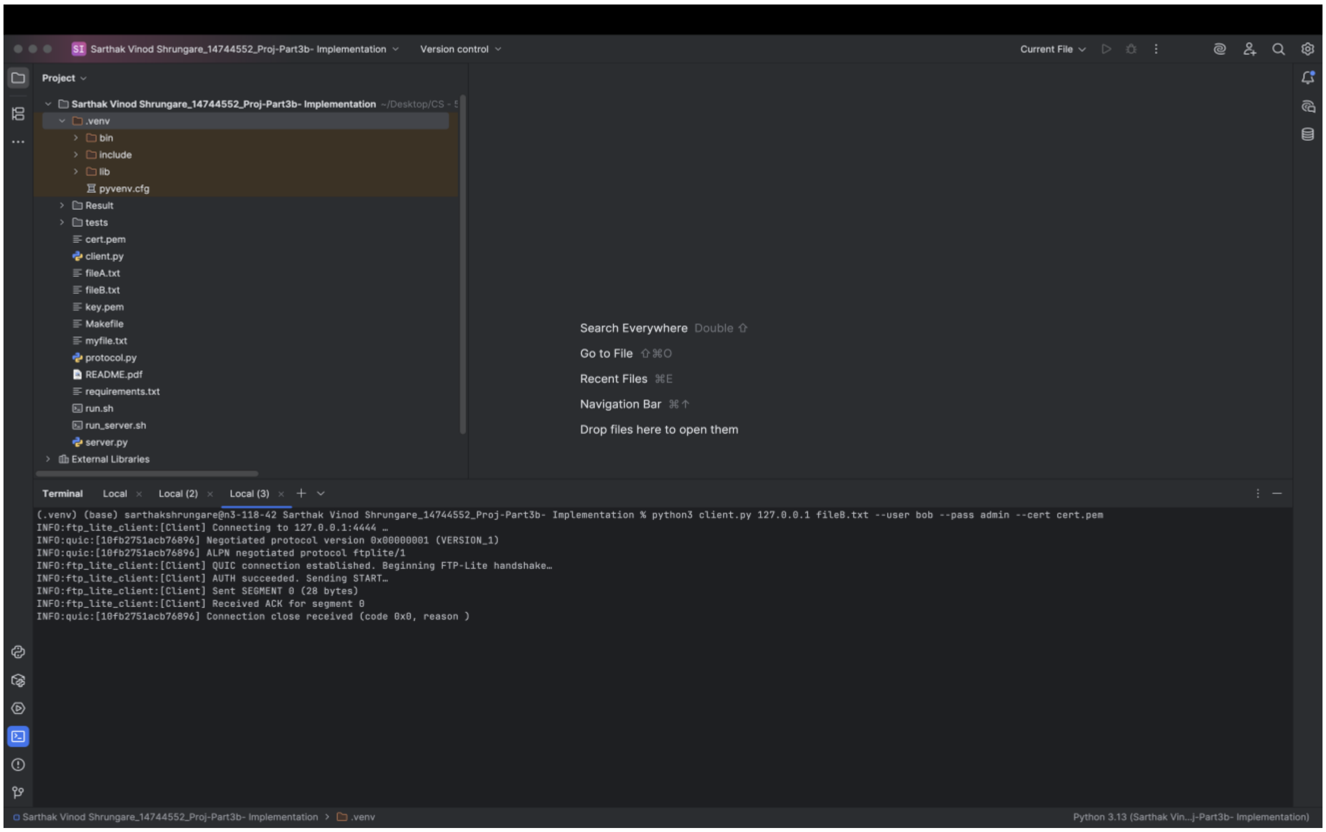
- **Terminal 2** (Client 1):

  python3 client.py 127.0.0.1 fileA.txt --user bob --pass admin --cert cert.pem



- **Terminal 3** (Client 2):

  python3 client.py 127.0.0.1 fileB.txt --user bob --pass admin --cert cert.pem



TP-Lite supports simultaneous uploads from multiple clients using Python's asyncio and QUIC.

This is tested by launching the server in one terminal and running two clients in parallel from separate terminals:

- **Client 1:** Uploads fileA.txt
- **Client 2:** Uploads fileB.txt

Both clients authenticate, initiate transfer, and send file chunks concurrently. The server maintains independent stateful sessions, processes PDUs asynchronously, and safely stores each file in the Result/ directory.

This demonstrates:

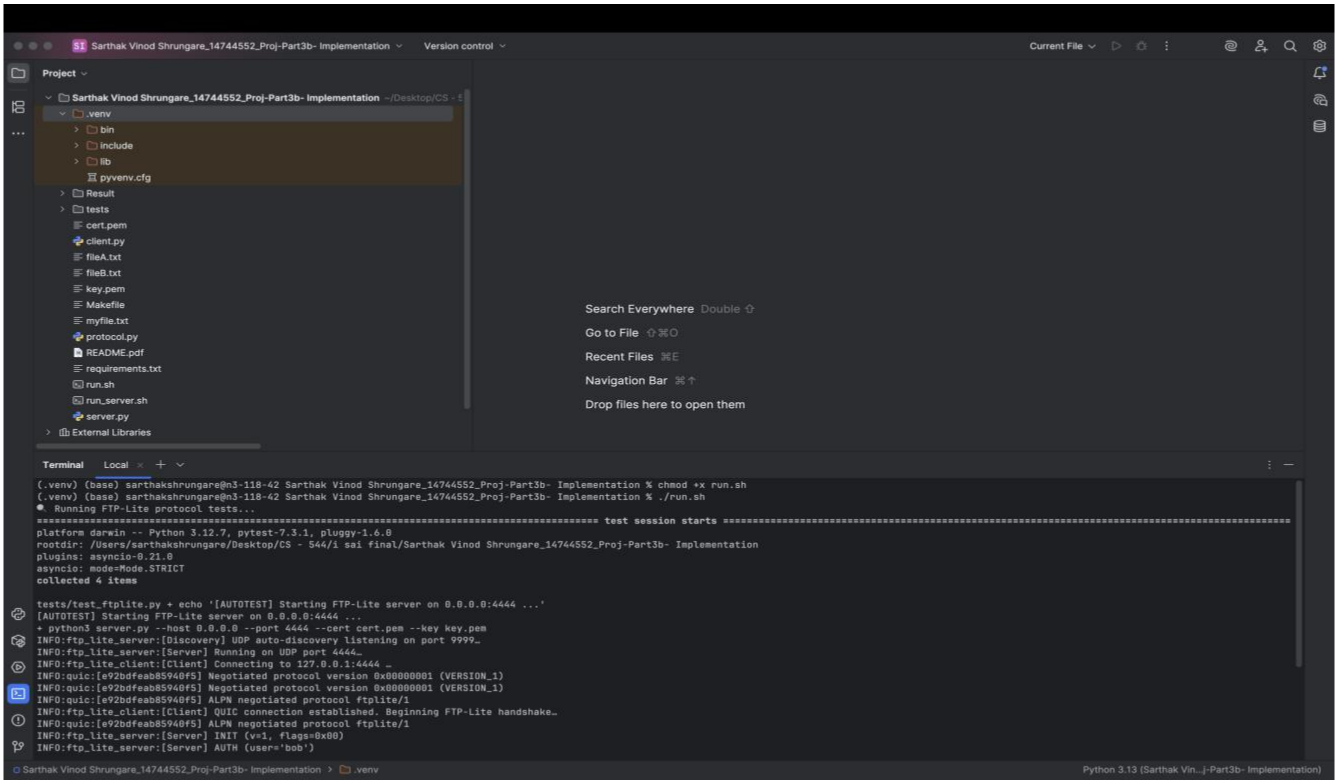
- True asynchronous concurrency using aioquic
- Accurate and reliable file transfer without blocking
- Protocol correctness under simultaneous load

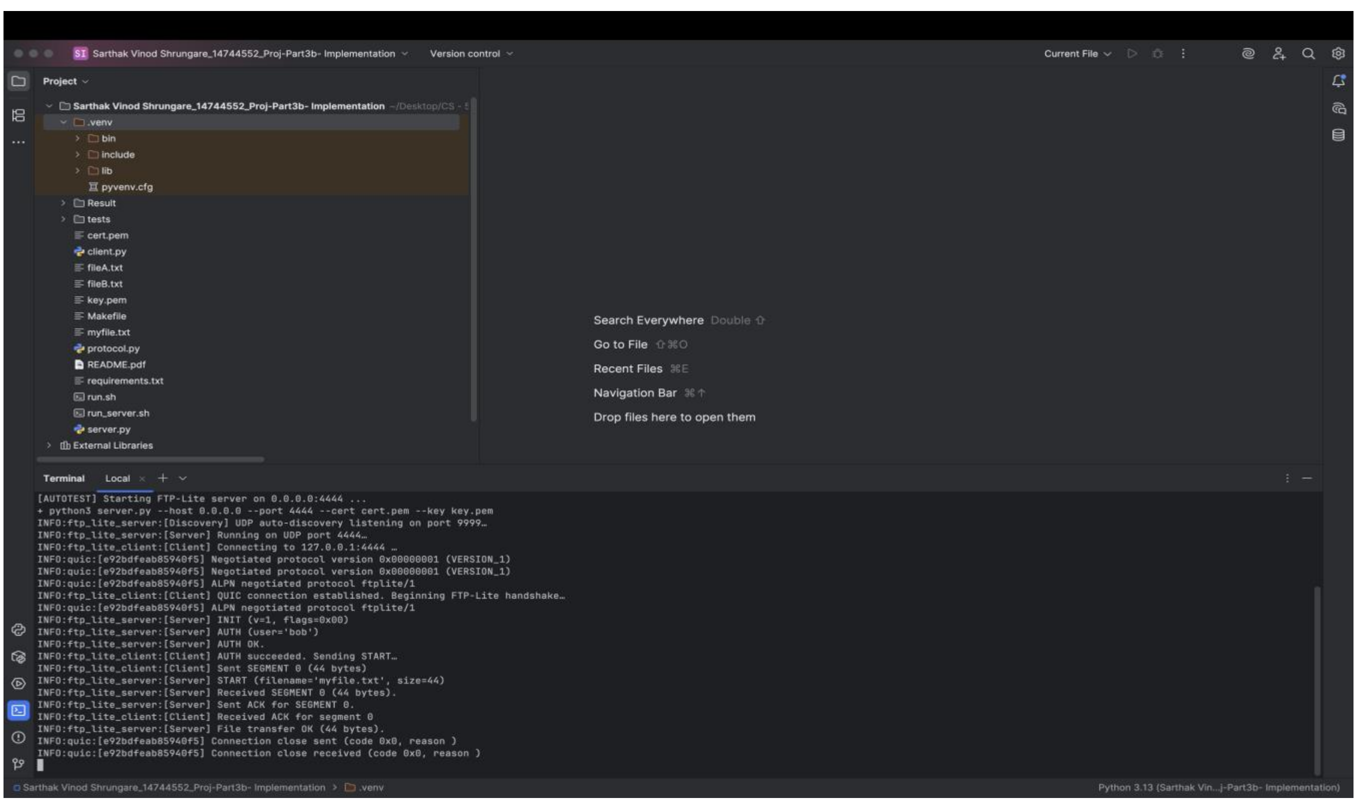
This showcases FTP-Lite's scalability, reliability, and protocol robustness in real-world concurrent upload scenarios.

Automated Testing

Run:

- chmod +x run.sh
- ./run.sh





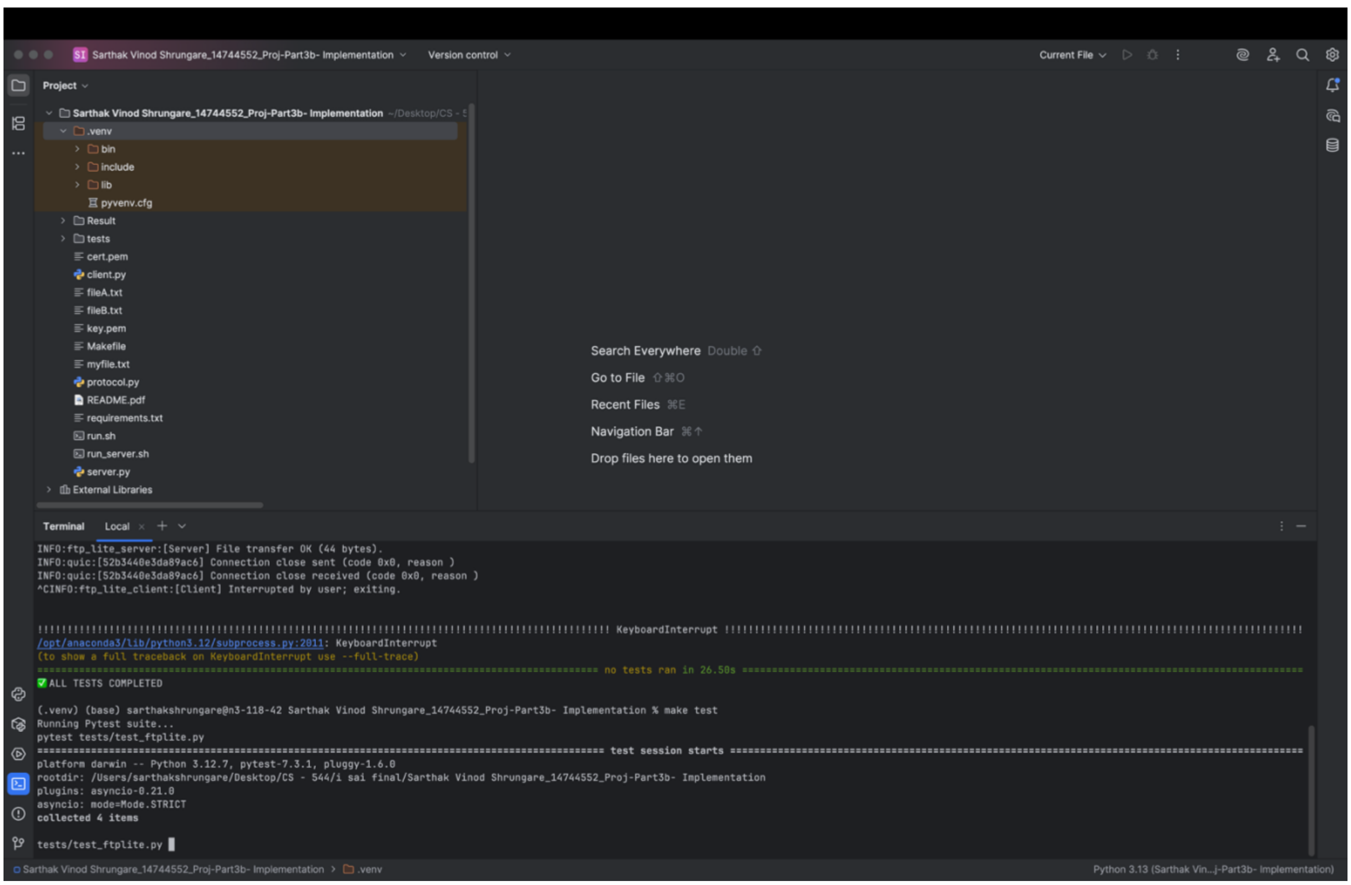
Above Automated Testing Includes tests for:

- Handshake + file upload.
- Authentication failure.
- Auto discovery.
- Fuzzing with invalid packets.

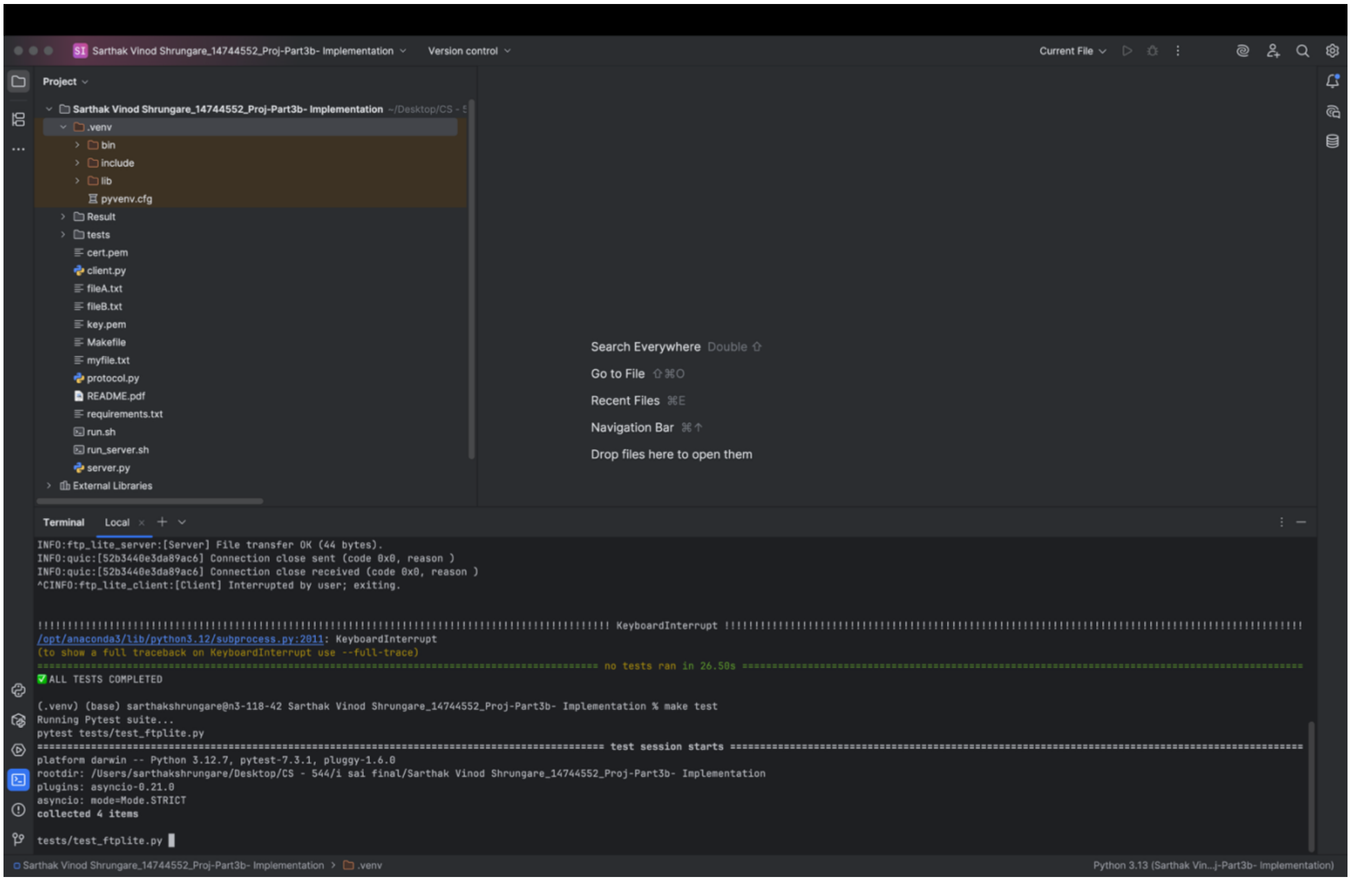
Makefile Support

To build or clean:

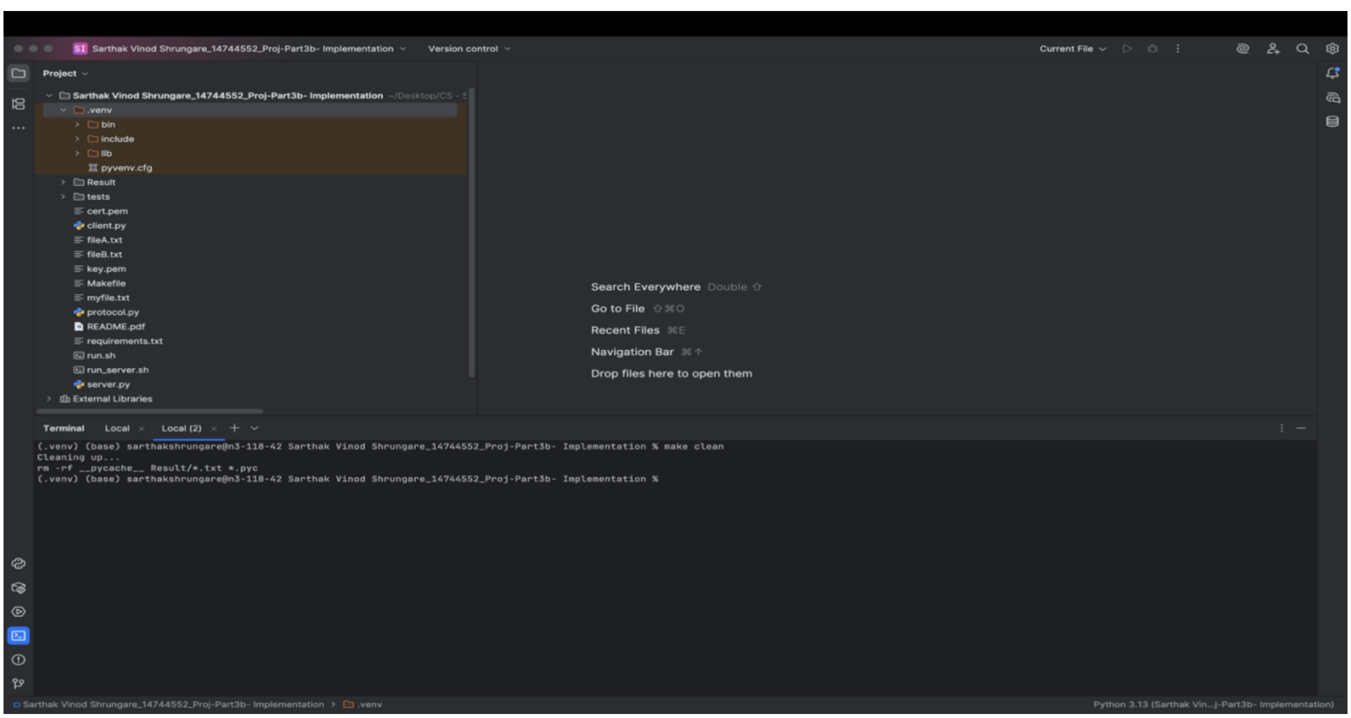
- make # Run server



- make test # Run test suite



- make clean # Clean up *.pyc or temp files



File Transfer Workflow: Client-to-Server Interaction

FTP-Lite follows a structured, stateful sequence to securely and reliably transfer files from a client to a server using QUIC and TLS. The following steps summarize the end-to-end workflow:

1. Connection Initialization:

- The client initiates a secure QUIC connection to the server (host:port, default 127.0.0.1:4444) using the provided TLS certificate (cert.pem).
- If the auto mode is used, the client broadcasts on UDP port 9999 to discover the server dynamically.

2. INIT State:

- The client sends an INIT PDU indicating protocol version support.
- The server validates the version and responds with an acknowledgment, transitioning to the authentication phase.

3. AUTH State:

- The client submits credentials using an AUTH PDU (e.g., username: bob, password: admin).
- The server validates the credentials:

  a) On success, it sends AUTH-OK.

  b) On failure, it responds with AUTH-ERR and closes the connection.

4. START State:

- The client sends a START PDU containing the file name and file size.
- The server prepares to store the incoming file under the Result/ directory, preserving the original filename.

5. SEGMENT State:

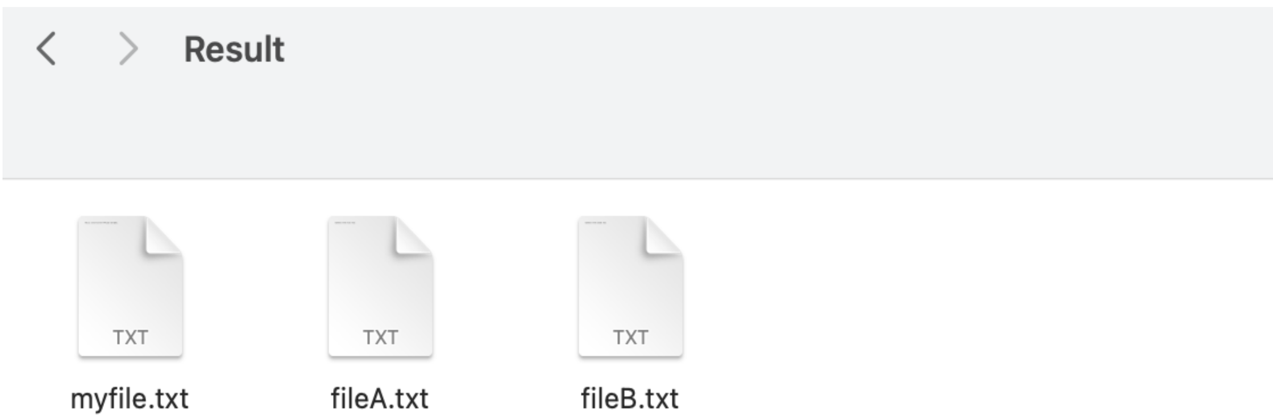
- The file is sent in binary chunks encapsulated in SEGMENT PDUs.
- Each segment includes a sequence number and payload.
- After receiving each segment, the server sends an ACK to confirm successful reception.
- This mechanism ensures ordered, reliable delivery and supports retransmission in case of failure.

6. COMPLETE State:

- Once all chunks are sent, the client issues a COMPLETE PDU.
- The server finalizes the file, closes the session, and logs a successful upload.

Result:

- All uploaded files are saved under the Result/ directory.
- The server maintains the original filenames (e.g., myfile.txt, fileA.txt, fileB.txt) and ensures data integrity.
- Concurrent uploads from multiple clients are supported via asyncio.



Feedback-Driven Design Evolution

During implementation, several changes were made to improve on the original design:

- Simplified PDU layout for easier parsing.
- Added file chunking and ACKs to improve large file reliability.
- Introduced auto discovery for user-friendly startup.
- Used asyncio and aioquic to enable real-time concurrent uploads.

This feedback loop highlights how real-world constraints and testing shaped protocol decision.

Extra Credit Features Implemented

- **Concurrent Server:** Handles multiple uploads concurrently using asyncio
- **Automated Test Suite:** Located in tests/test_ftplite.py, includes handshake, auth, auto-discovery, fuzzing.
- **Dynamic Server Discovery:** Via UDP broadcast on port 9999 (client auto mode).
- **Extensibility:** Version field in PDUs supports upgrades.
- **Code Quality:** Modular Python code, commented functions, Makefile for build/test automation.
- **Feedback & Refinement:** Improved from initial specification during implementation.
- **GitHub Repository:** https://github.com/shrungaresarthak/FTP-Lite.git
- **README Documentation:** This file includes setup, usage, testing, protocol design, and extra credit coverage.

Conclusion:

FTP-Lite developed from a design specification to a working and testable system capable of simulating many of the complications of real-world protocol stacks. It also includes modern transport technologies (QUIC and TLS) combined with structured and consistent application-layer design principles such as deterministic finite states automata (DFA), application-layer PDU structure, and stateful sessions. It serves to illustrate both practical experience in protocol design and emphasize robustness through capabilities such as asyncio for concurrent client handling, UDP broadcast server/client discovery, and pytest for automated protocol conformance testing. It serves as a practical prototype of how lightweight file transfer systems for the modern world may be designed and implemented using contemporary architectures that are secure and extensible.