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. **Generating TLS Certificates:**
- openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes

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:

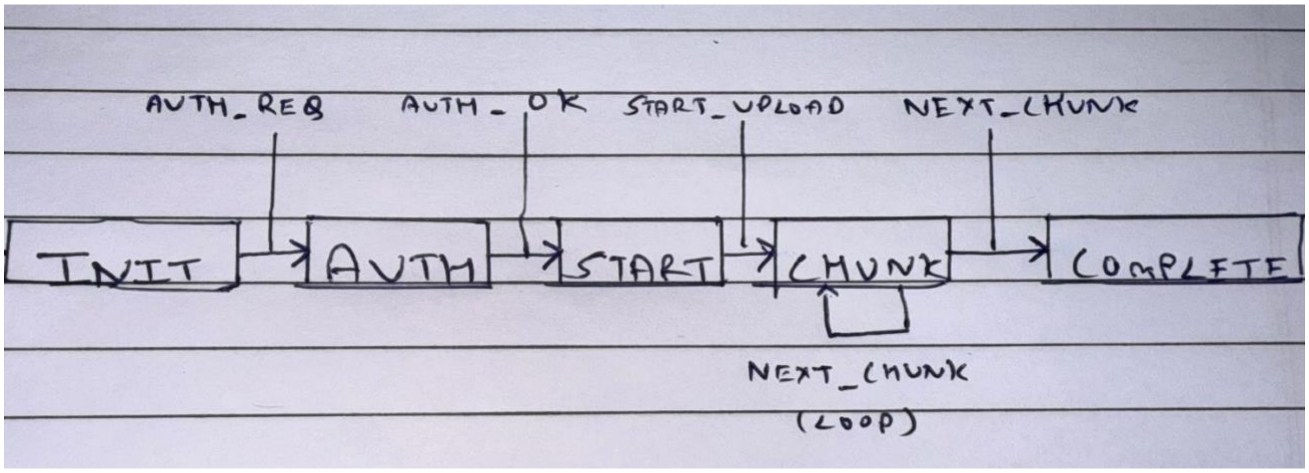
- FTP-Lite is a reliable client-server protocol that allows users to upload files to a server. It uses QUIC for transport and supports authentication, concurrency, and security using TLS.

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.

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

Usage Instructions

1. **Starting the Server:**

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

2. **Running the Client:**

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

3. **Interactive Mode:**

- python3 client.py 127.0.0.1 myfile.txt --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

Automated Testing

Run:

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

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 Storage Path

All uploaded files are saved in the Result/ directory on the server. The server preserves both the original filename and its contents as sent by the client (e.g., myfile.txt, fileA.txt, fileB.txt, etc.).

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 decisions.

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, fuzzingDynamic 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 spec 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.