A close-up of a logo

Description automatically generated

Lebanese American University

School of Arts and Sciences

Department of Computer Science and Mathematics

**CSC 430**

**Computer Networks**

**Advanced File Sharing System**

By

**Adnan Kabbani**

(202202088)

A REPORT submitted to DR. **Ayman Tajeddine** in fulfillment of the requirements for the course “CSC430: Computer Networks” in Computer Science

**4/27/2025**

*This page was intentionally left blank.*

***Table of Contents:***

I. Introduction.................................................................................................................................................4

II Project Overview...................................................................................................................................5

III. SystemRequirements.............................................................................................................................8

IV. Project Scope.......................................................................................................................................11

V. System Architecture.............................................................................................................................13

VI. Project Scope.......................................................................................................................................16

V. System Architecture .................................................................................................................. 5

VI. Technical Decisions ................................................................................................................. 8

VII. Implementation Details ........................................................................................................ 12

VIII. Server Implementation ........................................................................................................ 14

IX. Web Server Implementation .................................................................................................. 16

X. Client Implementation ............................................................................................................. 17

XI. Security Implementation ........................................................................................................ 18

XII. Error Handling and Recovery ............................................................................................... 19

XIII. Technical Challenges and Solutions ................................................................................... 20

XIV. Testing and Validation ........................................................................................................ 23

XV. Conclusion ............................................................................................................................ 26

XVI. References ........................................................................................................................... 28

**Introduction:**

**Project Overview**

The Advanced File Sharing System is a secure, multithreaded client-server application designed to facilitate file transfers in a distributed environment. The system provides a robust platform for uploading, downloading, and listing files with built-in security features, integrity verification, and resumable transfers.

The project implements both a command-line interface (CLI) and a web interface, catering to different user preferences and use cases. The system is designed with security in mind, requiring user authentication and implementing role-based access control.

**System Requirements:**

The system was designed to meet the following requirements:

* Functional Requirements
* File upload and download capabilities
* User authentication and authorization
* File listing and management
* Resumable transfers for interrupted operations
* Web and CLI interfaces
* Admin functionality for user management
* Admin functionality for user management
* Non-Functional Requirements
* Non-Functional Requirements
* Security (Authentication, authorization, file integrity)
* Reliability (Error handling, resumable transfers)
* Performance (Multithreading, efficient file handling)
* Usability (Intuitive interfaces, progress tracking)

**Project Scope:**

* Implementation of a TCP socket-based server
* Development of a command-line client
* Creation of a web interface using Flask
* User authentication and management system
* File integrity verification using SHA-256
* Resumable upload and download functionality
* Comprehensive logging and error handling

**System Architecture:**

**Client-Server Model:**

The system follows a traditional client-server architecture where:

Server: Handles client connections, authenticates users, processes file operations, and maintains the shared file repository.

Clients: Connect to the server, authenticate, and perform file operations (upload, download, list).

The server component is further divided into two main parts:  
1. Socket Server: Handles direct TCP connections and file transfers  
2. Web Server: Provides a user-friendly web interface

A screenshot of a computer

AI-generated content may be incorrect.

**Component Diagram:**

The system consists of the following key components:

**Socket Server (server.py)**

* Listens for client connections on port 9000
* Handles authentication and file operations
* Manages multithreading for concurrent clients
* Manages multithreading for concurrent clients

**Web Server (web\_server.py)**

* Runs a Flask application on port 5000
* Provides web interface for file operations
* Manages user sessions and UI interactions
* Database Module (database.py)
* Manages user credentials and roles
* Handles authentication verification
* Provides user management functions

**Client (client.py)**

* Connects to the socket server
* Provides command-line interface
* Manages file transfers and progress display

**Utility Module (utils.py)**

* Provides common functions for logging, file hashing, etc.
* Used by both server and client components
* Protocol Module (protocol.py)
* Defines the communication protocol between client and server
* Specifies message formats and constants

**Data Flow:**

The typical data flow during file operations:

* Authentication Flow
* Client connects to server
* Client sends AUTH message with credentials
* Server verifies credentials with database
* Server responds with success/failure
* Server responds with success/failure

**Upload Flow**

* Client sends UPLOAD message with file metadata
* Server acknowledges readiness
* Client sends file data in chunks
* Server verifies file integrity
* Server stores file and confirms completion
* Download Flow
* Client sends DOWNLOAD message with filename
* Server sends file metadata
* Client confirms readiness
* Server sends file data in chunks
* Client verifies file integrity

**Resume Flow**

* Client sends RESUME message with filename and byte position
* Server verifies resumability
* Transfer continues from specified position
* Integrity verification at completion

**Technical Decisions:**

**Network Protocol Design**

For the network communication protocol, I made the following key decisions:

TCP/IP: Used TCP rather than UDP for guaranteed delivery and order of packets

Custom application protocol: Implemented a message-based protocol on top of TCP

JSON encoding: Used for message content due to readability and flexibility

Length-prefixed messages: Each message is prefixed with an 8-byte length field

Chunked file transfer: Files are transferred in 8KB chunks for efficiency

The protocol was designed to be:  
- Extensible: New commands can be added without breaking compatibility  
- Self-describing: Messages contain enough information to be processed independently  
- Efficient: Minimal overhead while providing all necessary functionality

**Authentication Mechanism**

The authentication system was implemented with the following considerations:

Password hashing: SHA-256 hashing to avoid storing plaintext passwords

Session-based: Once authenticated, clients maintain a session

Role-based: Users have specific roles (admin or regular user)

Database-backed: User credentials stored in SQLite database

For the web interface, Flask-Login was used to provide secure session management and authentication.

**File Integrity Verification**

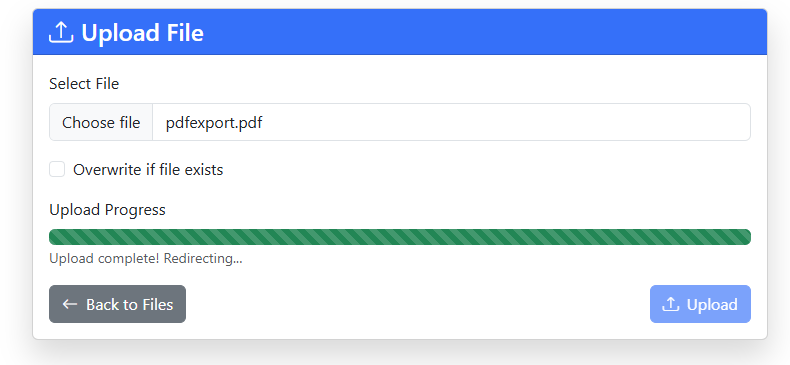
To ensure file integrity during transfers, I implemented:

SHA-256 hashing: Cryptographically secure hash function

End-to-end verification: Hash calculated at source and verified at destination

Automatic rejection: Files that fail hash verification are rejected

This approach ensures that files aren't corrupted during transfer and prevents potential tampering.



**Concurrency Model**

I chose a threading-based concurrency model:

One thread per client: Each client connection is handled in a dedicated thread

Thread synchronization: Locks used to protect shared resources

Resource tracking: Active clients and incomplete uploads tracked in thread-safe collections

This approach allows the server to handle multiple clients simultaneously while maintaining simplicity in the code structure.

**Web Interface**

For the web interface, I chose:

Flask: Lightweight web framework with minimal overhead

Flask-Login: For session management and authentication

Bootstrap: For responsive, modern UI design

AJAX: For real-time progress updates during file transfers

Flask was selected for its simplicity, flexibility, and seamless integration with Python.A screenshot of a computer

AI-generated content may be incorrect.

**Storage Design**

The storage architecture was designed with the following considerations:

File system-based: Files stored directly on the file system for simplicity and performance

Hierarchical structure: Separate directories for shared files and temporary storage

Versioning: Automatic versioning for duplicate filenames

**Implementation Details:**

**Core Modules**

**protocol.py:**

This module defines the constants and helper functions for the communication protocol:

HEADER\_SIZE (8 bytes): Allows representation of messages up to 2^64 bytes

CHUNK\_SIZE (8192 bytes): Balances memory usage and network efficiency

ENCODING (utf-8): Standard encoding for network communication

The module also contains functions for sending and receiving messages following the protocol format.

**utils.py:**

The utils module provides shared functionality across components:

* setup\_logger: Configures logging to both file and console
* calculate\_file\_hash: Computes SHA-256 hash for file integrity
* handle\_duplicate\_filename: Manages file versioning
* get\_file\_size: Returns file size in bytes
* chunk\_file: Generator for reading files in chunks

**database.py:**

The database module manages user authentication and storage:

* Database class: Manages SQLite database for user credentials
* User storage: Usernames, hashed passwords, and roles
* Authentication: Verifies credentials against stored values
* User management: Functions to add, delete, and modify users
* SQLite was chosen for simplicity while providing all necessary functionality.

**Server Implementation**

**Socket Server (server.py):**

The socket server handles direct TCP connections from clients:

* ServerSocket: Listens on port 9000 for incoming connections
* ClientHandler class: Threaded handler for each client connection
* Command processing: Methods for handling various client commands
* File operations: Upload, download, and listing functionality
* Authentication integration: Validates users via database module

The server uses multithreading to handle concurrent clients, with each client connection managed in a separate thread.

A screenshot of a computer

AI-generated content may be incorrect.

**Web Server (web\_server.py):**

The web server provides a user-friendly interface:

* Flask application: Routes for various user interactions
* User interface: Renders templates for file management
* Authentication: Login system using Flask-Login
* File operations: Upload, download, delete via web forms
* User management: Admin-only functions for user administration
* Progress tracking: AJAX-based progress updates

**Client Implementation:**

The client implementation (client.py) provides:

* Command-line interface: Arguments for different operations
* Socket communication: Connects to the server via TCP
* File transfer logic: Uploads, downloads, and resumption
* Progress display: Real-time progress bars (using tqdm)
* Error handling: User-friendly error messages

**Security Implementation:**

Security features were implemented at multiple levels:

* Authentication: Username/password verification
* Authorization: Role-based access control
* Data integrity: SHA-256 hash verification
* Secure filenames: Prevention of path traversal attacks
* Input validation: Sanitization of user inputs
* Session management: Secure handling of web sessions

**Error Handling and Recovery:**

The system implements robust error handling:

* Comprehensive logging: All actions and errors logged
* Graceful degradation: System continues functioning despite errors
* User feedback: Clear error messages for users
* Recovery mechanisms: Resume functionality for interrupted transfers

**Technical Challenges and Solutions**

**Handling Interrupted Transfers:**

Challenge: Network interruptions can cause incomplete file transfers, leading to data loss or corruption.

Solution:   
- Implemented resume functionality for both uploads and downloads  
- Server tracks partial uploads with byte positions  
- Clients can request to resume from the last successfully transferred byte  
- Temporary files used for in-progress transfers

**Ensuring File Integrity**

Challenge: Files might be corrupted during transfer or tampered with.

Solution:  
- SHA-256 hash calculation before and after transfer  
- Hash comparison to verify file integrity  
- Automatic rejection of files failing the integrity check  
- Logging of integrity failures for security auditing

**Concurrent Client Connections**

Challenge: Managing multiple client connections simultaneously without resource conflicts.

Solution:  
- Thread-based concurrency model  
- Thread-safe collections with proper locking  
- Separate connection handling for each client  
- Resource cleanup on client disconnection

**Security Concerns**

Challenge: Protecting against unauthorized access and ensuring data security.

Solution:  
- Authentication system with password hashing  
- Role-based access control  
- Secure session management  
- Input validation and sanitization  
- Protection against common web vulnerabilities in the Flask app

A screenshot of a computer

AI-generated content may be incorrect.

**Testing and Validation**

**Testing Methodology:**

The system was tested using:

* System testing: End-to-end functionality
* Edge case testing: Handling of unusual conditions

**Performance Testing:**

Performance was evaluated by:

* Load testing: Multiple simultaneous clients
* Large file testing: Transfer of files up to several GB
* Latency testing: Performance over network with varied latency

**Security Testing:**

Security was validated through:

* Authentication testing: Attempts with invalid credentials
* Authorization testing: Access control enforcement
* File integrity testing: Deliberate corruption of files
* Input validation testing: Malformed input handling

**Conclusion**

The Advanced File Sharing System successfully implements:

* Secure file sharing between multiple clients
* Web and command-line interfaces
* Authentication and role-based access
* File integrity verification
* Resumable transfers for reliability

The Advanced File Sharing System fulfills its core objective of providing a secure, reliable, and user-friendly platform for distributed file transfers. By combining a multithreaded TCP socket server with a modern Flask-based web interface, it offers both command-line and web clients that support resumable uploads/downloads, strong authentication, role-based access control, and end-to-end integrity verification. Comprehensive logging and robust error-handling ensure that users receive clear feedback and that interrupted operations can resume gracefully, minimizing data loss and downtime. Despite these successes, the system faces inherent scalability and security trade-offs. Python’s Global Interpreter Lock (GIL) limits raw throughput under heavy concurrency, and the lack of end-to-end encryption leaves file contents potentially exposed if the server is compromised. The current design also assumes a single centralized server, which may become a bottleneck or single point of failure as the number of clients grows.

**References:**

Python Documentation: https://docs.python.org/3/

Socket Programming in Python: https://realpython.com/python-sockets/

Flask Documentation: https://flask.palletsprojects.com/

Flask-Login Documentation: https://flask-login.readthedocs.io/

SHA-256 Algorithm: https://en.wikipedia.org/wiki/SHA-2

TCP/IP Protocol: https://en.wikipedia.org/wiki/Transmission\_Control\_Protocol

Werkzeug Documentation: https://werkzeug.palletsprojects.com/

Bootstrap Documentation: https://getbootstrap.com/docs/

Python Threading: https://docs.python.org/3/library/threading.html

SQLite Documentation: https://www.sqlite.org/docs.html