FTP

This project contains a simple FTP server and client implementation in C and Flutter.

Server

The server is implemented in C and can be found in the `server` directory.

Client

The client is implemented in Flutter and can be found in the `client` directory.

Usage

Server

A prebuilt binary is in `server/linux_amd64` and `server/linux_arm64`. Or you can build it yourself with `make all` in the `server/src` directory. For more details, refer to the `README.md` in the `server` directory.

Client

Prebuilt binaries for Linux can be found in `client/linux_amd64` and also `client/linux_arm64`. For more instructions to build the client, refer to the `README.md` in the `client` directory.

Implementation Details

Server

The following commands are implemented:

- USER (Handle user login)
- PASS (Handle password authentication)
- QUIT (Handle client disconnection)
- RETR (Retrieve a file)
- STOR (Store a file)
- PORT (Set up active mode data connection)
- PASV (Enter passive mode)
- TYPE (Set transfer type)
- LIST (List directory contents)
- MKD (Create a directory)
- CWD (Change working directory)
- PWD (Print working directory)
- RMD (Remove a directory)
- SYST (Get system type)
- ABOR (Abort current operation)
- EPSV (Enter extended passive mode)
- DELE (Delete a file)
- SIZE (Get file size)

Client

The client is a cross-platform FTP client built with Flutter, supporting multiple platforms (Windows, macOS, Linux, iOS, Android). Key features include:

- Cross-platform support (Windows, macOS, Linux, iOS, Android)
- User-friendly interface for FTP operations
- File upload and download capabilities
- Drag and drop support for file transfers

Note that web is not supported due to the limitations of raw TCP sockets in web environments.

Security Considerations:

The server implementation is basic and intended for educational purposes.

This highlights the need for additional security features in a production environment.

Valuable insights and potential areas for improvement

1. Security Enhancements: Implementing proper authentication, encryption, and access control mechanisms would be crucial for a production-ready FTP server and client.
2. Extended Protocol Support: Consider implementing more advanced FTP features or supporting FTPS (FTP over SSL/TLS) for secure file transfers.
3. User Interface Improvements: For the Flutter client, focus on creating an intuitive and user-friendly interface for file management and transfer operations.
4. Performance Optimization: Analyze and optimize the performance of both the server and client, especially for large file transfers or high-concurrency scenarios.
5. Error Handling and Logging: Implement robust error handling and logging mechanisms to facilitate debugging and improve user experience.
6. Expanded Test Coverage: Develop a more comprehensive test suite for both the server and client to ensure reliability across different scenarios and platforms.
7. Documentation: Enhance the documentation for both the server and client, including API references, usage examples, and deployment guidelines.
8. Containerization: Consider containerizing the server application for easier deployment and scalability.

By addressing these areas, the project could evolve from an educational tool to a more robust and feature-rich FTP solution suitable for real-world applications.