Post-Quantum Blockchain File Sharing System Documentation

Executive Summary

The Post-Quantum Blockchain File Sharing System is a secure client-server application designed to demonstrate the integration of post-quantum cryptography with blockchain technology for secure file sharing. The system uses CRYSTALS-Kyber for key exchange and CRYSTALS-Dilithium for digital signatures to achieve quantum-resistant security. Built using Java with Ant, the system implements multi-factor authentication using TOTP, establishes secure sessions, encrypts files with AES-256-GCM, and maintains an immutable blockchain ledger of all file transactions.

System Architecture

The system is divided into three main modules with clean separation of responsibilities:

1. **Client-Side Operations**: Handles user interaction and file management
2. **Server Infrastructure**: Manages authentication and file storage
3. **Blockchain Components**: Maintains the transaction ledger

Directory Structure

src/

├── client/

│ ├── Client.java

│ ├── ClientOptions.java

│ ├── NetworkManager.java

│ ├── FileOperations.java

│ └── CryptoManager.java

├── server/

│ ├── FileServer.java

```
│   ├── ClientHandler.java
│   ├── AuthManager.java
│   ├── FileManager.java
│   └── CryptoManager.java
├── blockchain/
│   ├── BlockchainManager.java
│   ├── Block.java
│   ├── Transaction.java
│   └── FileMetadata.java
├── common/
│   ├── Message.java
│   ├── User.java
│   ├── Config.java
│   └── Constants.java
└── pqcrypto/
    ├── KyberOperations.java
    ├── DilithiumOperations.java
    ├── TOTPManager.java
    ├── SymmetricCrypto.java
    └── CertificateManager.java
```

Key Features

- **Post-Quantum Cryptography**: Uses CRYSTALS-Kyber and CRYSTALS-Dilithium algorithms

- **Multi-Factor Authentication**: TOTP-based second factor using FreeOTP or Google Authenticator
- **Blockchain Transaction Ledger**: Immutable record of all file operations]
- **Post Quantum Key Exchange:** Key distribution takes place using CRYSTALS Kyber symmetric key encapsulation
- **Secure File Encryption**: AES-256-GCM for file content protection
- **Digital Signatures**: Each transaction is signed using Dilithium quantum-resistant signatures\

Security Requirements:

Man-in-the-Middle Attack Prevention

The system prevents man-in-the-middle attacks through several complementary mechanisms:

1. **Post-Quantum Key Exchange**:
    - Utilizes CRYSTALS-Kyber for all key exchange operations
    - Key encapsulation mechanism (KEM) generates strong shared secrets
    - Implementation: `KyberOperations.java` handles encapsulation/decapsulation
2. **Client-Server Authentication**:
    - Server validates client identity through multi-factor authentication
    - Client validates server through certificate verification
    - Implementation: `CertificateManager.java` verifies certificate chain integrity
3. **Message Authentication**:
    - All protocol messages include digital signatures
    - Signatures cover both message content and headers
    - Implementation: `Message.java` includes signature verification methods
4. **Connection Parameters Binding**:
    - Session establishment incorporates session-specific parameters
    - Protocol includes explicit authentication of key exchange
    - Implementation: `AuthManager.java` binds session context to key material

Replay Attack Prevention

The system prevents replay attacks through a comprehensive approach:

1. **Cryptographic Nonces**:
    - Every message includes a unique cryptographic nonce
    - Nonces are verified on receipt using NonceCache
    - Implementation: `NonceCache.java` maintains recently seen nonces
2. **Timestamping**:
    - All messages include timestamps validated against time windows

    - Server rejects messages outside the valid time window
    - Implementation: `Message.java` includes timestamp validation
3. **Message Sequencing**:
    - Protocol state machine enforces strict message ordering
    - Out-of-sequence messages are rejected
    - Implementation: `ClientHandler.java` enforces protocol state transitions
4. **Session-Specific Nonces**:
    - Each new session generates unique nonce seeds
    - Nonces are bound to specific sessions
    - Implementation: `AuthManager.java` manages session-specific nonce contexts

Ensuring Authenticity

The system ensures authenticity at multiple levels:

1. **Digital Signatures**:
    - All transactions signed with CRYSTALS-Dilithium
    - Every file upload creates signed blockchain transaction
    - Implementation: `DilithiumOperations.java` provides quantum-resistant signatures
2. **Multi-Factor Authentication**:
    - Password plus TOTP-based authentication
    - TOTP implementation follows RFC 6238
    - Implementation: `TOTPManager.java` handles TOTP generation/validation
3. **Blockchain Verification**:
    - Immutable record of all transactions maintains file history
    - Each block links cryptographically to previous blocks
    - Implementation: `BlockchainManager.java` verifies transaction chains
4. **File Integrity**:
    - SHA-3 hashing of all file content
    - Hash values stored in blockchain for verification
    - Implementation: `FileOperations.java` performs integrity checks

Ensuring Confidentiality

The system ensures confidentiality through multiple security layers:

1. **End-to-End Encryption**:
    - All files encrypted with AES-256-GCM before transmission
    - Symmetric keys encapsulated with Kyber
    - Implementation: `SymmetricCrypto.java` provides AEAD encryption
2. **Secure Channel**:
    - All communications encrypted with session keys

    - Forward secrecy through ephemeral Kyber keys
    - Implementation: `NetworkManager.java` encrypts all channel communications
3. **Key Protection**:
    - User private keys never transmitted
    - Kyber encapsulation protects file-specific symmetric keys
    - Implementation: `CryptoManager.java` handles key lifecycle
4. **Blockchain Privacy**:
    - Blockchain stores only encrypted symmetric keys
    - Metadata minimization in transactions
    - Implementation: `FileMetadata.java` contains only necessary encrypted data

Protocol Flows

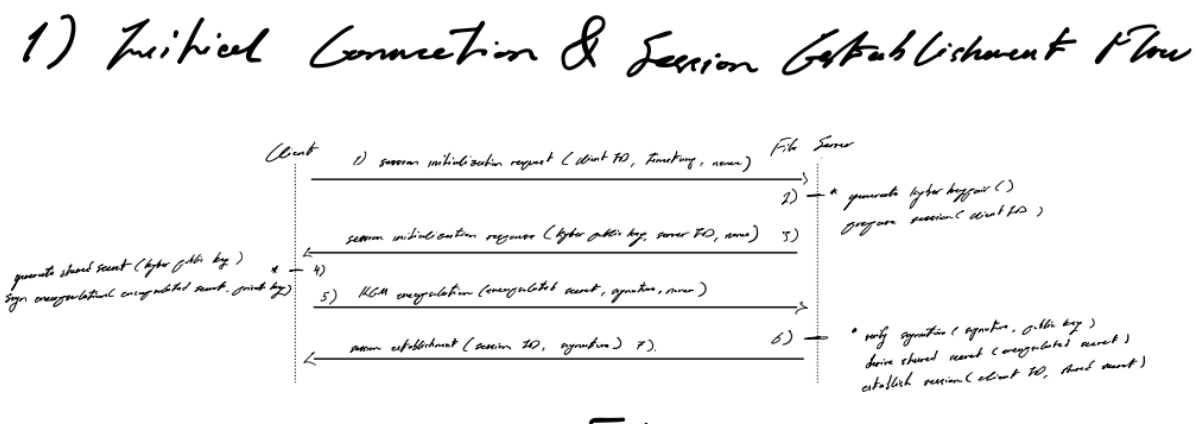
Flow 1: Session Establishment

Client initiates a secure connection with the server using Kyber key exchange:

1. Client generates Kyber keypair and sends public key to server
2. Server generates session key, encapsulates it with client's public key
3. Client decapsulates to retrieve the session key
4. Both parties now have a shared secret for encryption

Implementation Classes:

- Client: `client.NetworkManager`, `client.CryptoManager`
- Server: `pqcrypto.ClientHandler`, `pqcrypto.AuthManager`
- Crypto: `pqcrypto.KyberOperations`



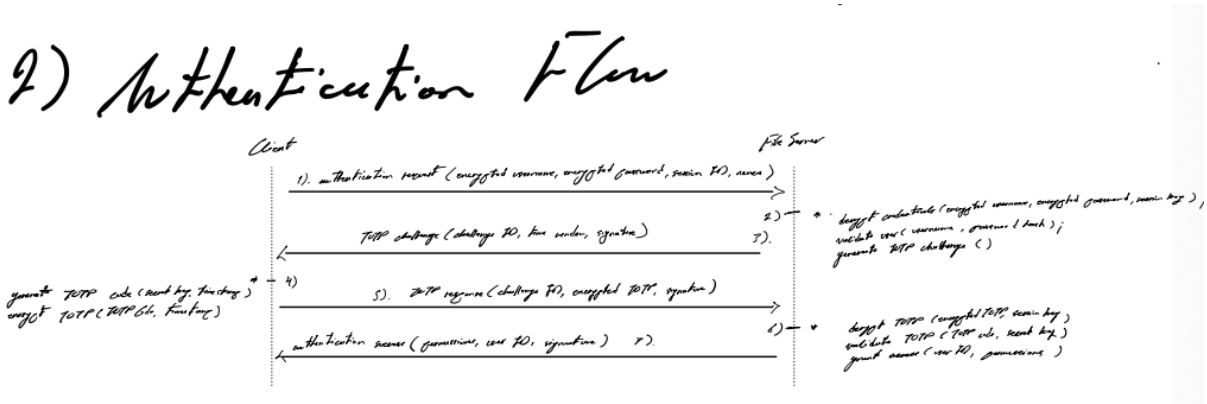
Flow 2: Authentication

Multi-factor authentication using password and TOTP:

1. Client sends username to server
2. Server challenges with nonce
3. Client responds with password hash and TOTP code
4. Server validates both factors
5. Session established with unique session ID

Implementation Classes:

- Client: `client.NetworkManager`, `client.CryptoManager`
- Server: `pqcrypto.AuthManager`, `pqcrypto.TOTPManager`
- Data: `common.User`, `common.Message`



Flow 3: File Upload

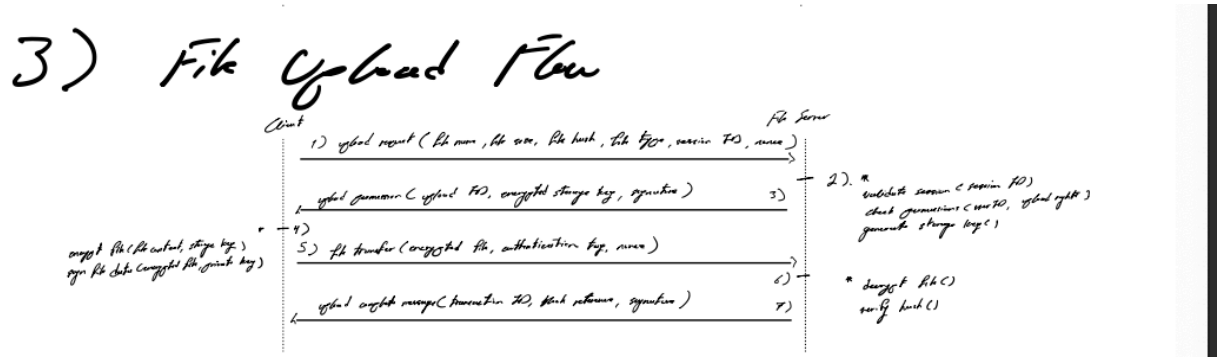
Secure file transmission and blockchain recording:

1. Client encrypts file with AES-256-GCM
2. Client sends encrypted file with metadata
3. Server verifies client signature
4. Server stores file and adds transaction to blockchain
5. Server confirms successful storage

Implementation Classes:

- Client: `client.FileOperations`, `client.NetworkManager`
- Server: `pqcrypto.FileManager`, `pqcrypto.ClientHandler`
- Blockchain: `blockchain.BlockchainManager`, `blockchain.Transaction`

- Crypto: `pqcrypto.SymmetricCrypto`



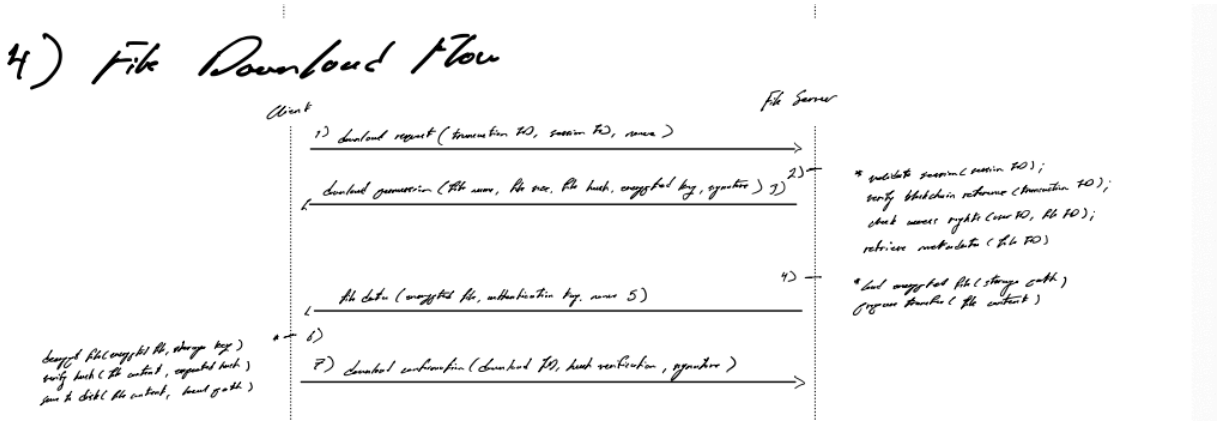
Flow 4: File Download

Secure file retrieval with blockchain verification:

1. Client requests file by hash
2. Server verifies client's access rights via blockchain
3. Server sends encrypted file
4. Client decrypts file using key from blockchain metadata
5. Client verifies file integrity with hash

Implementation Classes:

- Client: `client.FileOperations`, `client.NetworkManager`
- Server: `pqcrypto.FileManager`, `pqcrypto.ClientHandler`
- Blockchain: `blockchain.BlockchainManager`
- Crypto: `pqcrypto.SymmetricCrypto`



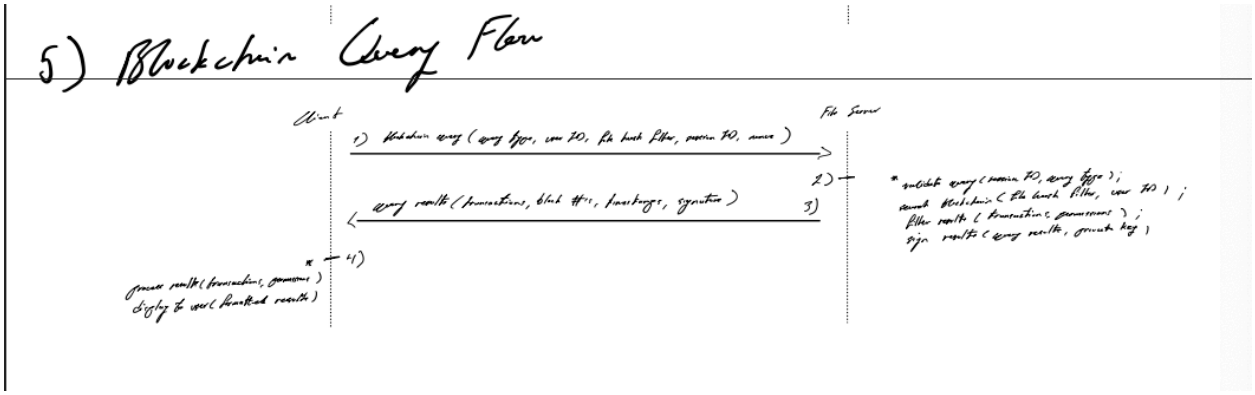
Flow 5: Blockchain Query

Access to transaction history:

1. Client requests blockchain information
2. Server authenticates request
3. Server returns relevant blockchain entries
4. Client displays transaction information

Implementation Classes:

- Client: `client.NetworkManager`
- Server: `pqcrypto.ClientHandler`
- Blockchain: `blockchain.BlockchainManager`, `blockchain.Block`, `blockchain.Transaction`



Flow 6: Connection Termination

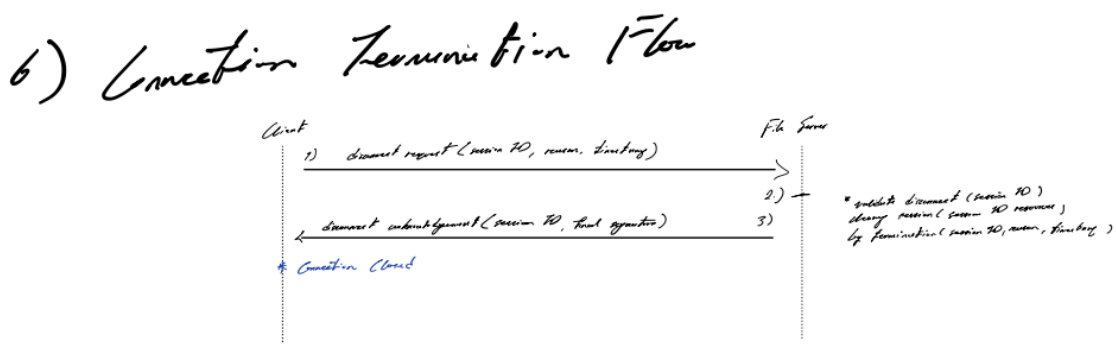
Secure session cleanup:

1. Client sends termination request
2. Server acknowledges, cleans up session data
3. Connection closed securely

Implementation Classes:

- Client: `client.NetworkManager`

- Server: `pqcrypto.ClientHandler`, `pqcrypto.AuthManager`



Class Descriptions

Client Module

Client.java

- Main application entry point
- Orchestrates client-side operations
- Manages application lifecycle
- Coordinates user actions
- Handles error recovery
- Loads configuration settings

ClientOptions.java

- Parses command-line arguments
- Defines available command flags (--upload, --download, --list, --config)
- Validates user input
- Provides usage instructions

NetworkManager.java

- Manages TCP socket connections
- Implements simplified protocol for file transfers
- Handles message serialization/deserialization
- Provides connection retry logic
- Manages secure channel establishment
- Handles protocol state transitions

CryptoManager.java (client)

- Centralizes cryptographic operations
- Manages Kyber key pairs
- Handles file encryption/decryption
- Generates/validates TOTP codes
- Creates/verifies Dilithium signatures
- Manages secure random number generation

FileOperations.java

- Handles file reading for uploads
- Manages file writing for downloads
- Performs file integrity checking
- Creates file metadata objects
- Handles file encryption/decryption

Server Module

FileServer.java

- Main server application entry point
- Binds to configured port
- Creates thread pool for client connections
- Loads server configuration
- Manages server lifecycle
- Coordinates server components

ClientHandler.java

- Manages individual client connections
- Implements protocol state machine
- Handles authentication flow
- Processes file requests
- Manages session keys
- Implements error handling

AuthManager.java

- Handles user authentication
- Manages TOTP validation
- Establishes session keys
- Tracks active sessions
- Implements security policies
- Manages authentication state

FileManager.java

- Manages physical file storage
- Handles file upload/retrieval
- Implements file access control
- Maintains file metadata
- Provides file verification

CryptoManager.java (server)

- Manages server's Dilithium key pair
- Handles Kyber key exchange
- Validates client signatures
- Manages certificate operations
- Implements nonce validation

Blockchain Module

BlockchainManager.java

- Maintains blockchain ledger
- Adds new transaction blocks
- Validates transactions
- Persists blockchain to JSON
- Provides transaction lookup
- Implements simple consensus

Block.java

- Represents a blockchain block
- Contains block header and transactions
- Implements block validation
- Provides serialization
- Calculates block hash
- Manages block linking

Transaction.java

- Represents file transactions
- Contains file metadata
- Includes digital signatures
- Provides transaction validation
- Handles transaction serialization
- Tracks transaction status

FileMetadata.java

- Stores file information
- Contains encrypted symmetric key
- Includes file hash
- Manages access control
- Provides serialization
- Tracks file ownership

Common Module

Message.java

- Defines protocol communication structure
- Implements message serialization
- Contains message types
- Includes security headers
- Provides validation methods
- Handles message encryption

User.java

- Represents user accounts
- Stores hashed password and TOTP secret
- Contains public keys
- Manages user permissions
- Provides serialization
- Handles user validation

Config.java

- Loads configuration from JSON
- Provides access to settings
- Validates configuration parameters
- Manages default values
- Handles configuration updates

Constants.java

- Defines system constants
- Contains crypto parameters
- Specifies protocol version
- Defines message types
- Includes network timeouts

PQ Crypto Module

KyberOperations.java

- Implements CRYSTALS-Kyber
- Generates key pairs
- Performs key encapsulation/decapsulation
- Manages key serialization
- Provides secure key storage
- Implements side-channel resistant operations

DilithiumOperations.java

- Implements CRYSTALS-Dilithium
- Generates signing key pairs
- Creates digital signatures
- Verifies signatures
- Handles signature serialization
- Manages key lifecycle

TOTPManager.java

- Generates TOTP codes for 2FA
- Validates TOTP with time windows
- Manages TOTP secrets
- Handles clock synchronization
- Implements HMAC-SHA256
- Provides backup code validation

SymmetricCrypto.java

- Implements AES-256-GCM
- Handles file encryption
- Performs file decryption
- Manages initialization vectors
- Provides streaming support
- Implements secure key handling

CertificateManager.java

- Acts as Certificate Authority
- Issues X.509 certificates
- Manages CA private key
- Handles certificate signing
- Maintains Certificate Revocation List
- Provides verification services

Storage Components

keystore.jks

- Securely stores private keys
- Contains server private keys
- Manages key aliases
- Provides secure key retrieval
- Implements key backup/recovery

ca.cer : these were something we decided to take out due to time constraints, they would've allowed us the ability to provide certificates to users, making it so that access to files is only possible if given a certificate from the owner, sort of like sponsorship, another idea we had was to also restrict access to entire sessions based on a sponsorship system, where the only way to even be able to see and interact with a collection of files if given sponsorship by one of the members, with a member being a person with access to one or more of the files within the database in question

- Contains root CA certificate
- Stores CA's public key
- Includes CA identity information
- Used for certificate verification
- Implements certificate chain of trust

blockchain.json

- Stores complete blockchain
- Contains blocks with transactions
- Implements file-based persistence
- Provides human-readable inspection
- Handles atomic writes

Implementation Details

Blockchain Structure

The blockchain is stored as a JSON file with the following structure:

```
{
  "blockchain": [
```

```
  {
    "index": 0,
    "timestamp": "2025-04-01T08:00:00Z",
    "previousHash":
"0000000000000000000000000000000000000000000000000000000000000000",
    "hash": "1a2b3c...",
    "transactions": []
  },
  {
    "index": 1,
    "timestamp": "2025-04-01T09:15:22Z",
    "previousHash": "1a2b3c...",
    "hash": "3e4f5g...",
    "transactions": [
      {
        "id": "tx1234567890",
        "timestamp": "2025-04-01T09:14:30Z",
        "uploader": "alice",
        "fileMetadata": {
          "fileName": "report.pdf",
          "fileSize": 1048576,
          "fileHash": "7f83b1...",
          "encryptedSymmetricKey": "A7B8C9...",
          "iv": "Z9Y8X7..."
        },
```

```
        "signature": "S1G2N3..."
      }
    ]
  }
 ]
}
```


TOTP Implementation

To implement TOTP (Time-based One-Time Password):

1. **Secret key generation**:
    - Generate a 20-byte random secret key during registration
    - Base32-encode for user display
    - Store the raw binary key (base64-encoded)
2. **TOTP calculation** (RFC 6238):
    - Get current time in seconds
    - Calculate time steps: (currentTime - T0) / 30
    - Calculate HMAC-SHA1 of time step using secret key
    - Extract 6-digit code using dynamic truncation
3. **TOTP verification**:
    - Calculate TOTP for current and adjacent time steps (±1)
    - Compare provided OTP with calculated values
    - Return true if any match

Command Line Interface

Client Commands

$ java -jar client.jar

usage:

client --register --user <username> --host <host> --port <portnum>

client --upload <filepath> --user <username> --host <host> --port <portnum>

client --download <filehash> --dest <directory> --user <username> --host <host> --port <portnum>

client --list --user <username> --host <host> --port <portnum>

client --verify <filehash> --user <username> --host <host> --port <portnum>

client --blockchain --user <username> --host <host> --port <portnum>

options:

-r, --register Register a new account

-u, --upload Upload a file to the blockchain

-d, --download Download a file from the blockchain

-l, --list List all available files

-v, --verify Verify file integrity on the blockchain

-b, --blockchain View blockchain transaction history

-usr, --user The username

-h, --host The host name of the server

-p, --port The port number for the server

-dst, --dest Destination directory for downloaded files

Server Commands

$ java -jar server.jar

usage:

server

server --config <configfile>

server --help

options:

-c, --config Set the config file

-h, --help Display the help

Usage Examples

Registering a New User

$ java -jar client.jar --register --user alice --host 127.0.0.1 --port 5001

Enter password:

Base 32 Key: jbswy3dpehpk3pxp

Private Key: [Base64-encoded private key will appear here]

Please add this key to your FreeOTP or Google Authenticator app by:

1. Opening the app

2. Clicking + to add a new account

3. Scanning this QR code or entering the base32 key manually

Uploading a File

$ java -jar client.jar --upload documents/report.pdf --user alice --host 127.0.0.1 --port 5001

Enter password:

Enter OTP: 123456

Authenticated.

Encrypting file...

Adding to blockchain...

File uploaded successfully!

File hash: 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069

Downloading a File

$ java -jar client.jar --download 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069 --dest downloads --user bob --host 127.0.0.1 --port 5001

Enter password:

Enter OTP: 654321

Authenticated.

Verifying file integrity on blockchain...

Downloading file...

Decrypting file...

File saved to downloads/report.pdf

Viewing Blockchain Information

$ java -jar client.jar --blockchain --user alice --host 127.0.0.1 --port 5001

Enter password:

Enter OTP: 123456

Authenticated.

Blockchain contains 3 blocks:

Block #1: Genesis block (2025-04-01 08:00:00)

Block #2: 2 transactions (2025-04-01 09:15:22)

Block #3: 1 transaction (2025-04-01 10:32:45)

Implementation Decisions

Cryptographic Parameters

- Kyber768 and Dilithium3 (NIST Level 3 security)
- Provides good balance between security and performance
- Has widespread library support

Blockchain Structure

- Simple hash chain validation
- File integrity verification via hashes
- Transaction signature verification with Dilithium3
- No distributed consensus mechanism required

File Handling

- Maximum file size: 5-10 MB
- Suitable for text files, PDFs, and small images
- Simple in-memory processing without chunking

Authentication

- No session expiration for simplicity
- Simple retry mechanism for failed authentication attempts
- No complex lockout policies needed

File Versioning

- Filename-based versioning (e.g., report.pdf → report_v2.pdf)
- Each version creates a new blockchain transaction
- Both versions stored independently

Transaction Receipts

- Include only essential information:
    - Transaction ID
    - Success/failure status
    - File hash
    - Timestamp
    - Operation type (upload/download)

Error Logging

- Standard Java logging to console/stdout
- Log only critical errors and major operations

- Include timestamp, operation type, and brief description
- No file/database logging needed

Key Integration Points

When implementing the flows, developers should note these critical integration points:

1. **Session Management**: The session_id flows from AuthManager through all operations
2. **Crypto Integration**: All encryption/decryption uses consistent interfaces from CryptoManager
3. **Blockchain Consistency**: All file operations must update the blockchain through BlockchainManager
4. **Message Protocol**: All communications use standardized Message objects
5. **Configuration**: Each component loads appropriate config from Config class

Conclusion

The Post-Quantum Blockchain File Sharing System demonstrates the integration of post-quantum cryptography with blockchain technology to create a secure file sharing application. The system prioritizes security through quantum-resistant algorithms while maintaining simplicity for small file transfers. The modular design allows for clear separation of concerns and future extension of functionality.