FTPLite — Software Design Specification (Section 1)

1. Architectural and component-level design

This section describes the architecture and detailed component design of FTPLite, a multi-client file transfer system.

FTPLite uses a **binary-safe TCP protocol** with structured JSON control envelopes and chunk-based file streaming. The server supports concurrent clients using one thread per connection.

FTPLite supports the following operations: - **UPLOAD**: upload a new file (identified by filename, server assigns file_id) - **DOWNLOAD**: download file content by file_id with resume - **LIST**: retrieve list of all stored file metadata sorted by newest first - **INFO**: fetch metadata for a particular file by file_id - **RESUME**: resume interrupted upload or download using resume_id - **HEARTBEAT**: connection health check - **QUIT**: end client session

Authentication, compression, and delete functionality are not supported.

1.1 Server Architecture

1.1.1 System Structure

The FTPLite Server consists of: - A **listener thread** that accepts incoming TCP clients - Multiple **ClientHandler threads**, one per active connection - A **CommandProcessor** to handle JSON control commands - A **FileManager** to manage file contents on disk - A **MetadataStore (SQLite)** to track file metadata and resume state

1.1.2 Server Protocol Overview

Messages consist of:

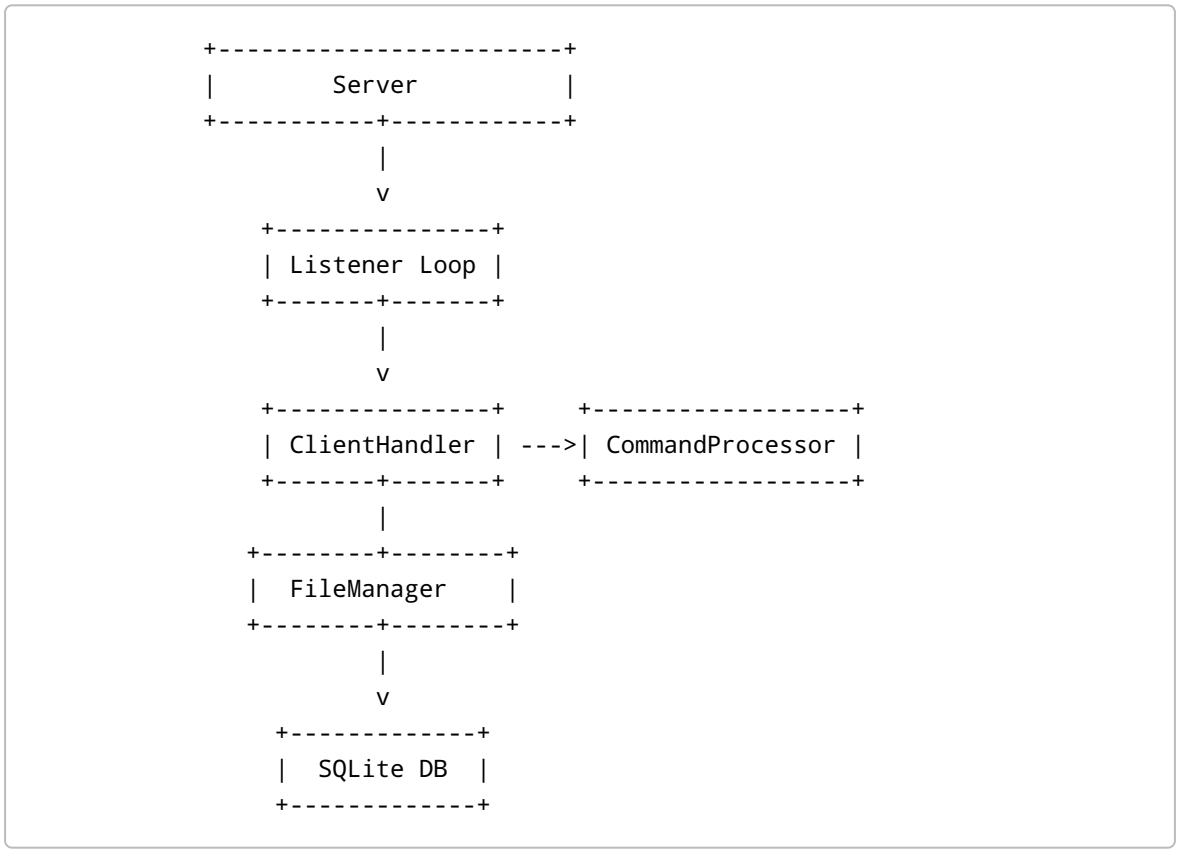
Field	Size	Type	Purpose
`length`	4 bytes	uint32	Total message size (header + body)
`type`	1 byte	uint8	`0=Control(JSON)`, `1=DataChunk`, `2=ACK/NACK`
`body`	variable	bytes	JSON control or raw binary chunk

ACK/NACK JSON example:

```
{ "ack": true, "offset": 32768 }
```

NACK may include `expected_offset`.

1.1.3 Server Architecture Diagram



1.1.4 Server Component Descriptions

Server

Responsible for startup/shutdown, listening, and managing client connections.

ClientHandler

Receives framed messages, forwards control envelopes to CommandProcessor, streams file chunks, and sends ACK/NACK.

CommandProcessor

Interprets JSON envelopes and initiates required operations.

FileManager

Handles file storage under configured root directory and streams content to/from disk.

MetadataStore (SQLite)

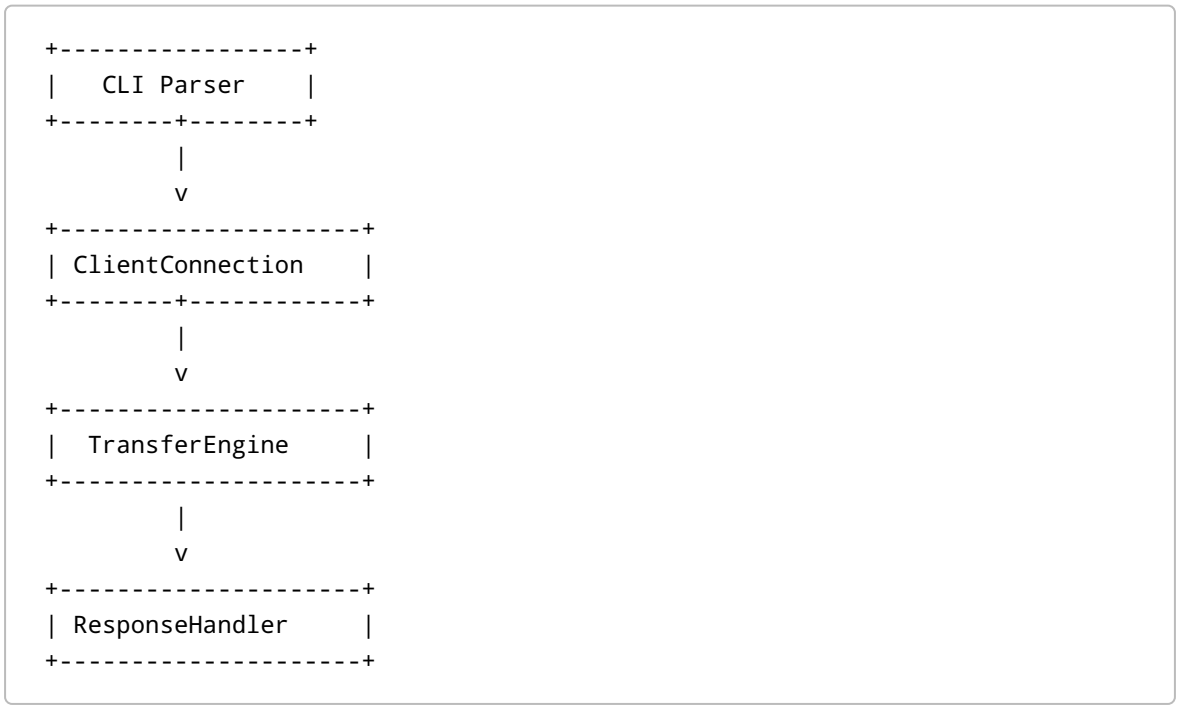
Persists file records and resume state.

1.2 Client Architecture

1.2.1 System Structure

The FTPLite Client is a console-based program that: - Parses CLI commands - Establishes a TCP connection - Sends JSON control envelopes - Streams file contents in chunks - Saves partial progress to `.resume` metadata - Handles retries, NACKs, and resume - Shows progress indicators

1.2.2 Client Architecture Diagram



1.2.3 Client Components

- **CLIParser**: input validation, converts commands to envelopes
- **ClientConnection**: framing TCP messages
- **TransferEngine**: chunk streaming, retries, HEARTBEAT
- **ResponseHandler**: handles JSON responses and errors

This architecture enables reliable resumable transfers with proper metadata tracking and integrity checks.

---

2. Data design

2.1 Internal software data structure

Framing header (in-memory representation)

```
struct FrameHeader {
  uint32_t length;
  uint8_t  type;
};
```

Control envelope (JSON ⇄ struct)

```
struct ControlEnvelope {
  std::string cmd;
  std::optional<int> file_id;
  std::optional<std::string> name;
  std::optional<uint64_t> size;
  std::optional<uint64_t> offset;
  std::optional<std::string> resume_id;
  std::optional<uint32_t> chunk_size;
};
```

ACK/NACK envelope (JSON ⇄ struct)

```
struct AckEnvelope {
  bool ack;
  uint64_t offset;
  std::optional<std::string> error_code;
  std::optional<uint64_t> expected_offset;
};
```

File metadata rows (in-memory mirrors of SQLite)

```
struct FileRow {
  int         file_id;
  std::string name;
  uint64_t    size;
  std::optional<std::string> checksum;
  std::string uploaded_at;
  int         download_count;
};

struct ResumeRow {
  std::string resume_id;
  int         file_id;
  uint64_t    offset;
  uint32_t    chunk_size;
  std::string timestamp;
};
```

2.2 Global data structure

Server-side globals

```
struct ServerConfig {
  uint16_t port;
  std::filesystem::path storage_root;
  std::filesystem::path db_path;
  int max_connections;
  std::string log_level;
};

struct ServerGlobals {
  ServerConfig config;
  std::unique_ptr<MetadataStore> meta;
  std::atomic<int> active_connections{0};
};
```

Client-side globals

```
struct ClientConfig {
  std::string server_hostport;
  std::filesystem::path download_dir;
  uint32_t chunk_size;
  int retry_limit;
  int ack_timeout_ms;
};

struct ClientSession {
  ClientConfig config;
  std::optional<std::string> resume_id;
  std::optional<int> file_id;
};
```

2.3 Temporary data structure

Client resume metadata file:

```
.ftplite_resume_<resume_id>.json
```

Example:

```
{
  "resume_id": "abc123",
  "file_id": 42,
```

```
  "server": "127.0.0.1:9000",
  "offset": 32768,
  "chunk_size": 32768
}
```

Server: partial files remain at final path.

2.4 Database schema (SQLite)

files table

```
file_id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
size INTEGER NOT NULL,
checksum TEXT,
uploaded_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
download_count INTEGER NOT NULL DEFAULT 0
```

resume table

```
resume_id TEXT PRIMARY KEY,
file_id INTEGER NOT NULL,
offset INTEGER NOT NULL,
chunk_size INTEGER NOT NULL,
timestamp DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY(file_id) REFERENCES files(file_id)
```

2.5 Class definition (.hpp excerpts)

MetadataStore

```
class MetadataStore {
public:
  explicit MetadataStore(const std::filesystem::path& db_path);
  int  insertFile(const std::string& name, uint64_t size,
std::optional<std::string> checksum);
  bool getFile(int file_id, FileRow& out);
  std::vector<FileRow> listFilesNewestFirst(int limit = 1000);
  bool upsertResume(const std::string& resume_id, int file_id, uint64_t
offset, uint32_t chunk_size);
  bool getResume(const std::string& resume_id, ResumeRow& out);
  bool updateFileSize(int file_id, uint64_t size);
  bool updateFileChecksum(int file_id, const std::string& checksum);
```

```
  bool incrementDownloadCount(int file_id);
};
```

FileManager

```
class FileManager {
public:
  explicit FileManager(std::filesystem::path root);
  bool readChunk(int file_id, uint64_t offset, size_t maxBytes,
std::vector<uint8_t>& out);
  bool writeChunk(int file_id, uint64_t offset, std::span<const uint8_t>
data);
  bool allocateForNewFile(int file_id, const std::string& name);
};
```

CommandProcessor

```
class CommandProcessor {
public:
  CommandProcessor(MetadataStore& meta, FileManager& fm);
  std::string processControl(const ControlEnvelope& in,
                             std::optional<int>& out_file_id,
                             std::optional<uint64_t>& out_offset,
                             std::optional<uint32_t>& out_chunk_size);
};
```

---

3. Performance Consideration

Goal: prioritize **high throughput** while preserving resumability and correctness.

3.1 Protocol parameters (throughput-optimized defaults)

- **Chunk size:** default 128 KiB (configurable; CLI override allowed). Larger chunks reduce per-chunk overhead.
- **ACK strategy:** cumulative ACK every *N* chunks (default **N=4**) or every **250 ms**, whichever comes first. ACK JSON includes the highest contiguous `offset` committed.
- **NACK behavior:** on checksum or gap, server sends NACK with `expected_offset`. Sender rewinds to that offset and retransmits.
- **Heartbeat:** client sends HEARTBEAT if no control or data frames exchanged for **2 s**; server replies immediately.
- **Timeouts:** ACK wait timeout **3 s**; after **retry_limit=3** failures, connection aborts and client persists resume metadata.

3.2 Networking

- Enlarge socket buffers (`SO_SNDBUF` / `SO_RCVBUF`) to **512 KiB**.

- Allow OS TCP autotuning; do not disable Nagle by default (bulk transfer). Enable `TCP_NODELAY` only for control-only sessions if needed.
- Use blocking I/O per connection to keep implementation simple; one thread per client.

3.3 File I/O

- Use buffered reads/writes with the same **128 KiB** granularity.
- Preallocate file handle on upload start; extend file size as chunks commit to minimize fragmentation.
- Flush to disk periodically (every 1–2 MiB or 1 s) to bound data loss on crash while avoiding excessive fsyncs.

3.4 Database

- Wrap resume updates in short transactions; batch `resume.offset` updates to once per cumulative ACK rather than per chunk.
- Use indices on `uploaded_at` and `file_id` as defined to keep LIST/INFO O(log n).

3.5 Concurrency limits

- `max_connections` default **100**; reject beyond limit with a control error to protect system stability.
- Thread stack size kept minimal; avoid large per-connection heap allocations beyond buffers.

3.6 Target baselines (tunable)

- 1 Gbps LAN, single client: saturate link with chunk size $\geq$ 256 KiB.
- Mixed clients (10–50): maintain $\geq$ 80% aggregate NIC utilization given disk is not the bottleneck.

Note: All parameters are configuration-driven; tuning can be done without code changes.
