

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY  
UNIVERSITY OF TECHNOLOGY  
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



## COMPUTER NETWORKS

---

### Assignment

# *DEVELOP A NETWORK APPLICATION at Ho Chi Minh City University of Technology – VNU-HCM*

---

Instructor:

Students: Nguyễn Tấn Hưng - 2352439  
Trần Gia Lâm - 2352670  
Phan Trần Quốc Tuấn - 2353273  
Nguyễn Hữu Cầu - 2352129

HO CHI MINH CITY, July 2025



## Member List & Workload

No.	Full Name	Student ID	Tasks	Completion (%)
1	Nguyễn Tân Hưng	2352439	Code for Server.py	100%
2	Nguyễn Hữu Cầu	2352129	Code for client.py & client_ui.py	100%
3	Trần Gia Lâm	2352670	Write report	100%
4	Phan Trần Quốc Tuấn	2353273	Describe overview app	100%

Table 1: Member List & Workload



## Contents

<b>1</b>	<b>Introduction and Objectives</b>	<b>6</b>
1.1	Introduction . . . . .	6
1.2	Objectives . . . . .	6
1.2.1	Overall Objective . . . . .	6
1.2.2	Specific Objectives — Phase 1 (Specification) . . . . .	6
1.2.3	Implementation & Quality Goals — Phase 2 (Implementation & Evaluation)	6
1.3	Scope & Assumptions . . . . .	7
<b>2</b>	<b>System Overview</b>	<b>7</b>
2.1	Architecture Summary . . . . .	7
2.2	Components and Roles . . . . .	7
2.2.1	Index Server . . . . .	7
2.2.2	Client/Peer . . . . .	7
2.3	Control Plane (Server API over TCP) . . . . .	7
2.4	Data Plane (Peer-to-Peer Transfer) . . . . .	8
2.5	State and Lifecycle . . . . .	8
2.6	Concurrency Model . . . . .	8
2.7	Assumptions and Non-Goals . . . . .	8
<b>3</b>	<b>Functions Definition</b>	<b>9</b>
3.1	Scope and Actors . . . . .	9
3.2	Command Surface (CLI/GUI) . . . . .	9
3.3	Function Summary . . . . .	9
3.4	REGISTER . . . . .	10
3.4.1	Purpose . . . . .	10
3.4.2	Preconditions . . . . .	10
3.4.3	Main Behavior . . . . .	10
3.4.4	Postconditions . . . . .	10
3.4.5	Errors / Acceptance . . . . .	10
3.5	PUBLISH . . . . .	10
3.5.1	Purpose . . . . .	10
3.5.2	Preconditions . . . . .	10
3.5.3	Main Behavior . . . . .	10
3.5.4	Postconditions . . . . .	10
3.5.5	Errors / Acceptance . . . . .	10
3.6	LOOKUP . . . . .	11
3.6.1	Purpose . . . . .	11
3.6.2	Preconditions . . . . .	11
3.6.3	Main Behavior . . . . .	11
3.6.4	Postconditions . . . . .	11
3.6.5	Errors / Acceptance . . . . .	11
3.7	DISCOVER . . . . .	11
3.7.1	Purpose . . . . .	11
3.7.2	Preconditions . . . . .	11
3.7.3	Main Behavior . . . . .	11
3.7.4	Postconditions . . . . .	11
3.7.5	Errors / Acceptance . . . . .	11



---

3.8	PING . . . . .	11
3.8.1	Purpose . . . . .	11
3.8.2	Preconditions . . . . .	12
3.8.3	Main Behavior . . . . .	12
3.8.4	Postconditions . . . . .	12
3.8.5	Errors / Acceptance . . . . .	12
3.8.6	Note (Consistency) . . . . .	12
3.9	HEARTBEAT . . . . .	12
3.9.1	Purpose . . . . .	12
3.9.2	Preconditions . . . . .	12
3.9.3	Main Behavior . . . . .	12
3.9.4	Postconditions . . . . .	12
3.9.5	Errors / Acceptance . . . . .	12
3.10	LEAVE . . . . .	12
3.10.1	Purpose . . . . .	12
3.10.2	Preconditions . . . . .	12
3.10.3	Main Behavior . . . . .	13
3.10.4	Postconditions . . . . .	13
3.10.5	Errors / Acceptance . . . . .	13
3.11	FETCH (Peer-to-Peer Data Plane) . . . . .	13
3.11.1	Purpose . . . . .	13
3.11.2	Preconditions . . . . .	13
3.11.3	Main Behavior . . . . .	13
3.11.4	Postconditions . . . . .	13
3.11.5	Errors / Acceptance . . . . .	13
3.12	Behavioral Requirements (Concurrency, Timeouts, Repository) . . . . .	13
3.13	Non-Goals and Optional Extensions . . . . .	14
<b>4</b>	<b>Application-Layer Protocols</b>	<b>14</b>
4.1	Transport & Framing . . . . .	14
4.2	Common Message Envelope . . . . .	14
4.3	Message Definitions . . . . .	14
4.3.1	REGISTER . . . . .	14
4.3.2	Example (JSON Lines) . . . . .	15
4.3.3	PUBLISH . . . . .	15
4.3.4	LOOKUP . . . . .	16
4.3.5	Example (JSON Lines) . . . . .	16
4.3.6	DISCOVER . . . . .	16
4.3.7	PING . . . . .	17
4.3.8	HEARTBEAT . . . . .	17
4.3.9	LEAVE . . . . .	17
4.4	Error Taxonomy . . . . .	18
4.5	Server Session State Machine . . . . .	18
4.6	Data Plane Protocol (Peer-to-Peer) . . . . .	18
4.7	Timers, Retry & Idempotency . . . . .	19
4.8	Compatibility & Extensions (Non-Blocking) . . . . .	19



---

<b>5 Detailed Design</b>	<b>20</b>
5.1 Components and Responsibilities . . . . .	20
5.2 Data Structures . . . . .	21
5.3 Control-Plane Handlers . . . . .	21
5.4 Data Plane (Peer-to-Peer) . . . . .	22
5.5 Threading and Concurrency . . . . .	22
5.6 Error Handling and Timeouts . . . . .	22
5.7 Configuration and Defaults . . . . .	23
5.8 Extensibility . . . . .	23
<b>6 Manual document</b>	<b>23</b>
6.1 Purpose and Scope . . . . .	23
6.2 System Requirements . . . . .	23
6.3 Installation . . . . .	23
6.4 Architecture Recap . . . . .	24
6.5 Starting the System . . . . .	24
6.5.1 Start Index Server . . . . .	24
6.5.2 Start a Peer (CLI) . . . . .	24
6.5.3 Start a Peer (GUI) . . . . .	24
6.6 Control Messages (How They Look) . . . . .	24
6.7 Typical Workflows . . . . .	24
6.7.1 Publish (uploader) . . . . .	24
6.7.2 Find Sources (requester) . . . . .	25
6.7.3 Fetch (peer-to-peer) . . . . .	25
6.7.4 Liveness & Lifecycle . . . . .	25
6.8 Operator Tasks (Server Side) . . . . .	25
6.9 Quick Start Recipes . . . . .	26
6.9.1 Single-machine demo . . . . .	26
6.9.2 GUI demo (two peers) . . . . .	26
6.10 Command Reference (CLI) . . . . .	26
6.11 File Management . . . . .	26
6.12 Troubleshooting . . . . .	26
6.13 Operational Best Practices . . . . .	27
6.14 Safety & Limitations . . . . .	27
<b>7 Validation and Performance Evaluation</b>	<b>27</b>
7.0.1 Client start . . . . .	27
7.0.2 Client Running . . . . .	27
<b>8 Extensions</b>	<b>28</b>
8.1 Protocol Versioning & Feature Negotiation . . . . .	28
8.2 Like-torrent Essential Components (Optional Metadata) . . . . .	28
8.3 Multi-direction / Multi-source Transferring (Swarming) . . . . .	28
8.4 Range / Resume Downloads . . . . .	29
8.5 Integrity Verification . . . . .	29
8.6 Backpressure & Rate Limiting . . . . .	29
8.7 UNPUBLISH . . . . .	29
8.8 Peer Selection Heuristics . . . . .	30
8.9 Client-side Caching & Auto Re-publish . . . . .	30



8.10 Security (Stretch Goal) . . . . .	30
8.11 NAT Traversal (Stretch Goal) . . . . .	30
8.12 Atomic Finalize & Repository Policy . . . . .	30
8.13 Diagnostics & Admin . . . . .	30
8.14 Error Map (Extended) . . . . .	30
<b>9 Appendices</b>	<b>30</b>
<b>10 REFERENCES</b>	<b>31</b>



# 1 Introduction and Objectives

## 1.1 Introduction

We implement a P2P file-sharing app with a *central index server* that stores only *metadata* (which host has which files). File data never goes through the server; peers transfer bytes directly over TCP to avoid a server bottleneck. The **control plane** uses TCP with **JSON Lines** (one JSON per line, CRLF, UTF-8) and supports: REGISTER, PUBLISH, LOOKUP, DISCOVER, PING, HEARTBEAT, LEAVE. On REGISTER, the server issues a session\_id and ttl; the client sends periodic HEARTBEAT to keep the session alive. The **data plane** is minimal: the downloader sends GET <fname> and the uploader replies OK 200 with a Size header, then streams the raw bytes (errors like ERR 404). The client is multi-threaded so it can upload and download concurrently. The CLI/GUI exposes the required minimum: *publish* and *obtain* (lookup → fetch), plus DISCOVER/PING for visibility/liveness.

## 1.2 Objectives

### 1.2.1 Overall Objective

Design and implement a reliable TCP-based P2P file-sharing application with a clear application-layer protocol and a clean separation between control and data planes.

### 1.2.2 Specific Objectives — Phase 1 (Specification)

1. **Function Definition:** specify purpose, inputs/outputs, and error cases for:
  - Server/API: REGISTER, PUBLISH, LOOKUP, DISCOVER, PING, HEARTBEAT, LEAVE.
  - Peer-to-peer: **FETCH** (downloader ↔ uploader) after a successful LOOKUP.
2. **Protocol Design:** standardize *JSON Lines* + *CRLF* and required fields (`type`, `cseq`, `session_id`; replies include `ok/code`), define an error taxonomy (e.g., 200/400/401/404/409/410/429/500/503), basic timeouts, and the request/reply sequences for the operations above.
3. **Security/Integrity (baseline vs. extensions):** the core transfers bytes as-is; *hash verification* and *resume* are recorded as *extensions* (not mandatory in the core).

### 1.2.3 Implementation & Quality Goals — Phase 2 (Implementation & Evaluation)

4. **Concurrency & Robustness:** multi-threaded client (listener serves multiple GET connections, workers handle LOOKUP/FETCH); explicit error handling on both planes.
5. **Usability:** provide the minimum per brief: *publish* and *obtain* on the client; DISCOVER/PING via the API for inspection/liveness.
6. **Validation & Performance:** sanity tests (REGISTER; PUBLISH→DISCOVER; LOOKUP→FETCH; PING; LEAVE) and basic metrics (lookup latency, fetch throughput, error rate, resource usage).
7. **Extensions (optional):** peer selection (RTT/load), Range/OK 206 (resume), LOOKUP caching, auto re-publish after fetch, bandwidth caps, hash-based integrity verification.



### 1.3 Scope & Assumptions

Peers are reachable over IP (same LAN or routable). Advanced NAT traversal, DHT-based discovery, and end-to-end encryption are out of scope for the core (they may appear as extensions). The server holds only metadata; the core data plane is **GET** → **OK 200 + Size** → bytes. Resume and cryptographic verification are considered extensions.

## 2 System Overview

### 2.1 Architecture Summary

Our system is a P2P file-sharing app with a *central index server* that stores only *metadata* (which host has which files). File bytes never pass through the server; peers transfer data directly over TCP. The **control plane** (registration, catalog updates, lookups, liveness) uses JSON Lines over TCP (one JSON per line, CRLF, UTF-8). The **data plane** (file transfer) is a minimal text-header exchange followed by a raw byte stream.

### 2.2 Components and Roles

#### 2.2.1 Index Server

Keeps a *session table* (issued **session\_id**, **ttl**, **last\_seen**, host info) and a *file index* mapping **fname** to a *list* of peer descriptors {**host**, **ip**, **p2p\_port**, **size**, **last\_seen**, optional **hash**}. The server receives JSON-Line requests and returns JSON-Line responses; replies include common fields **type**, **cseq**, **ok/code**, **time** plus payload.

#### 2.2.2 Client/Peer

Registers with the server, publishes its local catalog, looks up sources, then downloads directly from peers. Each client also runs a TCP listener to serve inbound **GET** requests (uploader) while it can simultaneously act as a downloader (concurrent uploads and downloads).

### 2.3 Control Plane (Server API over TCP)

**Transport & framing.** TCP; one JSON object per message; messages delimited by CRLF (\r\n).

#### Supported operations.

- **REGISTER** — start a session (server returns **session\_id**, **ttl**).
- **PUBLISH** — announce files: {**fname**, **size**, optional **hash**}.
- **LOOKUP** — ask who has a given **fname**; returns peer descriptors.
- **DISCOVER** — list the catalog of a specific host.
- **PING** — liveness check for a host.
- **HEARTBEAT** — periodic keep-alive to refresh the session.
- **LEAVE** — graceful session termination and index cleanup.

#### Example (short).



```
# REGISTER
{"type":"REGISTER","cseq":1,"host":{"name":"alice","p2p_port":6000}}\r\n
{"type":"REGISTER-OK","cseq":1,"ok":true,"code":200,
 "session_id":1,"ttl":60,"time":"..."}\r\n

# LOOKUP
{"type":"LOOKUP","cseq":2,"session_id":1,"fname":"a.txt"}\r\n
{"type":"LOOKUP-OK","cseq":2,"ok":true,"code":200,
 "peers": [{"host":"bob","ip":"10.0.0.2","p2p_port":6001,
             "size":1234,"last_seen":"..."}],
 "time":"..."}\r\n
```

## 2.4 Data Plane (Peer-to-Peer Transfer)

After a successful LOOKUP, the downloader connects to the uploader's p2p\_port and uses a minimal exchange:

```
# Downloader -> Uploader
GET <fname>\r\n
\r\n

# Uploader -> Downloader (success)
OK 200\r\n
Size: <n>\r\n
\r\n
<file-bytes...>

# Or error
ERR 404 Not Found\r\n
\r\n
```

(Resumable ranges and hash verification are treated as extensions; the core streams full files with OK 200 + Size.)

## 2.5 State and Lifecycle

REGISTER (get session\_id, ttl) → optionally PUBLISH → on demand LOOKUP → peer-to-peer transfer → periodic HEARTBEAT → visibility via DISCOVER/PING → LEAVE.

## 2.6 Concurrency Model

A client runs a TCP listener (thread-per-connection) to serve multiple inbound **GETs**, has workers for outbound LOOKUP/fetch, and a periodic HEARTBEAT task — enabling simultaneous uploads and downloads.

## 2.7 Assumptions and Non-Goals

Peers are reachable over IP (same LAN or routable). Advanced NAT traversal, DHT, and end-to-end encryption are out of scope for the core. The server keeps only metadata (no file data). Integrity checks (hashes) and resumable transfer can be added as extensions.



### 3 Functions Definition

#### 3.1 Scope and Actors

This section defines the user-visible *functions* of the P2P file-sharing system and the observable behaviors at the application boundary.

- **Peer/Client** registers, publishes files, discovers/looks up sources, fetches files P2P, and maintains liveness via heartbeats.
- **Index Server** maintains sessions and a metadata index mapping `fname` to providers; it does *not* relay file contents.
- **User** interacts via CLI/GUI. The required minimum is: client `publish/fetch`; operator queries `discover/ping` are invoked *through the client* (control-plane API).

#### 3.2 Command Surface (CLI/GUI)

- **Client (CLI):** `publish <fname>`, `lookup <fname>`, `fetch <fname>`, `leave`
- **Client (GUI):** pick `<local_path>` to copy into the repository and set `<fname>`; then `lookup → fetch`
- **Server-side ops (via API):** `discover <hostname>`, `ping <hostname>`

*Note.* The codebase supports publishing from GUI by choosing a local file (copied into the repo). The CLI's `publish` expects the file to already exist inside the repository directory under `<fname>`.

#### 3.3 Function Summary

#	Function	Actor	Description
1	REGISTER	Client ↔ Server	Establish/refresh a session; server returns <code>session_id</code> and <code>ttl</code> .
2	PUBLISH	Client → Server	Advertise local files { <code>fname</code> , size (, hash?)}; server upserts index.
3	LOOKUP	Client ↔ Server	Request providers for <code>fname</code> ; server returns a list of peers (possibly empty).
4	DISCOVER	Client/Operator ↔ Server	List the catalogue (files) of a given <code>hostname</code> .
5	PING	Client/Operator ↔ Server	Report liveness of <code>hostname</code> as <code>alive:{true false}</code> .
6	HEARTBEAT	Client → Server	Keep the session alive; server refreshes <code>last_seen</code> and echoes <code>ttl</code> .
7	LEAVE	Client → Server	Gracefully end the session; server prunes this host from all provider lists.
8	FETCH (Data)	Client ↔ Client	Transfer file bytes directly P2P using a minimal request/response then raw stream.

Table 2: Functions overview



### 3.4 REGISTER

#### 3.4.1 Purpose

Establish (or refresh) a client session so subsequent control-plane requests are attributable.

#### 3.4.2 Preconditions

Client can reach the Index Server; client knows the control-plane TCP address/port.

#### 3.4.3 Main Behavior

1. Client submits host information `{name, p2p_port}`; the server infers `ip` from the TCP connection.
2. Server creates or refreshes a session and sets expiry based on `ttl`.
3. Server returns `session_id` and `ttl`.

#### 3.4.4 Postconditions

A valid session exists; the host table is up to date.

#### 3.4.5 Errors / Acceptance

Missing/invalid fields → failure; acceptance: `ok=true` with non-empty `session_id`.

### 3.5 PUBLISH

#### 3.5.1 Purpose

Advertise local files (logical name `fname`) so other peers can discover and fetch them.

#### 3.5.2 Preconditions

Valid `session_id`; file exists in the local repository (GUI can copy from `<local_path>` into the repo); repository naming policy satisfied.

#### 3.5.3 Main Behavior

1. Client supplies a list `files[{fname, size (, hash?)}]`.
2. Server upserts index entries: `fname` → providers including this host with metadata.
3. Server returns `accepted` (number of valid entries).

#### 3.5.4 Postconditions

`fname` becomes visible in the index and associated with this host.

#### 3.5.5 Errors / Acceptance

Unknown/expired session → unauthorized; acceptance: `accepted` equals the number of valid inputs.



## 3.6 LOOKUP

### 3.6.1 Purpose

Retrieve the current list of peers that can serve `fname`.

### 3.6.2 Preconditions

Valid session; `fname` provided.

### 3.6.3 Main Behavior

1. Client requests `LOOKUP(fname)`.
2. Server responds with `peers[{host, ip, p2p_port, size (), hash?}, last_seen}]` (possibly empty).

### 3.6.4 Postconditions

Client has candidates to initiate **FETCH**.

### 3.6.5 Errors / Acceptance

If no provider exists, return an empty list; acceptance: well-formed peer list.

## 3.7 DISCOVER

### 3.7.1 Purpose

List all files currently published by a given `hostname`.

### 3.7.2 Preconditions

Valid session; target `hostname` provided.

### 3.7.3 Main Behavior

Server returns `files[{fname, size (), hash?}]` for that host.

### 3.7.4 Postconditions

Read-only; no state change.

### 3.7.5 Errors / Acceptance

Unknown host → empty result or explicit “not found” per policy; acceptance: list returned successfully.

## 3.8 PING

### 3.8.1 Purpose

Report whether a `hostname` currently has an active session.



### 3.8.2 Preconditions

None beyond reachability of the server.

### 3.8.3 Main Behavior

Server checks liveness and returns `alive:{true|false}`.

### 3.8.4 Postconditions

None.

### 3.8.5 Errors / Acceptance

Always returns a status; acceptance: `alive=true` for active session, otherwise `false`.

### 3.8.6 Note (Consistency)

Earlier drafts allowed PING to include `ip` and `p2p_port`; the current implementation returns only `alive`. Extensions remain future work.

## 3.9 HEARTBEAT

### 3.9.1 Purpose

Keep the session alive and refresh liveness metadata.

### 3.9.2 Preconditions

An existing `session_id`.

### 3.9.3 Main Behavior

1. Client periodically sends HEARTBEAT.
2. Server updates `last_seen` and may echo the current `ttl`.

### 3.9.4 Postconditions

Session remains valid while heartbeats continue.

### 3.9.5 Errors / Acceptance

Unknown session → failure; acceptance: `ok=true` and freshness updated.

## 3.10 LEAVE

### 3.10.1 Purpose

Gracefully terminate a session and clean server-side references to this host.

### 3.10.2 Preconditions

Valid `session_id`.



### 3.10.3 Main Behavior

1. Client sends LEAVE.
2. Server invalidates the session and removes this host from all provider lists.

### 3.10.4 Postconditions

Future requests require a new REGISTER; LOOKUP no longer lists this host.

### 3.10.5 Errors / Acceptance

Idempotent: if already invalid, reply OK/no-op; acceptance: reply indicates successful cleanup.

## 3.11 FETCH (Peer-to-Peer Data Plane)

### 3.11.1 Purpose

Obtain file contents directly from a peer without involving the server in content transfer.

### 3.11.2 Preconditions

A prior LOOKUP returned at least one source; uploader is listening on its P2P port.

### 3.11.3 Main Behavior

1. Downloader opens a TCP connection to the uploader and requests `fname`.
2. Uploader replies success (`OK 200 + Size`) and streams the bytes, or reports `ERR 404`.
3. Downloader writes to a temporary path and finalizes into the repository atomically.

### 3.11.4 Postconditions

The file becomes locally available for future sharing.

### 3.11.5 Errors / Acceptance

Connection or transfer failure may be retried by policy; acceptance: received byte count equals advertised size (hash check is optional if provided).

## 3.12 Behavioral Requirements (Concurrency, Timeouts, Repository)

- **Concurrency.** Server and peers handle multiple simultaneous connections (thread-per-connection or equivalent), enabling concurrent downloads.
- **Timeouts/Retry.** Control-plane requests and data transfers use finite timeouts; failed attempts may be retried with bounded backoff.
- **Repository Semantics.** The downloader writes directly to the target path and closes on success; on failure it cleans up the partial file when possible. (Atomic finalize via temp-rename can be added as an extension.)



### 3.13 Non-Goals and Optional Extensions

Out of scope for the core but compatible with the design:

- **UNPUBLISH** (withdraw a file) and **Range/Resume** downloads (OK 206-like).
- **Integrity** via mandatory content hash; **Rate limiting**; **Peer selection** heuristics.

These may appear as extensions and are not required for correctness of the core functions above.

## 4 Application-Layer Protocols

### 4.1 Transport & Framing

All control-plane messages are exchanged over **TCP** using a **single persistent connection per client session**. Each application message is one **JSON object** terminated by \r\n (JSON Lines). For every request, the server returns *exactly one* reply with the same **cseq**. In addition, the server *may* push asynchronous events (e.g., new client or publish notifications) on the same connection, interleaved with replies.

### 4.2 Common Message Envelope

Requests carry **type**, **cseq**, and (for authenticated ops) **session\_id**. Replies echo **cseq** and carry **ok**, **code**, and **time**.

#	Request Field	Type/Req.	Description
1	type	string / req	Message type (e.g., "REGISTER", "LOOKUP").
2	cseq	int / req	Client sequence (monotonic per connection).
3	session_id	int / opt	Present after REGISTER for authenticated ops.

Table 3: Common request envelope (control-plane)

#	Reply Field	Type/Req.	Description
1	type	string / req	Reply type (e.g., "REGISTER-OK", "LOOKUP-OK").
2	cseq	int / req	Echoed from request.
3	ok	bool / req	true on success, else false.
4	code	int / req	Application status (200, 400, 401, 404, 500, ...).
5	time	string / req	Server time (ISO 8601).

Table 4: Common reply envelope (control-plane)

### 4.3 Message Definitions

#### 4.3.1 REGISTER

**Semantics.** Create/refresh a client session; the server issues **session\_id** and **ttl** (seconds).



#	REGISTER Request	Type/Req.	Description
1	type	string / req	"REGISTER".
2	cseq	int / req	Client sequence.
3	host.name	string / req	Logical hostname (e.g., "alice").
4	host.ip	string / opt	Reachable IP if known (server can infer from TCP).
5	host.p2p_port	int / req	Peer listener port (e.g., 6000).
6	host.agent	string / opt	Client agent/version (e.g., "p2p/1.0").

Table 5: REGISTER — request fields

#	REGISTER Reply	Type/Req.	Description
1	type,cseq,ok,code,time	mixed / req	Common reply envelope.
2	session_id	int / req	Identifier to use in subsequent requests.
3	ttl	int / req	Session TTL in seconds.

Table 6: REGISTER — reply fields

#### 4.3.2 Example (JSON Lines)

```
{"type": "REGISTER", "cseq": 1,
  "host": {"name": "alice", "ip": "192.0.2.10", "p2p_port": 6000, "agent": "p2p/1.0"}
}\r\n
{"type": "REGISTER-OK", "cseq": 1, "ok": true, "code": 200,
  "session_id": 42, "ttl": 60, "time": "2025-11-07T11:28:00Z"}
}\r\n
```

#### 4.3.3 PUBLISH

**Semantics.** Advertise files so other peers can discover them.

#	PUBLISH Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.
2	files[].fname	string / req	Logical filename.
3	files[].size	int / req	Size in bytes.
4	files[].hash	string / opt	Content hash if computed.

Table 7: PUBLISH — request fields

#	PUBLISH Reply	Type/Req.	Description
1	type,cseq,ok,code,time	mixed / req	Common reply envelope.
2	accepted	int / req	Number of accepted entries.

Table 8: PUBLISH — reply fields



#### 4.3.4 LOOKUP

**Semantics.** Return the list of providers for fname.

#	LOOKUP Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.
2	fname	string / req	Logical filename to locate.

Table 9: LOOKUP — request fields

#	LOOKUP Reply	Type/Req.	Description
1	type,cseq,ok,code,time	mixed / req	Common reply envelope.
2	peers[].host	string / req	Provider hostname.
3	peers[].ip	string / req	Provider IP.
4	peers[].p2p_port	int / req	Provider peer port.
5	peers[].size	int / req	File size in bytes.
6	peers[].hash	string / opt	Optional content hash.
7	peers[].last_seen	string / opt	Optional liveness hint (timestamp).

Table 10: LOOKUP — reply payload

#### 4.3.5 Example (JSON Lines)

```
{"type": "LOOKUP", "cseq": 3, "session_id": 42, "fname": "kit.zip"}\r\n{"type": "LOOKUP-OK", "cseq": 3, "ok": true, "code": 200, "time": "...", "peers": [{"host": "alice", "ip": "192.0.2.10", "p2p_port": 6000, "size": 7340032, "hash": null, "last_seen": "2025-11-07T11:28:35Z"}]}\r\n
```

#### 4.3.6 DISCOVER

**Semantics.** List all files published by a given host.

#	DISCOVER Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.
2	host	string / req	Target hostname.

Table 11: DISCOVER — request fields



#	DISCOVER Reply	Type/Req.	Description
1	type,cseq,ok,code,timemixed / req		Common reply envelope.
2	files[] . fname	string / req	Logical filename.
3	files[] . size	int / req	Size in bytes.
4	files[] . hash	string / opt	Optional content hash.

Table 12: DISCOVER — reply payload

#### 4.3.7 PING

**Semantics.** Report whether a host currently has an active session.

#	PING Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.
2	host	string / req	Target hostname.

Table 13: PING — request fields

#	PING Reply	Type/Req.	Description
1	type,cseq,ok,code,timemixed / req		Common reply envelope.
2	alive	bool / req	true if the host has an active session; else false.

Table 14: PING — reply payload

#### 4.3.8 HEARTBEAT

**Semantics.** Keep the session alive; refresh liveness metadata.

#	HEARTBEAT Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.

Table 15: HEARTBEAT — request fields

#	HEARTBEAT Reply	Type/Req.	Description
1	type,cseq,ok,code,timemixed / req		Common reply envelope.
2	ttl	int / opt	Current TTL (seconds), if echoed.

Table 16: HEARTBEAT — reply payload

#### 4.3.9 LEAVE

**Semantics.** End session; prune all index entries belonging to this host.



#	LEAVE Request	Type/Req.	Description
1	type,cseq,session_id	mixed / req	Envelope.

Table 17: LEAVE — request fields

#	LEAVE Reply	Type/Req.	Description
1	type,cseq,ok,code,time	mixed / req	Common reply envelope.
2	removed	int / opt	Number of pruned entries (if reported).

Table 18: LEAVE — reply payload

#### 4.4 Error Taxonomy

#	Code	Name	Meaning
1	200	OK	Successful operation.
2	400	BAD_REQUEST	Malformed/missing field(s).
3	401	UNAUTHORIZED	Missing/invalid session_id.
4	404	NOT_FOUND	No such host/file (e.g., LOOKUP empty, GET missing).
5	409	CONFLICT	Duplicate-illegal state (reserved for extensions).
6	429	TOO_MANY_REQUESTS	Backpressure/throttling (extensions).
7	500	INTERNAL_ERROR	Unexpected server error.

Table 19: Application error codes

#### 4.5 Server Session State Machine

**States:** NEW → REGISTERED → LEAVING / EXPIRED.  
**Transitions:** NEW on REGISTER → REGISTERED; REGISTERED on HEARTBEAT (refresh TTL); REGISTERED on timeout → EXPIRED; REGISTERED on LEAVE → LEAVING (prune) → NEW.

Figure 1: Framed server session state machine (placeholder)

#### 4.6 Data Plane Protocol (Peer-to-Peer)

One TCP connection per file transfer. Minimal request/response header followed by a raw byte stream.

Client --> Uploader:

GET <fname>\r\n\r\n

Uploader --> Client (success):

OK 200\r\n

Size: <n>\r\n

\r\n

<-- n raw bytes -->



```
Uploader --> Client (not found):  
ERR 404 Not Found\r\n\r\n
```

**Acceptance/MUST.** The downloader **MUST** verify that the total received bytes equals the advertised **Size**. If a hash was published, the downloader **SHOULD** verify the content hash after completion.

#### 4.7 Timers, Retry & Idempotency

- **Timeouts.** Control-plane connect/read: 3–5 s (per request); data-plane inactivity:  $\approx$ 30 s.
- **Retry/backoff.** Up to 3 retries with bounded exponential backoff for transient errors (policy-level).
- **Idempotency.** REGISTER, HEARTBEAT, and PUBLISH are designed to be idempotent; replays do not corrupt server state.

#### 4.8 Compatibility & Extensions (Non-Blocking)

The protocol is forward-compatible with optional features: **UNPUBLISH**, ranged/resumable transfers (206-like), integrity enforcement (mandatory hash), rate limiting, and peer selection. These extensions do not alter the base semantics above.



## 5 Detailed Design

### 5.1 Components and Responsibilities

#	Component	Responsibilities
1	<b>TCP Listener &amp; Dispatcher (Server)</b>	Accept connections; spawn per-connection worker that reads a <i>loop</i> of JSON Lines (CRLF); parse type; dispatch handler; keep the connection open for replies and async events.
2	<b>Event Broadcaster (Server)</b>	Push asynchronous {"event": ...} messages (e.g., NEW_CLIENT, PUBLISH, LEAVE) to all connected clients over their persistent control channels.
3	<b>SessionManager (Server)</b>	Issue session_id with ttl; track expiry (epoch), last_seen (ISO 8601), host, ip, p2p_port; maintain hostname→session_id map.
4	<b>FileIndex (Server)</b>	Map fname → list of peer entries {host, ip, p2p_port, size, (optional hash), last_seen}. Upsert on PUBLISH; prune on LEAVE/expiry.
5	<b>CleanupWorker (Server)</b>	Periodically (every 10s) remove expired sessions and prune file entries of inactive hosts.
6	<b>Reply Builder (Server)</b>	Build common reply envelopes via make_reply(req, type, ok, code, extra).
7	<b>BufferedConnection (Shared)</b>	Utility wrapper to buffer incoming bytes and parse one JSON Line per call; also provides send_msg(obj). Used by both server and client.
8	<b>ControlChannel (Client)</b>	One persistent TCP connection to server (JSON Lines). A receiver thread processes: (i) replies (matched by cseq); (ii) async events. A pending_replies map (cseq→Event) enables blocking waits with timeout.
9	<b>HeartbeatTask (Client)</b>	Send HEARTBEAT every 30s to keep the session alive (server default ttl=60s).
10	<b>Publisher (Client)</b>	Announce files via PUBLISH. In GUI, a selected local file may be copied into the repository before publish.
11	<b>Lookup/Discover APIs (Client)</b>	Request LOOKUP/DISCOVER to get providers or a host catalog.
12	<b>P2P Uploader (Client)</b>	TCP listener on p2p_port; handle GET and stream bytes with OK 200 + Size.
13	<b>P2P Downloader (Client)</b>	Pick a peer from LOOKUP result; download the file; verify total bytes received equals Size; remove partial file on failure.
14	<b>UI/CLI (Client)</b>	Buttons/commands: REGISTER/CONNECT, PUBLISH, LOOKUP, DISCOVER, PING, LEAVE, FETCH.

Table 20: Components & Responsibilities



## 5.2 Data Structures

#	Structure	Fields
1	<b>Session</b>	<code>session_id:int, host:string, ip:string, p2p_port:int, ttl:int, expiry:float, last_seen:string, agent?:string</code>
2	<b>FileIndex</b>	<code>"fname": [ {host, ip, p2p_port, size, hash?, last_seen}, ... ], ...</code>
3	<b>Reply Envelope</b>	<code>"type":string, "cseq":int, "ok":bool, "code":int, "time":string, ...</code>
4	<b>Event Message (Server→Client)</b>	<code>"event": "NEW_CLIENT" "PUBLISH" "LEAVE", "host":string, "files": [string], "time":string</code>
5	<b>PendingReplies (Client)</b>	<code>pending_replies:{cseq→Event}, reply_data:{cseq→dict}</code>

Table 21: Key Data Structures

## 5.3 Control-Plane Handlers

#	Operation	Behavior (inputs → outputs)
1	REGISTER	Create/refresh session; reply with <code>session_id</code> , <code>ttl</code> . Broadcast <code>NEW_CLIENT</code> event.
2	PUBLISH	Upsert <code>{fname, size, hash?}</code> for the caller's host ( <code>ip, p2p_port, last_seen</code> ); reply <code>accepted</code> . Broadcast PUBLISH.
3	LOOKUP	Return providers for a given <code>fname</code> (list of peer entries).
4	DISCOVER	List catalog for a specific host.
5	PING	Return <code>alive=true false</code> (code 200 if alive, 404 otherwise).
6	HEARTBEAT	Refresh liveness; reply echoes current <code>ttl</code> .
7	LEAVE	Invalidate session; prune host entries; reply may include removed count; broadcast LEAVE.

Table 22: Control-Plane Handlers



## 5.4 Data Plane (Peer-to-Peer)

Request / Response framing
# Downloader -> Uploader GET <fname>\r\n\r\n
# Uploader -> Downloader (success) OK 200\r\nSize: <n>\r\n\r\n<file-bytes...>
# Uploader -> Downloader (not found) ERR 404 Not Found\r\n\r\n

Table 23: P2P Transfer Protocol

## 5.5 Threading and Concurrency

#	Process	Concurrency Model
1	Server	Main acceptor + per-connection worker threads (control plane) + background <i>Cleanup Worker</i> . Event Broadcaster iterates all active connections.
2	Client	Persistent control socket (one receiver thread; requests are serialized by <code>cseq</code> ); TCP listener for GET (thread-per-connection); periodic HEARTBEAT; one or more downloader workers.

Table 24: Concurrency Model

## 5.6 Error Handling and Timeouts

#	Policy	Details
1	Control-plane timeouts	Client waits up to <b>5s</b> per request (synthetic code=408 on timeout). One reply per <code>cseq</code> .
2	Data-plane inactivity	$\approx 30\text{s}$ idle timeout on peer sockets.
3	Retry/backoff	Up to 3 retries with bounded exponential backoff (policy-level; not enforced by server).
4	Idempotency	REGISTER, HEARTBEAT, PUBLISH are safe to replay.
5	Acceptance (data)	Downloader <b>MUST</b> verify received bytes == <code>Size</code> ; remove partial file on failure. (Hash verification optional if provided.)

Table 25: Timeouts, Retry, Idempotency, Acceptance



## 5.7 Configuration and Defaults

#	Parameter	Default / Note
1	Server bind	0.0.0.0:5050.
2	Client server addr	127.0.0.1:5050 (configurable).
3	Session TTL	60s default on server; HEARTBEAT every 30s.
4	Cleanup interval	10s (server <i>Cleanup Worker</i> ).
5	Request timeout	5s per control-plane request.
6	Repository	Local directory ./<name>_repo; ensure free disk space.

Table 26: Config & Defaults

## 5.8 Extensibility

#	Extension	Compatibility Note
1	Integrity hash	Optional in PUBLISH/LOOKUP/(data-plane trailer); backward-compatible.
2	Resumable ranges	GETRANGE/OK 206; base framing kept.
3	Multi-source (swarm)	Split file into chunks; control-plane unchanged.
4	Rate limiting / fairness	Token-bucket or fair-queue at uploader; independent of control-plane.

Table 27: Optional Extensions (Non-Blocking)

# 6 Manual document

## 6.1 Purpose and Scope

This manual explains how to install, configure, run, and operate the P2P file sharing system (central index server + peers). It covers command workflows (Publish/Lookup/Fetch, Discover, Ping), the optional GUI client, and operator tasks.

## 6.2 System Requirements

- **OS:** Windows 10+, Ubuntu 20.04+/Debian, or macOS 12+.
- **Python:** 3.8+ (standard library; `tkinter` for GUI).
- **Network:** Peers must be reachable over IP/TCP; allow inbound connections on each peer's P2P port (e.g., 6000/6001) and the server control port (default 5050).

## 6.3 Installation

1. Install Python 3.8+ and ensure `python/pip` are on PATH.
2. Copy the project folder containing `server.py`, `client.py`, `client_ui.py`, `utils.py`.
3. (Linux only) If GUI is needed, install `python3-tk`.



## 6.4 Architecture Recap

- **Index Server** stores only metadata: which host has which files; it never relays file bytes.
- **Control plane:** JSON Lines over TCP (one JSON object terminated by CRLF \r\n) on a persistent connection.
- **Data plane:** downloader connects to peer's p2p\_port; request GET <fname>; uploader replies OK 200 + Size: n then streams raw bytes (or ERR 404).
- The client is multi-threaded and can upload and download concurrently.

## 6.5 Starting the System

### 6.5.1 Start Index Server

```
python server.py
```

The console prints the bind address and port (e.g., 0.0.0.0:5050).

### 6.5.2 Start a Peer (CLI)

```
python client.py --name alice --p2p-port 6000
```

An interactive shell appears with commands: publish, lookup, fetch, discover, ping, leave, exit.

### 6.5.3 Start a Peer (GUI)

```
python client_ui.py
```

Input **CLIENT NAME** and **P2P PORT**, click **START CLIENT**, then use buttons: **PUBLISH FILE**, **LOOKUP**, **DISCOVER**, **PING**, **LEAVE**. In the GUI, a fetch is triggered after you select a result from **LOOKUP**. The activity log shows request/response messages.

## 6.6 Control Messages (How They Look)

Each message is one JSON object ended by \r\n (JSON Lines).

```
# REGISTER
{"type":"REGISTER","cseq":1,"host":{"name":"alice","p2p_port":6000}}\r\n
# Typical reply
>{"type":"REGISTER-OK","cseq":1,"ok":true,"code":200,"session_id":1,"ttl":60,"time":"..."}\r\n
```

## 6.7 Typical Workflows

### 6.7.1 Publish (uploader)

1. Place the file in the client's repository directory (GUI can copy from a chosen local path into the repo before publish).
2. CLI: publish <fname>; GUI: click **PUBLISH FILE** and choose a file.
3. The server records {fname, size} under your host entry; reply PUBLISH-OK.



#### 6.7.2 Find Sources (requester)

```
lookup <fname>      # list peers that have <fname>
discover <hostname> # list files published by <hostname>
```

The server returns peer descriptors {host, ip, p2p\_port, size, last\_seen,...}.

#### 6.7.3 Fetch (peer-to-peer)

After lookup, fetch the file directly from a listed peer.

**CLI:**

```
fetch <fname>
```

**Wire framing (for reference):**

```
# Downloader -> Uploader
GET <fname>\r\n
\r\n
# Uploader -> Downloader (success)
OK 200\r\n
Size: <n>\r\n
\r\n
<file-bytes...>
# Not found
ERR 404 Not Found\r\n
\r\n
```

The downloader verifies total received bytes equals the advertised **Size** and cleans up any partial file on failure.

#### 6.7.4 Liveness & Lifecycle

```
ping <hostname> # check if host is alive
leave            # gracefully remove session and index entries
```

The client automatically sends HEARTBEAT every **30s** (server default ttl is **60s**) to keep the session alive.

### 6.8 Operator Tasks (Server Side)

- Monitor server console: it logs REGISTER, PUBLISH, LOOKUP, DISCOVER, PING, HEARTBEAT, and cleanup events.
- Ensure the control-plane port (default 5050) is open on the host firewall.
- Verify stale-session cleanup (entries pruned after TTL expiry).



## 6.9 Quick Start Recipes

### 6.9.1 Single-machine demo

1. Terminal A: `python server.py`
2. Terminal B: `python client.py --name alice --p2p-port 6000` then publish `demo.txt`
3. Terminal C: `python client.py --name bob --p2p-port 6001` then lookup `demo.txt` and fetch `demo.txt`.

### 6.9.2 GUI demo (two peers)

Run `python client_ui.py` twice (two processes), use names `alice` and `bob`; publish on `alice`, LOOKUP on `bob`, then select a peer in the results to start the fetch.

## 6.10 Command Reference (CLI)

- `publish <fname>` — announce `fname` (size auto-detected) to the index.
- `lookup <fname>` — list peers that have `fname`.
- `fetch <fname>` — download `fname` from a listed peer.
- `discover <host>` — list the catalog for `host`.
- `ping <host>` — check liveness (`alive:true/false`).
- `leave` — gracefully end the session and remove host entries.
- `exit` — quit client.

## 6.11 File Management

- Put files to be published under the client's repository directory (rename locally to handle duplicates).
- Ensure free disk space for downloads; the downloader writes exactly `Size` bytes and removes partial files on failure.

## 6.12 Troubleshooting

Symptom	Likely cause / Fix
<code>lookup</code> returns empty	No peer has published that filename; ensure uploader ran <code>publish</code> successfully; check server log for a PUBLISH event.
Fetch fails or stalls	Peer's <code>p2p_port</code> blocked by firewall; open the port; verify IP/port from <code>lookup</code> result; ensure the uploader is running.
<code>ping</code> shows <code>alive:false</code>	Host not registered or TTL expired; (re)start the client; HEARTBEAT runs every 30s to keep the session alive.
<code>leave</code> has no effect	You may be using a different host/session; check server log; restart client and register again.
GUI does not start	Install <code>python3-tk</code> (Linux) or ensure Python's Tkinter is available.

Table 28: Troubleshooting quick reference

## 6.13 Operational Best Practices

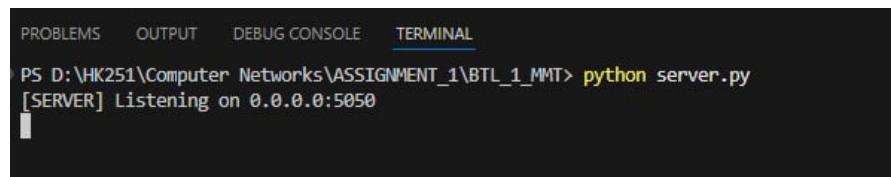
- Run `leave` before closing a client to keep the index clean.
- Keep `HEARTBEAT` running for long-lived sessions to avoid TTL expiry.
- For multiple concurrent downloads to the same uploader, ensure adequate uplink bandwidth and CPU.

## 6.14 Safety & Limitations

Authentication and encryption are not enforced at the application layer in the base version; operate on a trusted LAN. The index server stores runtime metadata only; file contents are never stored on the server.

# 7 Validation and Performance Evaluation

## 7.0.1 Client start



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS D:\HK251\Computer Networks\ASSIGNMENT_1\BTL_1_MMT> python server.py
[SERVER] Listening on 0.0.0.0:5050
```

Figure 2: Terminal output when the client runs successfully

## 7.0.2 Client Running

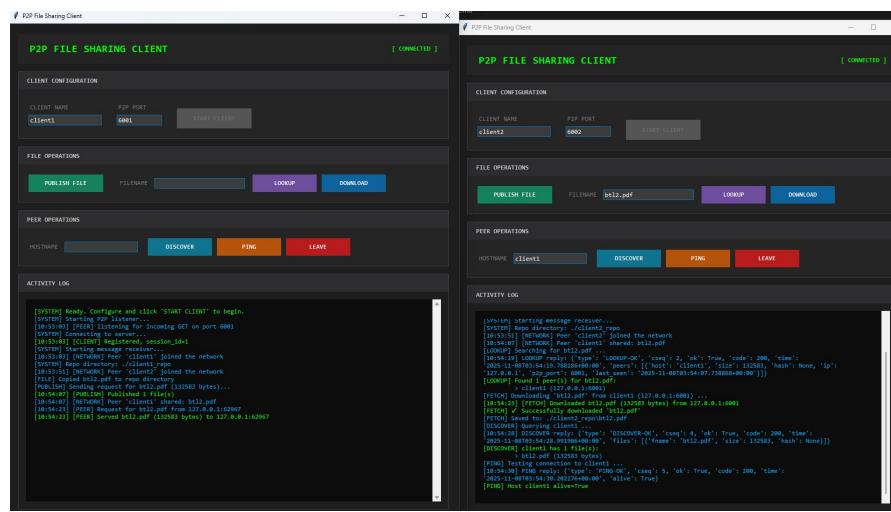


Figure 3: Client interface after startup: Client 1 (left) publishes file, Client 2 (right) downloads the file.



## 8 Extensions

**Compatibility.** All items below are *additive*; legacy clients/servers still interoperate. **Status:** unless noted, these are *design-only* in this submission.

### 8.1 Protocol Versioning & Feature Negotiation

- Add optional `proto:"p2p/1.0"` and `features: []` in REGISTER request; server reply may echo proto and advertise `features_supported: []`.
- Unknown fields MUST be ignored; unknown features are silently skipped.

**Example.**

```
{"type": "REGISTER", "cseq": 1,
  "host": {"name": "alice", "p2p_port": 6000, "agent": "p2p/1.0"},
  "proto": "p2p/1.0", "features": ["hash", "range"]}
\r\n
{"type": "REGISTER-OK", "cseq": 1, "ok": true, "code": 200,
  "session_id": 42, "ttl": 60, "time": "...", "proto": "p2p/1.0",
  "features_supported": ["hash", "range", "unpublish"]}
\r\n
```

### 8.2 Like-torrent Essential Components (Optional Metadata)

- **Metainfo** (JSON): compact descriptor for a file (or multi-file set).
- **Info**: structured fields for single/multi-file catalogs.
- **Piece Length**: default 512 KiB (demo: 128 KiB) to balance concurrency vs overhead.
- **Pieces**: SHA-256 per-piece digests (hex/base64, 32B each).
- **Name/Length/Files**: as typical—single or multi-file mode.

**Publish fields (optional).**

```
files[].hash = <sha256(full-file)>
files[].manifest = {"chunk_size": 524288, "hashes": [<sha256_0>, ...]}
```

### 8.3 Multi-direction / Multi-source Transferring (Swarming)

- Download the same object from **multiple peers in parallel** using disjoint byte ranges.
- Assembler writes pieces into a temp file, verifies per-piece hashes (if provided), then *atomically* finalizes to the repository (temp-rename).
- Scheduler prefers diverse sources; failed ranges are retried on different peers.



## 8.4 Range / Resume Downloads

- Data-plane verb `GETRANGE <fname> <start>-<end>` with reply `OK 206`, headers `Size` and `Content-Range`.
- Resume by scanning local file size (or piece bitmap) and requesting the tail or missing ranges.
- Verify total bytes and, if available, full or per-piece hashes before finalize.

Wire example.

```
# Downloader -> Uploader
GETRANGE movie.mp4 1048576-2097151\r\n\r\n

# Uploader -> Downloader
OK 206\r\n
Size: 10000000\r\n
Content-Range: bytes 1048576-2097151/10000000\r\n
\r\n
<---- 1048576 bytes ---->
```

## 8.5 Integrity Verification

- Optional at publish: `files[] .hash` and/or `files[] .manifest`.
- Uploader may send `Hash: <sha256>` in data-plane reply; downloader **SHOULD** verify before finalize.

## 8.6 Backpressure & Rate Limiting

- Control/data-plane overload signals: `code=429` with `Retry-After:<sec>` or `BUSY 503` (data-plane) then close.
- Client uses bounded exponential backoff; uploader enforces token-bucket or per-connection caps.

## 8.7 UNPUBLISH

- Withdraw previously advertised files without closing the session.

Message shape.

```
# Request
{"type":"UNPUBLISH","cseq":17,"session_id":42,
 "files":[{"fname":"a.txt"}, {"fname":"b.zip"}]}\r\n

# Reply
{"type":"UNPUBLISH-OK","cseq":17,"ok":true,"code":200,
 "removed":2,"time":"..."}\r\n
```



## 8.8 Peer Selection Heuristics

- Passive scoring by connect time/RTT and recent `last_seen`; prefer low-RTT, low-failure peers.
- Optional LOOKUP hints: `rtt_ms`, `capacity`; clients remain free to choose.

## 8.9 Client-side Caching & Auto Re-publish

- Cache successful LOOKUP(`fname`) with TTL (e.g., 30s) and negative cache for misses (e.g., 5s).
- After a successful fetch, optionally auto-PUBLISH the file to improve availability (opt-in).

## 8.10 Security (Stretch Goal)

- Control-plane over TLS (server certificate pinning); optional TLS for data-plane.
- Optional `token` field in REGISTER/PUBLISH/LOOKUP for authenticated deployments.

## 8.11 NAT Traversal (Stretch Goal)

- UPnP/NAT-PMP auto port mapping for `p2p_port`; STUN for reflexive IP discovery; TURN as fallback relay.
- No change to control-plane schema; `host.ip` may reflect public-reflexive address.

## 8.12 Atomic Finalize & Repository Policy

- Write to `<fname>.part` then `rename()` to `<fname>` on success; remove `.part` on error.
- Optional duplicate policy: `foo (1).ext`/hash-based naming to avoid collisions.

## 8.13 Diagnostics & Admin

- Optional STATS op (server): counters for sessions, publishes, lookups, active connections.
- Optional event types: "event": "PUBLISH" | "LEAVE" | "EXPIRE", ... already flow on persistent channels.

## 8.14 Error Map (Extended)

- **410 GONE**: resource/session no longer available.
- **429 TOO\_MANY\_REQUESTS**: apply `Retry-After`.
- **503 UNAVAILABLE**: temporary maintenance/overload.

## 9 Appendices

Repo: [https://github.com/Kun-05/BTL1\\_MMT\\_251](https://github.com/Kun-05/BTL1_MMT_251)



## 10 REFERENCES