Assignment 2: Peer-to-Peer File Synchronizer
Report (Test Cases & Results)

Zifan Si (Student #400265867)

February 18, 2026

GitHub link for this project: `https://github.com/ZifanSi/p2p-file-synchronizer`

1 Overview

This project implements a peer-to-peer (P2P) file synchronizer using a centralized tracker for peer discovery and directory aggregation. Each peer:

- establishes one persistent TCP connection to the tracker,
- advertises its working directory file metadata (name, integer mtime) on startup,
- sends periodic keepalive messages to maintain liveness (every 5 seconds),
- receives a directory response from the tracker after each Init/KeepAlive,
- downloads files that are missing locally or have older mtimes,
- serves file requests to other peers using a TCP protocol with a `Content-Length` header.

2 Environment

- OS: Windows (local testing)
- Python: 3.12.10
- Network: localhost (127.0.0.1)

3 Project Layout Used for Testing

```
Project/
  tracker.py
  run_all.bat
  Peer1/
    fileSynchronizer.py
    fileA.txt
  Peer2/
    fileSynchronizer.py
    fileB.txt
  Peer3/
    fileSynchronizer.py
    fileC.txt
```

4 How to Run

Tracker:

```
python tracker.py 127.0.0.1 9000
```

Peers (each started from its own folder):

```
cd Peer1 && python fileSynchronizer.py 127.0.0.1 9000
cd Peer2 && python fileSynchronizer.py 127.0.0.1 9000
cd Peer3 && python fileSynchronizer.py 127.0.0.1 9000
```

Optional batch script (starts tracker + 3 peers in separate Windows terminals):

```
@echo off
set IP=127.0.0.1
set PORT=9000

start "TRACKER" cmd /k "python tracker.py %IP% %PORT%"

timeout /t 1 >nul

start "PEER1" cmd /k "cd /d %~dp0Peer1 && python fileSynchronizer.py %IP% %PORT%"
start "PEER2" cmd /k "cd /d %~dp0Peer2 && python fileSynchronizer.py %IP% %PORT%"
start "PEER3" cmd /k "cd /d %~dp0Peer3 && python fileSynchronizer.py %IP% %PORT%"
```

5 Protocol Summary

5.1 Peer → Tracker

Messages are newline-terminated UTF-8 JSON objects.

- Init (sent exactly once on startup):

  ```
  {"port": <p>, "files": [{"name": "...", "mtime": <int>}, ...]}\n
  ```

- KeepAlive (sent every 5 seconds):

  ```
  {"port": <p>}\n
  ```

5.2 Tracker → Peer

For every Init/KeepAlive received, the tracker returns one directory response (newline-terminated JSON):

```
{"fileA.txt": {"ip":"...", "port":..., "mtime":...}, ...}\n
```

5.3 Peer ↔ Peer

Requester sends:

```
<filename>\n
```

Server responds:

```
Content-Length: <size>\n
<raw file bytes>
```

6 Selected Runtime Output Evidence

The following excerpts are copied from an actual run on localhost with three peers.

Tracker (excerpt)

```
Waiting for connections on port 9000
Client connected with 127.0.0.1:60411
Client connected with 127.0.0.1:60412
Client connected with 127.0.0.1:60413
client server127.0.0.1:8000
client server127.0.0.1:8001
client server127.0.0.1:8002
```

Peer startup + directory responses (excerpt)

```
Peer2:
Waiting for connections on port 8000
('connect to:127.0.0.1', 9000)
received from tracker: {"fileB.txt": {"ip": "127.0.0.1", "port": 8000, "mtime":
    1771289244}}

Peer1:
Waiting for connections on port 8001
('connect to:127.0.0.1', 9000)
received from tracker: {"fileB.txt": {"ip": "127.0.0.1", "port": 8000, "mtime":
    1771289244},
"fileA.txt": {"ip": "127.0.0.1", "port": 8001, "mtime": 1771289241},
"fileC.txt": {"ip": "127.0.0.1", "port": 8002, "mtime": 1771289247}}

Peer3:
Waiting for connections on port 8002
('connect to:127.0.0.1', 9000)
received from tracker: {"fileB.txt": {"ip": "127.0.0.1", "port": 8000, "mtime":
    1771289244},
"fileA.txt": {"ip": "127.0.0.1", "port": 8001, "mtime": 1771289241},
"fileC.txt": {"ip": "127.0.0.1", "port": 8002, "mtime": 1771289247}}
```

7 Test Cases and Results

This section documents test cases designed to validate all requirements in the protocol specification and grading rubric.

7.1 TC0 (Rubric: get_file_info, 1pt): Working-directory file filtering

Goal: Verify `get_file_info()` returns only valid files in the local working directory, ignores subdirectories and `.py/.dll/.so`, and reports integer mtimes.

Setup: In `Peer1/` create:

- `fileA.txt`
- `junk.dll`, `junk.so`, `temp.py`
- subfolder `sub/` containing `subfile.txt`

Steps:

1. Start Peer1 from `Peer1/`.
2. Observe tracker directory response content received by peers.

Expected:

- Only `fileA.txt` is advertised by Peer1 (no subfolder files, no `.py/.dll/.so`).
- Directory response contains integer `mtime` values.

Observed: Peer directory responses contained only `fileA.txt`, `fileB.txt`, `fileC.txt` (no filtered extensions or subfolder files). The returned `mtime` fields were integers, e.g.:

```
"fileA.txt": {"ip":"127.0.0.1","port":8001,"mtime":1771289241}
```

Result: PASS.

7.2 TC1 (Rubric: get_next_available_port, 1pt): Port selection and bind success

Goal: Verify each peer selects an available port and binds successfully (no collisions).

Steps:

1. Start tracker on `127.0.0.1:9000`.
2. Start three peers.

Expected: Each peer prints a distinct listening port and does not print a bind error.

Observed:

```
Peer2: Waiting for connections on port 8000
Peer1: Waiting for connections on port 8001
Peer3: Waiting for connections on port 8002
```

Result: PASS.

7.3 TC2 (Rubric: FileSynchronizer initializer, 1pt): Tracker connection and persistent communication

Goal: Verify the initializer correctly sets up sockets, connects to tracker, advertises port/files (Init), and continues keepalive cycles.

Steps:

1. Start tracker.
2. Start Peer1/Peer2/Peer3.
3. Observe tracker accepts 3 client connections and prints peer addresses.

4. Observe peers repeatedly print directory responses across time (keepalive cycles).

Expected:

- Tracker logs 3 client connections.
- Tracker lists peer servers `127.0.0.1:8000/8001/8002`.
- Peers receive directory response after Init and subsequent keepalives.

Observed: Tracker output shows 3 connections and peer servers; peers repeatedly receive directory JSON:

```
Client connected with 127.0.0.1:60411
Client connected with 127.0.0.1:60412
Client connected with 127.0.0.1:60413
client server127.0.0.1:8000
client server127.0.0.1:8001
client server127.0.0.1:8002
```

Result: PASS.

7.4 TC3 (Rubric: sync discovery/retrieve, 1pt): Convergence (missing file download)

Goal: Verify peers download missing files and converge so all peers have the same file set.
Setup: Initial state:

- Peer1 has only `fileA.txt`
- Peer2 has only `fileB.txt`
- Peer3 has only `fileC.txt`

Steps:

1. Start tracker and peers.
2. Wait for 1–2 sync cycles (a few seconds).
3. Check each peer folder contents.

Expected: Each peer downloads the other missing files and ends with A/B/C.
Observed: After running, each peer folder contained:

```
Peer1: fileA.txt fileB.txt fileC.txt
Peer2: fileA.txt fileB.txt fileC.txt
Peer3: fileA.txt fileB.txt fileC.txt
```

Directory responses remained stable across cycles, listing all three files with metadata.
Result: PASS.

7.5 TC4 (Rubric: sync overwrite newer, 1pt): Newer mtime version propagates

Goal: Verify a newer file version (larger mtime) overwrites older copies on other peers.
Important assumption: The specification states file contents do not change after a peer starts execution. Therefore, this test is executed by stopping peers, editing a file, then restarting.
Steps:

1. Stop all peers.

2. Edit `Peer2/fileB.txt` (append a line) and save (mtime increases).
3. Restart tracker (if needed) and restart peers.
4. Wait 1–2 sync cycles.
5. Compare `Peer1/fileB.txt` and `Peer3/fileB.txt` with Peer2.

Expected: Peer1 and Peer3 download the newer `fileB.txt`. Their `mtime` is set to the advertised value via `os.utime()`.

Observed: Updated `fileB.txt` propagated to other peers; contents matched Peer2 after synchronization.

Result: PASS.

7.6 TC5 (Rubric: process_message, 2pt): Peer-to-peer file serving with Content-Length

Goal: Verify the serving peer responds with the correct `Content-Length` header and sends exactly that many bytes.

Steps:

1. Ensure Peer3 has `fileC.txt`.
2. Delete `Peer1/fileC.txt`.
3. Wait for sync so Peer1 fetches `fileC.txt` from the peer listed in the directory response (Peer3).
4. Verify downloaded file size and content matches the source file in Peer3.

Expected:

- Peer1 receives a response starting with `Content-Length: <size>`.
- Peer1 reads exactly `<size>` bytes and writes the file in binary mode.
- Downloaded file matches source (size and content).

Observed: Peer1 successfully re-downloaded `fileC.txt`. The downloaded file matched the source content and size, consistent with correct `Content-Length` framing and exact byte transfer.

Result: PASS.

7.7 TC6 (Rubric: failure handling, 1pt): Timeout/failure and discard partial files

Goal: Verify failed transfers are handled gracefully (timeouts/failures do not leave partial files) and synchronization continues.

Steps:

1. Start tracker and all peers; confirm all peers have A/B/C.
2. Stop Peer3 (simulate peer failure).
3. Delete `Peer1/fileC.txt`.
4. Wait 1–2 sync cycles.
5. Check `Peer1/` for leftover partial file (e.g., `fileC.txt.part`).
6. Restart Peer3 and wait for another sync cycle.

Expected:

- When Peer3 is down, Peer1 cannot download `fileC.txt`.

- No partial file remains after a failed transfer.
- After Peer3 restarts, Peer1 downloads `fileC.txt` successfully.

Observed: With Peer3 stopped, `fileC.txt` was not retrieved and no `.part` file remained. After restarting Peer3, Peer1 downloaded `fileC.txt` successfully.

Result: PASS.

8 Results Summary

Rubric Requirement / Test Case	Result
TC0: `get_file_info()` filtering rules (1pt)	PASS
TC1: `get_next_available_port()` (1pt)	PASS
TC2: FileSynchronizer initializer (1pt)	PASS
TC5: `process_message()` + Content-Length (2pt)	PASS
TC3: `sync()` discovery + retrieve missing files (1pt)	PASS
TC4: `sync()` overwrite newer mtime (1pt)	PASS
TC6: Timeouts/failures + discard partial files (1pt)	PASS

9 Conclusion

The implementation was validated against the protocol specification and grading rubric using custom test cases. The tests demonstrated correct working-directory file discovery and filtering, dynamic port selection, persistent tracker communication with periodic keepalives, peer-to-peer file serving using a `Content-Length` header, convergence to a consistent file set across peers, propagation of newer file versions via modification time, and correct handling of peer failures without leaving partial files.