P2P File Synchronizer — Report (Test Cases & Results)

Zifan Si (Student #400265867)

February 18, 2026

1 Overview

This assignment implements a peer-to-peer file synchronizer using a centralized tracker. Each peer:

- discovers peers and file metadata from the tracker,
- downloads missing or newer files from other peers (by mtime),
- serves file requests from peers using a simple TCP protocol with a `Content-Length` header.

2 Environment

- OS: Windows (local testing)
- Python: 3.x
- Network: localhost (127.0.0.1)

3 Project Layout Used for Testing

```
Project/
  tracker.py
  run_all.bat
  fileSynchronizer.py
  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:

```
run_all.bat
```

5 Protocol Summary

5.1 Peer → Tracker

Messages are newline-terminated JSON.

- Init (once): {"port": <p>, "files": [{"name":..., "mtime":...}, ...]}
- KeepAlive (periodic): {"port": <p>}

5.2 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, and are referenced in the test cases below.

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

7.1 TC0: get_file_info() Filtering Rules

Goal: Verify only valid files in the local directory are included, and filtering matches the rules.

Setup: In `Peer1/` create:

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

Steps:

1. Start Peer1 and observe what it advertises (via tracker directory response content).

Expected:

- Only `fileA.txt` is included.
- No subfolder files appear.
- `mtime` values are integers.

Observed: Only `fileA.txt` appeared in the directory listing; ignored `.py/.dll/.so` and subfolder.

Result: PASS.

7.2 TC1: get_next_available_port() / Bind Success

Goal: Verify each peer binds to an available port (no collisions).

Steps:

1. Start tracker on `127.0.0.1:9000`.
2. Start Peer1, Peer2, Peer3.

Expected: Each peer prints `Waiting for connections on port <p>` with distinct ports.

Observed: From runtime output:

```
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: Tracker Registration & Directory Aggregation

Goal: Verify peers register to the tracker and the tracker directory contains all files.

Steps:

1. Initial state: Peer1 has `fileA.txt`, Peer2 has `fileB.txt`, Peer3 has `fileC.txt`.
2. Start tracker and peers.

Expected: Tracker accepts connections and peers receive a directory JSON containing A/B/C with (ip, port, mtime).

Observed: From tracker output:

```
Client connected with 127.0.0.1:60411
Client connected with 127.0.0.1:60412
Client connected with 127.0.0.1:60413
```

And from peer output (directory contains all three files with metadata):

```
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}}
```

Result: PASS.

7.4 TC3: Missing File Download (Convergence)

Goal: Verify peers download missing files and all peers converge to the same set.

Steps:

1. Ensure Peer1 has only `fileA.txt`, Peer2 only `fileB.txt`, Peer3 only `fileC.txt`.
2. Start tracker and peers; wait for 1–2 sync cycles.

Expected: Each peer downloads the missing files and ends with A/B/C.

Observed: After running, directory listings showed 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
```

Also, the repeated directory responses showing all three files remained stable across sync cycles.

Result: PASS.

7.5 TC4: Newer Version Wins (mtime Update)

Goal: Verify a newer file version propagates based on modification time.

Steps:

1. Edit `Peer2/fileB.txt` (append a line) and save.
2. Wait 1–2 sync cycles.
3. Compare `Peer1/fileB.txt` and `Peer3/fileB.txt` content to Peer2.

Expected: Peer1 and Peer3 fetch the updated `fileB.txt` and match Peer2. File `mtime` becomes the newer value.

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

Result: PASS.

7.6 TC5: Peer Serving Protocol (Content-Length Correctness)

Goal: Verify file transfers use the correct `Content-Length` and exact bytes.

Steps:

1. Delete `Peer1/fileC.txt`.
2. Wait for sync so Peer1 fetches `fileC.txt` from Peer3.
3. Verify the downloaded file size matches the source and the content is identical.

Expected: Peer1 receives `Content-Length: <size>` and writes exactly that many bytes.

Observed: Downloaded `fileC.txt` matched the source content and size.

Result: PASS.

7.7 TC6: Failure Handling (Peer Down / Discard Partial)

Goal: Verify that failed downloads do not leave partial files and synchronization continues.

Steps:

1. Start tracker and all peers; confirm all have A/B/C.
2. Stop Peer3 process (simulate a peer crash).
3. Delete `Peer1/fileC.txt`.
4. Wait 1–2 sync cycles.
5. Check Peer1 directory for any leftover `fileC.txt.part`.
6. Restart Peer3 and wait for sync again.

Expected:

- When Peer3 is down, Peer1 cannot download `fileC.txt`.
- No partial file (`.part`) remains after a failed transfer.
- After Peer3 restarts, Peer1 successfully downloads `fileC.txt`.

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

Test Case	Result
TC0: get_file_info() filtering	PASS
TC1: port selection / bind	PASS
TC2: tracker registration + directory	PASS
TC3: missing file download	PASS
TC4: mtime update propagation	PASS
TC5: Content-Length correctness	PASS
TC6: failure handling / discard partial	PASS

9 Conclusion

The implementation was validated with tests covering file filtering, port selection, tracker interaction, peer file serving, synchronization convergence, update propagation by mtime, and basic failure handling.