Assignment 2: Peer-to-Peer File Synchronizer

Rong Zheng

January 2026

1 Introduction

In this programming assignment, you will develop a simple peer-to-peer (P2P) file sharing application that synchronizes files among peers. It has the flavor of Dropbox but instead of storing and serving files using the cloud, we utilize a peer-to-peer approach. Recall that the key difference between the client-server architecture and the peer-to-peer architecture for distributed applications is that in the latter, the end host acts both as a server (e.g., to serve file transfer requests) and as a client (e.g., to request a file). One challenge in peer-to-peer applications is that peers do not initially know each other's presence and IP addresses. As such, a server, called `tracker` in this project, is needed to facilitate the "discovery" of peers. **Binary codes of the tracker are provided to you for testing purposes**. In this assignment, you only need to develop the file synchronizer program that runs on the peers.

Figure 1 illustrates the architecture of our P2P file sharing application. The tracker runs in a server (bottom), while each peer (top row) runs a file synchronizer. In the example, "Tracker" will track live peer nodes and store the meta info of files on 3 peers. Initially, each peer node each has one local file (fileA.txt, fileB.txt, fileC.txt). After they connect to the tracker and synchronize with other peers, each peer will eventually have all three files locally. In this assignment, "client", "peer" and "synchronizer" all mean the same thing and are used interchangeably.

After unzip the file for Assignment 2 on Avenue, you will find

- `instructions.pdf` – this document.
- `Skeleton.py` – the skeleton code for the file synchronizer you will develop (search for YOUR CODE for code segments to fill). Rename to fileSynchronizer.py
- `tracker.py` – tracker code for testing

2 Protocol Specification

This section formally specifies the communication protocol used by the peer-to-peer (P2P) file synchronizer system. The protocol consists of two components:

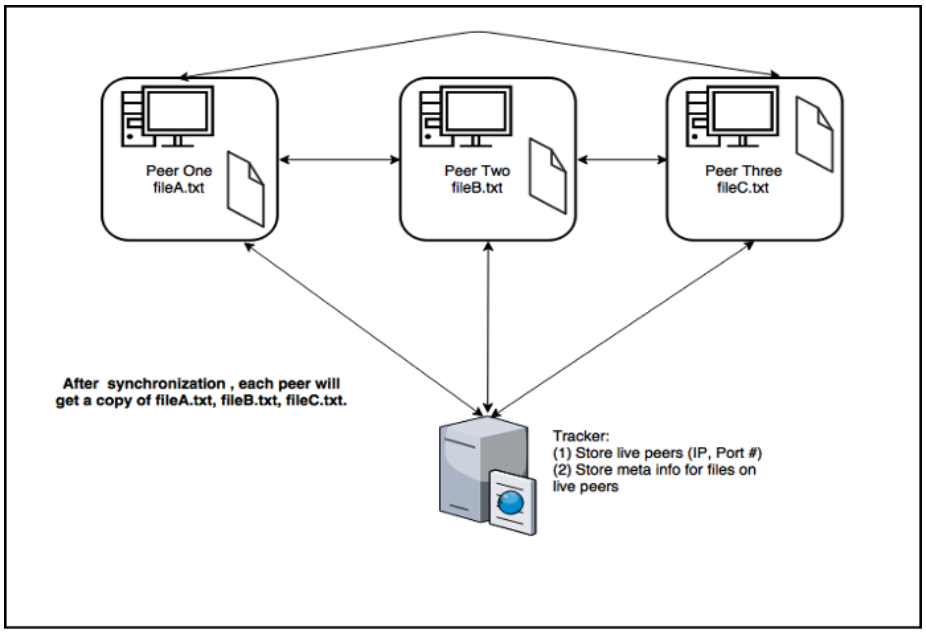

Figure 1: Architecture of the P2P file sharing application.

- Peer-to-Tracker protocol (directory and liveness management)
- Peer-to-Peer protocol (file transfer)

Unless otherwise stated, all communication uses TCP sockets.

2.1 Terminology

- **Peer**: A file synchronizer instance. The terms *peer*, *client*, and *synchronizer* are used interchangeably.
- **Tracker**: A centralized server responsible for peer discovery and directory maintenance.
- **Working directory**: The directory from which a peer advertises and serves files.

2.2 General Assumptions

- All peers and the tracker communicate over reliable TCP connections.
- System clocks across hosts are sufficiently synchronized for file modification times to be meaningfully compared.
- Only files in the working directory are considered; subdirectories are ignored.

- File contents do not change after a peer starts execution.

2.3 Peer-to-Tracker Protocol

2.3.1 Connection Model

Each peer establishes exactly **one persistent TCP connection** to the tracker. This connection is reused for all subsequent communication with the tracker.

2.3.2 Message Encoding and Framing

- All messages exchanged between a peer and the tracker are encoded in UTF-8 JSON.
- Each JSON message is sent as a **single JSON object terminated by a newline character (`\n`)**.
- The receiver reads from the TCP stream until a complete JSON object (ending in `\n`) is received.

2.3.3 Initial Message

Upon startup, a peer MUST send exactly one **Initial Message** to the tracker.

Format

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

Semantics

- `port` is the TCP port number on which the peer listens for incoming file requests.
- `files` is a list of files in the peer's working directory.
- `mtime` is the file's last modified time, rounded down to an integer (seconds since epoch).

2.3.4 Keepalive Message

After sending the Initial Message, the peer MUST send a **Keepalive Message** to the tracker every 5 seconds.

Format

```
{
  "port": <int>
}
```

Semantics The Keepalive Message informs the tracker that the peer is still active. No file metadata is included.

2.3.5 Directory Response Message

For every Initial Message and every Keepalive Message received, the tracker sends exactly one **Directory Response Message** to the peer.

Format

```
{
  "<filename>": {
      "ip": <string>,
      "port": <int>,
      "mtime": <int>
  },
  ...
}
```

Semantics

- Each key corresponds to a unique filename.
- For each filename, the tracker returns information for **at most one peer**.
- If multiple peers advertise the same filename, the tracker selects the peer with the largest (lastest) `mtime`.
- The directory represents the tracker's current global view of the system.

2.4 Peer Directory Processing Rules

Upon receiving a Directory Response Message, the peer MUST:

1. Compare each listed file against its local copy (if any).
2. Identify files that are either:
   - not present locally, or
   - present but have an older modification time locally.
3. Request each such file exactly once from the respective peer using the IP and port number information contained in the Directory Response Message.

2.5 Peer-to-Peer File Transfer Protocol

2.5.1 Connection Model

- A peer listens on its advertised file-serving port for incoming TCP connections.
- Each file request is handled using a separate TCP connection.
- Connections are closed immediately after the requested file has been sent.

Peers MAY handle incoming file requests sequentially or concurrently.

2.5.2 File Request Message

Format The File Request Message consists of the filename encoded in UTF-8 and terminated by a single newline character (`\n`).

Semantics The requesting peer opens a TCP connection to the serving peer and sends exactly one filename. No additional data is sent on this connection by the requester.

2.5.3 File Response Message

Format The File Response Message consists of:

1. A text header line specifying the file size in bytes:

   ```
   Content-Length: <size>\n
   ```

2. Immediately followed by exactly `<size>` bytes of file content. File contents are transferred as raw bytes. Peers MUST open files in binary mode for reading and writing (e.g., `"rb"` and `"wb"`). No character encoding is applied to file contents.

Semantics

- `<size>` is a non-negative integer specifying the number of bytes in the file.
- File contents are transferred as raw bytes; peers MUST read and write files in binary mode.
- After sending the file content, the serving peer closes the TCP connection.
- The receiving peer reads exactly `<size>` bytes after parsing the header.

2.5.4 File Storage Rules

- If fewer than `<size>` bytes are received, the transfer is considered failed and the partially received file MUST be discarded.
- Upon successful receipt of all `<size>` bytes, the file is written to the working directory.
- The local modification time is set to the value specified in the Directory Response Message using `os.utime()`.

If a connection times out or terminates prematurely, the partially received file MUST be discarded.

2.6 Timeouts and Failures

- Peers SHOULD set reasonable socket timeouts when connecting to other peers.
- If a peer fails to respond to a file request, the requesting peer skips that file and proceeds to the next file to be retrieved.
- If the connection to the tracker fails, the peer MAY terminate execution.

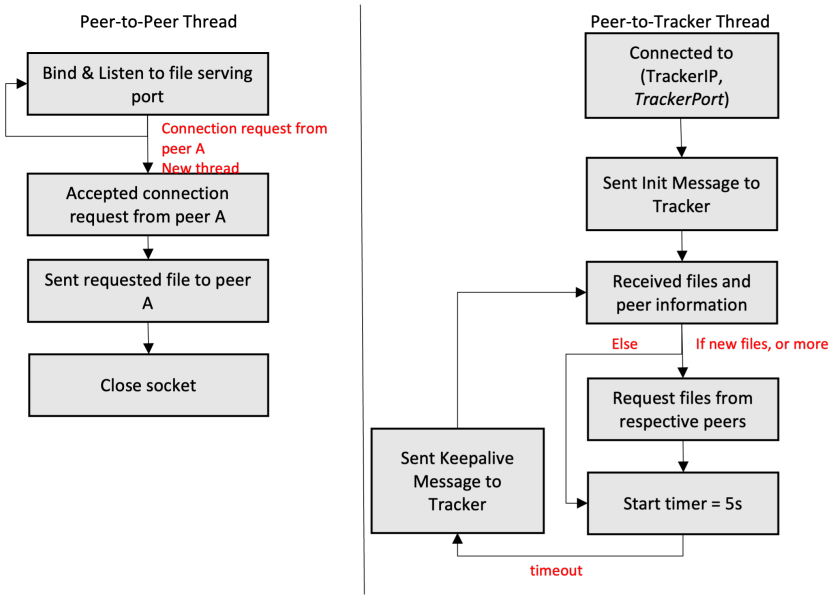

Figure 2: State Diagram of the file synchronizer

Figure 2 summarizes the workflow of a file synchronizer.

3 Tips for Implementation and Testing

- Your implementation should be based on Python 3 for compatibility
- Create seperate test folders to run your fileSynchronizer.py
- Use 'ctrl' + '\' to terminate a running tracker or peer.
- You may test your program on the same host with different port numbers or different hosts.

4 Rubrics and Submission

You are expected to come up with your own test cases to verify that your implementation satisfies the requirements. For grading, the following rubric will be followed:

- (1pt) `get_file_info()` – get file info in the local directory
- (1pt) `get_next_available_port()` – returns the next available port
- (1pt) completion of the initializer of FileSynchronizer class
- (2pt) `process_messge()` – return the requested file with proper header to a peer
- (1pt) `sync()` – discovery and retrieve new files from other peers
- (1pt) `sync()` – overwrite a local file if a new version exists on other peers
- (1pt) handling tracker and peer timeout and failure
- (2pt) document test cases and results

For submission,

1. Your code should be named `fileSynchronizer.py`.
2. Document your test cases and results in `report.pdf`.
3. Upload `report.pdf` and `fileSynchronizer.py` to Avenue.