Programming Assignment 1: Peer-to-Peer (P2P) File Synchronization (Part I)

Due Feb. 16th, 11:59pm (by timestamps in gitlab)

In this and next programming assignments, you will develop a simple P2P file sharing application that synchronize 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 is that in the later, 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. As such, a server node, called *FileTracker* in this project, is needed to facilitate the "discovery" of peers. In Part I, you will develop the server program for *FileTracker*. In Part II, you will develop the *FileSynchronizer* program for the peers.

In the project folder, you will find

- *FileSynchronizer* and *Tracker* binary executables. Different versions have been compiled for different OS. This is used to understand how the application works and to test your tracker program. (If you cannot any suitable for your platform, please contact the TAs).
- Skeleton code for FileTracker.

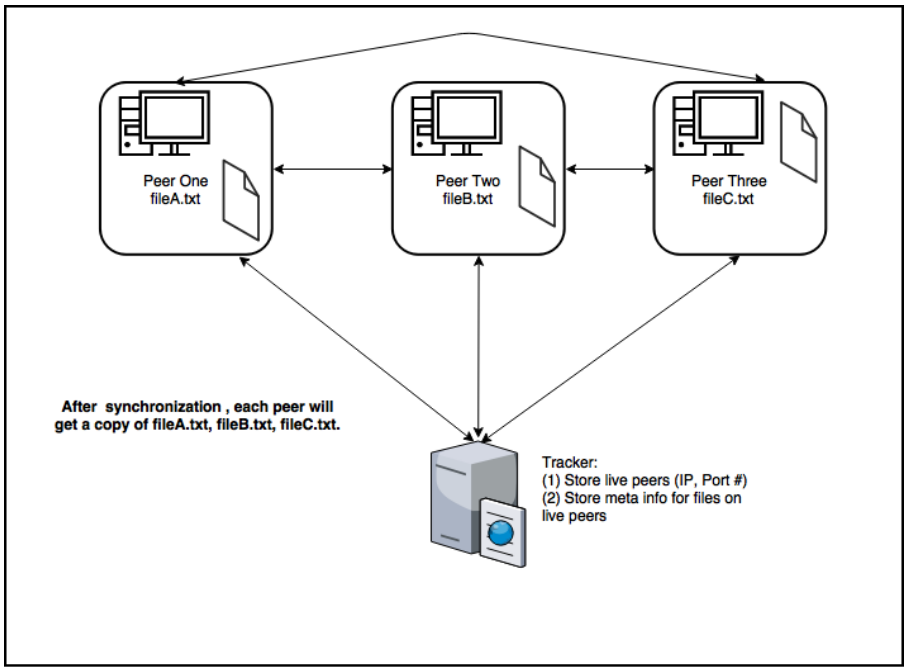

Figure 1 Application Architecture

The figure above illustrates the architecture of the application. In the example, "Tracker" will trace the live peer nodes and store the meta info of files on 3 peers. In this example, 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 three files locally.

Specifications:

- Tracker keeps track of live FileSynchronizer peers and maintains a directory of files, IP addresses, port numbers of the peer with the file. The tracker server's IP address (*TrackerIP*) and port number (*TrackerPort*) shall be specified as command line arguments. When contacted by *FileSynchronizer* peers, the tracker sends with the directory using a TCP socket. (note the tracker does not store the content of the files. In the skeleton code, you can find 'self.users' and 'self.files' field for these purposes.)
- The tracker listens to its specified port. Upon accepting a new connection (from a user), a new thread is spawned. The file information on the peer node will be uploaded to the tracker, along with the IP address of the peer and *FileSynchronizerServerPort*, the port for serving file transfer requests to other peers.
- The peer node periodically sends a keepalive message to the tracker to inform the tracker that it is online. When the tracker receives the message, it will refresh the state of the peer and also send the directory info to the peer. Otherwise, if the tracker does not get any keepalive message from the peer node for 180 seconds, it will remove entries of the files associated with the peer.
- FileSynchronizer communicates with the tracker through a single TCP socket to
    1. send an *Init message* containing names and modified time of its local files as well as *FileSynchronizerServerPort* when it first starts
    2. send a *keepalive* message containing *FileSynchronizerServerPort* every 5 seconds to the tracker
    3. receive the directory information maintained by the tracker

The state diagram of the tracker server is given in Figure 1. (Application-layer) Messages exchanged between Tracker and FileSynchronizer are encoded in *JSON* specified in Table 1.

Table 1 Message Format

Message Type	Purpose	Action	Message Format (key, value)
***Init* Message**	From FileSynchronizer to Tracker. • Upload meta file information and *FileSynchronizerServer Port* • Request directory	Tracker responds with a directory response message	{'port':int, 'files':[{'name':string,'mtime':long}]}
KeepAlive Messsage	From FileSynchronizer to Tracker. Inform the tracker that the peer is still alive.	1. Tracker refreshes its timer for the respective peer 2. Tracker responds with a directory response message	{'port':int}
Directory Response Message	From Tracker to FileSynchronizer. Return the directory at the tracker.	Peer retrieve new or modified files from other peers	{filename:{'ip':,'port':,'mtime':}

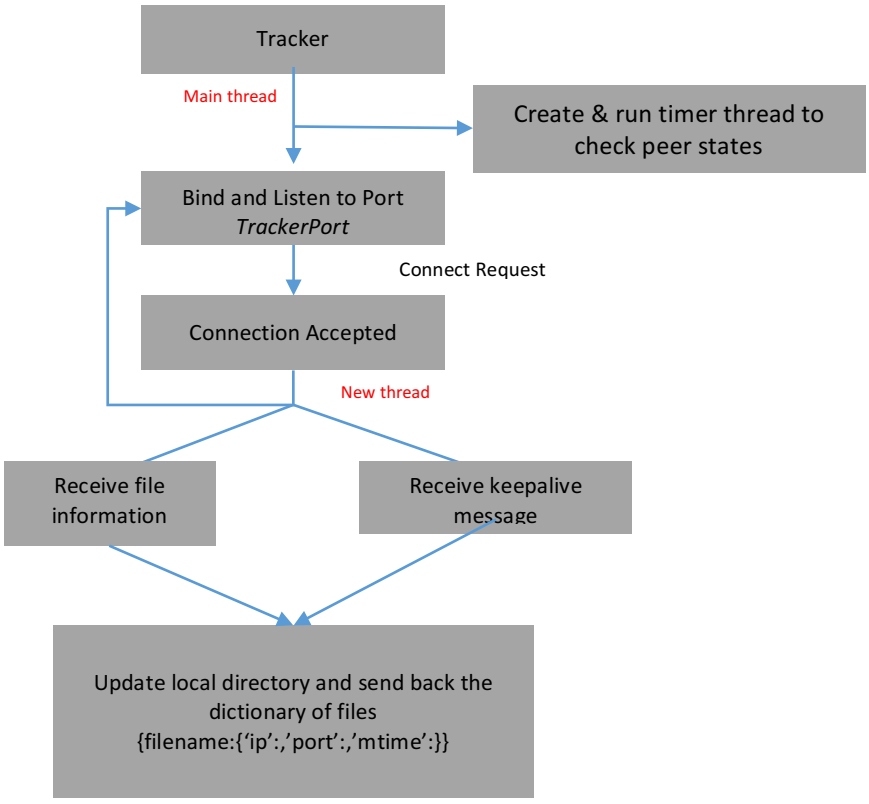

Figure 2 State Diagram of Tracker

For simplicity, 1) we only consider files in the same where FileSynchronizer runs (i.e., not subfolders); 2) we do not consider changes *after* FileSynchronizer starts such as file deletion, modification, and addition – in other words, Init message that contains local file information is sent only once; 3) if multiple files from different peers have the same file name, the one with the *latest* modified timestamp will be kept by the tracker; and 4) only one TCP connection is needed between a peer and a tracker.

Your task for this lab is to complete the implementation of the tracker.

Instructions:

1. Your implementation should be based on Python 2.7 for compatibility
2. In multithread programming, pay attention to synchronization issues. Codes that modify common data structures should be accessed by at most one thread at a time (Hint: use self.lock)
3. It is a good programming practice to perform unit testing for each step of your development when key new functionality is implemented. You may test your program on the same host with different port numbers or different hosts.

Submission:

1. Submit your source codes in a SINGLE file called 'tracker.py' to 'PA1/src' folder in your gitlab project.

Appendix: Steps to test your implementation for the Tracker server.

As an example (on Linux machine), the steps below provide guide to test your implementation.

(1) First, start the tracker code and specify the IP address and listening port.

```
a@ubuntu:~/4c03/assignment3$ python tracker.py 127.0.0.1 8080
Waiting for connections on port 8080
```

(2) Prepare two test folders, e.g.,"test1" and "test2", and unzip the fileSynchronizer to these folders. Add "test_file.txt" in "test1" folder

(3) In folder "test1", we run the FileSynchronizer to connect to the tracker. The FileSynchronizer will also start a listening thread for serving file request.

```
a@ubuntu:~/4c03/assignment3/test1$ ./fileSynchronizer 127.0.0.1 8080
Waiting for connections on port 8001
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
connect to:127.0.0.1 8080
```

(4) Do the same in folder "test2"

```
a@ubuntu:~/4c03/assignment3/test2$ ./fileSynchronizer 127.0.0.1 8080
Waiting for connections on port 8002
connect to:127.0.0.1 8080
test_file.txt
Receiving...
Receiving...
Receiving...
Receiving...
Receiving...
Receiving...
```

After the connection of "test2", you will find "f1.txt", which is the result of synchronization from "test1"

(5) You can print the received messages, IP address and port number for debugging

```
a@ubuntu:~/4c03/assignment3$ python tracker.py 127.0.0.1 8080
Waiting for connections on port 8080
Client connected with 127.0.0.1:40302
client server127.0.0.1:8001
client server127.0.0.1:8001
client server127.0.0.1:8001
client server127.0.0.1:8001
client server127.0.0.1:8001
client server127.0.0.1:8001
client server127.0.0.1:8001
Client connected with 127.0.0.1:40304
client server127.0.0.1:8001
client server127.0.0.1:8002
client server127.0.0.1:8001
client server127.0.0.1:8002
client server127.0.0.1:8001
client server127.0.0.1:8002
client server127.0.0.1:8001
client server127.0.0.1:8002
client server127.0.0.1:8001
client server127.0.0.1:8002
client server127.0.0.1:8001
client server127.0.0.1:8002
```