Programming Assignment 2: P2P File Sharing Tracker

Due date: March 15th 11:59pm (by timestamps in gitlab)

In this and the next programming assignments, you will develop a simple 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 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 tracker in this project, is needed to facilitate the "discovery" of peers. In this assignment, **you will develop such a tracker**.

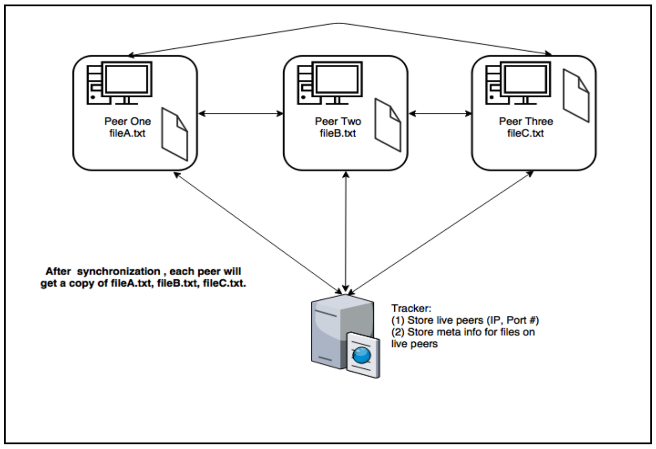

Peer One
fileA.txt
Peer Two
fileB.txt
Peer Three
fileC.txt
After synchronization , each peer will get a copy of fileA.txt, fileB.txt, fileC.txt.
Tracker:
(1) Store live peers (IP, Port #)
(2) Store meta info for files on live peers

Figure 1 P2P file sharing architecture

The figure above illustrates the architecture of the application. In the example, "Tracker" traces the live peer nodes and store only the meta info of files (NOT THE ACTUAL FILES!) on 3 peers. In this example, initially, each peer node 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 the PA2 folder, you will find:

- ✓ "instructions.pdf" - this document.
- ✓ "Skeleton.py" - the skeleton code for the **tracker** you will develop.

Note: As shown in Fig.1, the tracker is supposed to run in a centralized server, while each client will run a file synchronizer. You will develop the file synchronizer in the next assignment. However, you will need clients to test your implementation of the tracker. Therefore, we provide binary executables for testing purposes. You can download binary executables of the file synchronizer for 64-bit Windows, Mac and Linux. https://www.dropbox.com/sh/cxzpib967ajdf1p/AACR79m5AyTonpy2qSlCdStla?dl=0. If none of these binary files works for you, you can test your code using the 64-bit Linux version on the departmental Linux machine mills.cas.mcmaster.ca.

Specifications:

- The tracker keeps track of live peers and maintains a directory of files and their corresponding peers' IP addresses, port numbers. To start the tracker, the IP address and port number shall be specified as command line arguments. When contacted by clients (peers), the tracker sends the files information using a TCP socket. The tracker does not store the content of the files, while only stores the meta information. In the skeleton code, you can find 'self.users' and 'self.files' field that serve such purposes.

- The tracker listens to its specified port. Upon accepting a new connection (from a peer), a new thread is spawned to communicate with the peer. The **meta information of files** on the peer node will be first **uploaded to the tracker**, along with the **IP address** and the **port for serving file requests on the peer**.
- 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 information back** 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.
- Peers communicates with the tracker through a single TCP socket to
    1. send an **initial message** containing **names** and **modified time** of its local files as well as **file request serving port** when it first starts.
    2. send a **keepalive** message containing **file request serving port** 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 2 and messages exchanged between tracker and peers are encoded in **JSON** specified in Table 1.

Table 1 Message Format

Message Type	Purpose	Message Format (key, value)	Actions Followed by the Message

Initial Message	From **peers to the tracker**:
	(1) Upload file information and file serving port.
	(2) Request directory information stored in the tracker.
		{‘port’:int, ’files’: [{‘name’:string,’mtime’:long}]}
		
		e.g.,
		{"port": 8001, "files": [{"mtime": 1548878750.8774116, "name": "fileA.txt"}]}
			Tracker responds with a directory response message (See the **third row for the format**)

Keep-alive Message	From **peers to the tracker**:
	(1) inform the tracker that the peer is still alive.
		{‘port’:int}
		
		e.g.,
		{"port": 8001}
			(1) The tracker refreshes its timer for the respective peer
			(2) The tracker responds with a directory response message (**third row** for format)

Directory Response Message	From **the tracker to peers**: (1) Return the directory at the tracker.	{filename:{‘ip’:,’port’:,’mtime’:}	Peer retrieve new or modified files from other peers

		e.g., {"fileB.txt": {"ip": "127.0.0.1", "mtime": 1548878750.8776329, "port": 8002}, "fileA.txt": {"ip": "127.0.0.1", "mtime": 1548878750.8774116, "port": 8001}}	

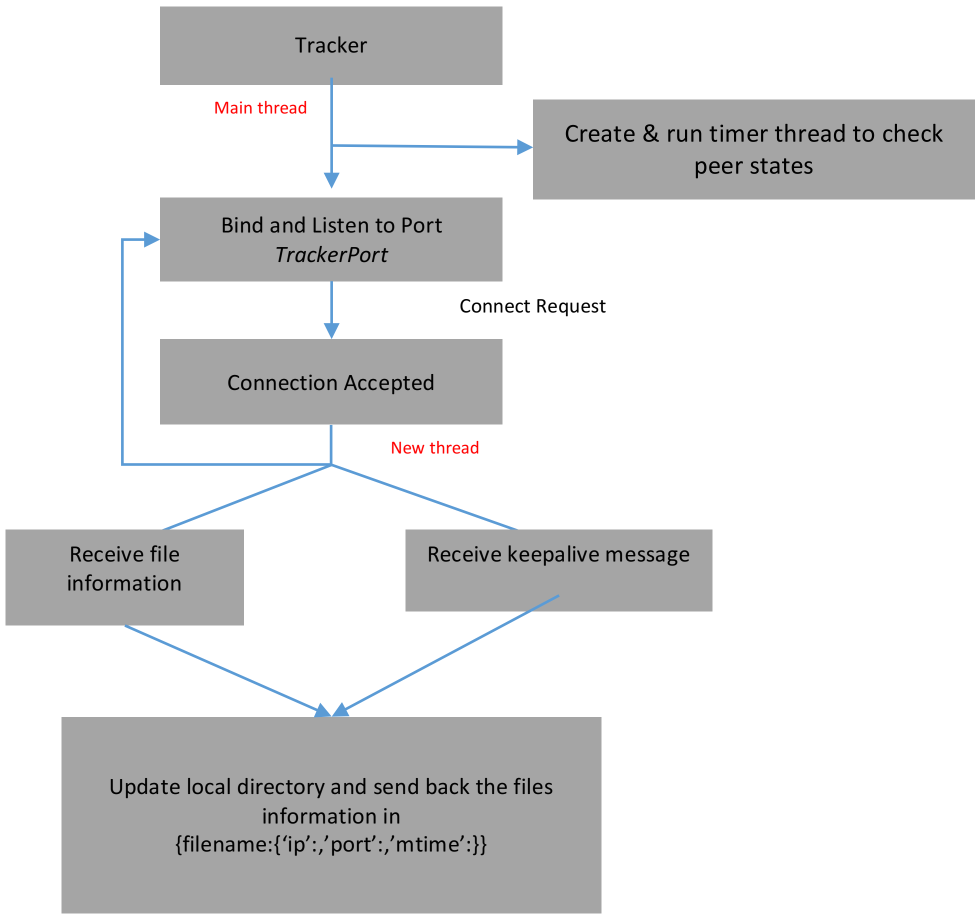

Tracker
Main thread
Create & run timer thread to check peer states
Bind and Listen to Port *TrackerPort*
Connect Request
Connection Accepted
New thread
Receive file information
Receive keepalive message
Update local directory and send back the files information in {filename:{'ip':,'port':,'mtime':}}

Figure 2 State Diagram of the tracker

For simplicity,
1) We only consider files within the same folder where the peer or synchronizer runs (**no subfolders**).
2) We do not consider changes **after** peers start, such as file deletion, modification, and addition. In other words, **the (initial) 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.
4) Only one TCP connection is needed between a peer and a tracker.

Implementation Steps:

1. Read this document carefully, especially the specifications part before attempting coding. This document contains almost all the information needed.
2. Download a suitable binary executable (fileSynchronizer) for your platform from https://www.dropbox.com/sh/cxzpib967ajdf1p/AACR79m5AyTonpy2qSlCdStla?dl=0 for testing.
3. Copy Skeleton.py to 'tracker.py' and go over the skeleton to be familiar with the data structures used
4. Start coding and testing.
5. If you have any question, please post on piazza or go to lab sessions.

Note:

1. Your implementation should be based on Python 3 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:

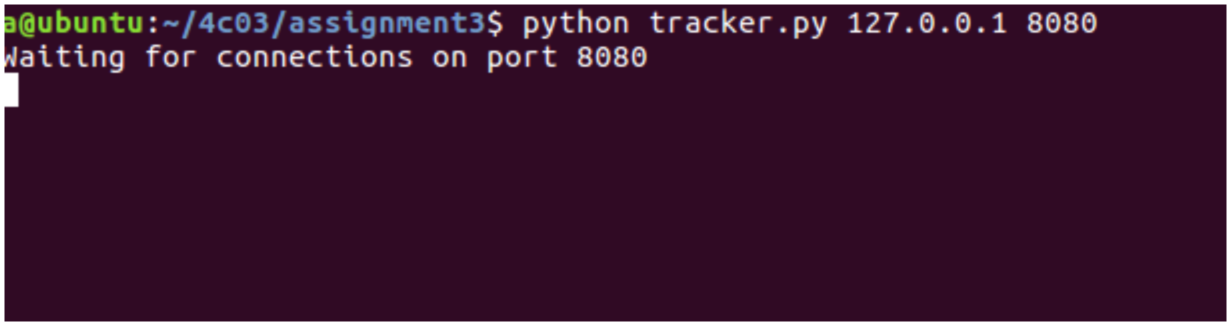
1. Submit your source codes in a SINGLE file called 'tracker.py' to 'PA2/tracker.py' folder in your gitlab project. (DO NOT USE OTHER NAMES!)
2. Document your test cases or demos in 'PA2/report.pdf' in gitlab.

Appendix I: Steps to test your implementation for the tracker.

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

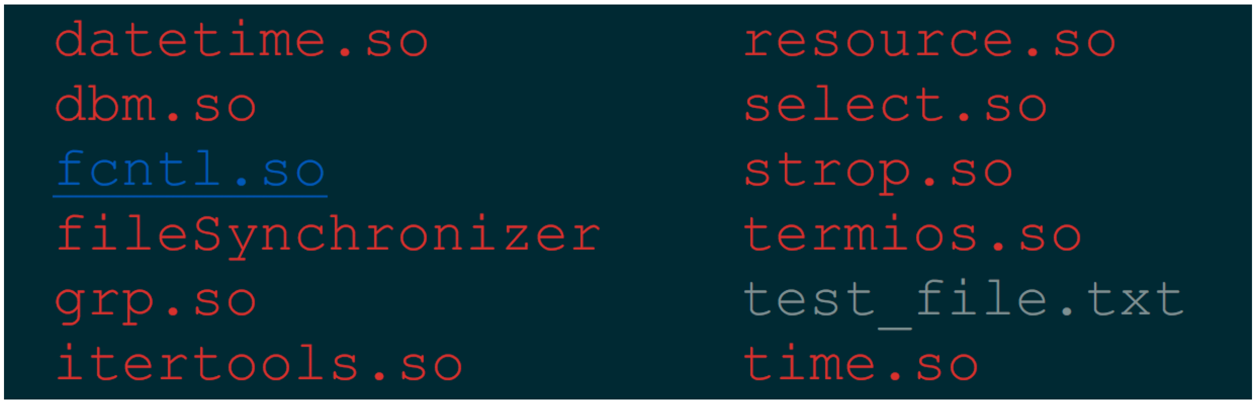
Suppose the working directory is currently under ~/4c03/assigment3:

(1) First, start the tracker you developed based on the skeleton and specify the IP and listening port.



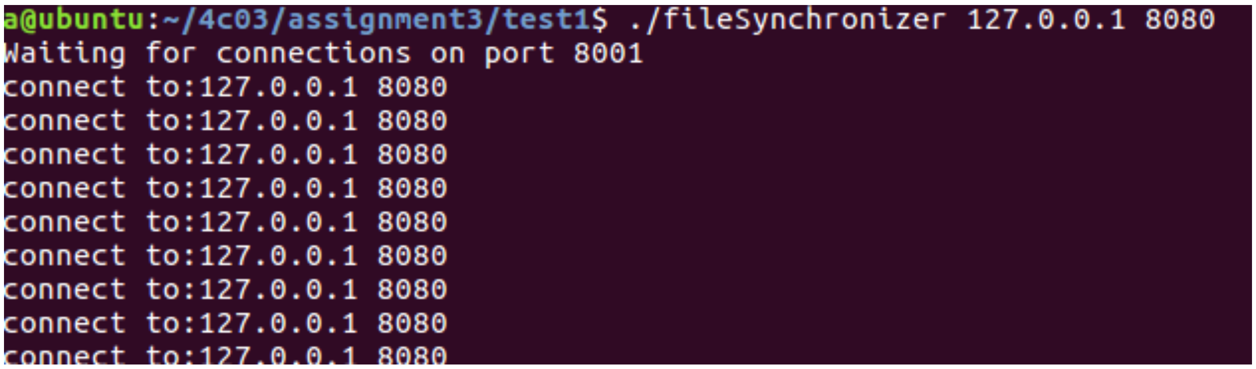
```
a@ubuntu:~/4c03/assignment3$ python tracker.py 127.0.0.1 8080
Waiting for connections on port 8080
```

(2) Prepare two test folders, e.g.,"test1" and "test2". Copy and unzip fileSynchronizer in both folders. You may find many *.so files, don't be scared, they are necessary to run the "fileSynchronizer". Create "test_file.txt" in "test1" folder.



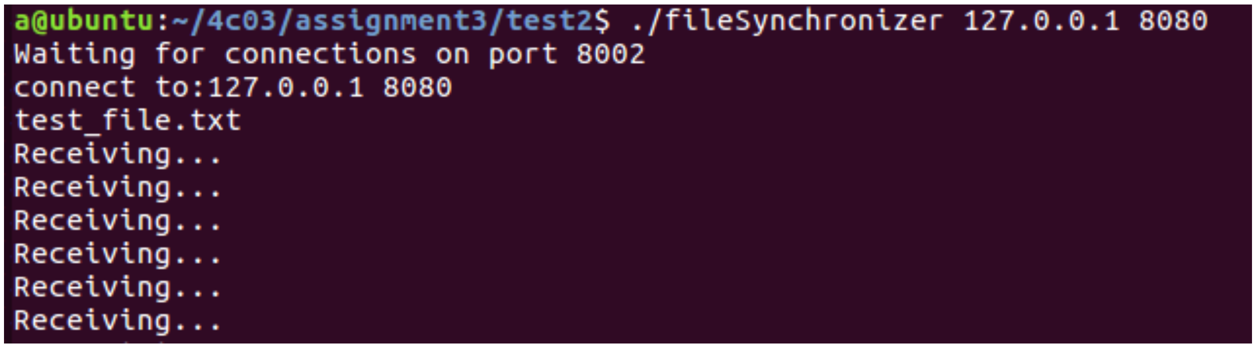
```
datetime.so             resource.so
dbm.so                  select.so
fcntl.so                strop.so
fileSynchronizer        termios.so
grp.so                  test_file.txt
itertools.so            time.so
```

(3) In folder "test1", run “fileSynchronizer” to connect to the tracker.



```
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" by creating a different text file such as “xxyy.txt” with any content. (you may need to open another console window)



```
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 "test_file.txt", which is the result of synchronization from "test1"

Reference:

- JSON encoder and decode https://docs.python.org/3/library/json.html
- Socket programming https://docs.python.org/3/library/socket.html
- Python threading https://docs.python.org/3/library/threading.html