Programming Assignment: Peer-to-Peer File Synchronizer

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.

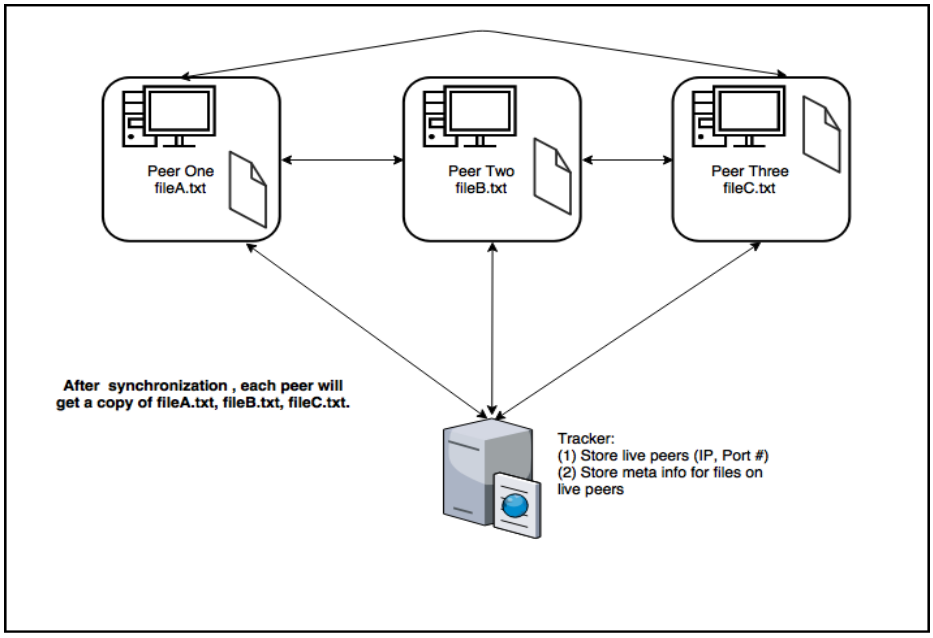

Figure 1 P2P file sharing architecture

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 trace 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.
- ✓ BIN – a folder which contains the tracker in binary format (include Linux, Mac, Windows, and a specific version for mills.cas.mcmaster.ca Linux server in CAS department).

Specifications:

- The synchronizer takes as command line arguments, the *IP address* and *the port number* (Port A) that the tracker runs.
- The synchronizer chooses **an available port** (Port B) to bind to and listen for file requests from other peers. This port will be used to accept incoming connection requests and **transfer files to other peers**.
- The synchronizer **communicates with the tracker** through a single TCP socket to
    1. send an **Initial message** containing names and the last modified time in seconds (round down to integer if it is a float in your platform) of its local files as well as **file serving port** (Port B) when it first starts.
    2. send **Keepalive** messages containing the **file serving port** every **5** seconds to the tracker
    3. receive the directory information maintained by the tracker

- Upon reception of an Initial message or Keepalive message from a peer, the tracker refreshes the state of the peer and also sends **Directory Response message** to the peer. (*this has been implemented for you* in the tracker)
- Upon receiving a **Directory Response message** from the tracker, the synchronizer parses the message and identifies files that are **either new or more recent** than its own copy by comparing with the **modified time** of local files of the same names. If such a file is found, it **connects the corresponding peer** using the peer's IP address and Port number contained in the message. Upon a successful connection, it sends a **File Request message** with the name of the requested file to the said peer. The file from the peer shall be stored in the same directory that the synchronizer runs. Make sure to set the modified time for the newly retrieved file to the **modified time** of the file in the Directory Response Message. This process is repeated until all new or more recent files contained in the **Directory Response Message** have been fetched.
- Upon receiving **a File Request message** from a peer, the synchronizer responds with the content of the file.
- In the case that a connection to a peer times out (e.g., using socket.settimeout) or there is no response to the **File Request message**, the synchronizer should proceed to the next file if any in the **Directory Response message**.
- The synchronizer should close client sockets that are not in use to other peers.

As clear from the specification, the file synchronizer implements both **a server and a client**. It **acts as a server to respond to file requests from other peers**. It is also **a client when requesting files from other peers**. Therefore, it is advisable to use multi-thread programming for this purpose. The state diagram of the file synchronizer is given in Figure 2. Messages exchanged between tracker and peers (synchronizers) are encoded in JSON as specified in Table 1. Messages exchanged between peers are NOT encoded in JSON, instead, they are either plain text (filename, text file content) or binary (binary file content) as listed in Table 2.

Table 1 Message Format between tracker and synchronizer

Message Type	Purpose	Message Format (key, value)	Actions in response to 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':int}]} Note: round the mtime to integer if it is a float. e.g., {"port": 8001, "files": [{"mtime": 1548878750, "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':} e.g., {"fileB.txt": {"ip": "127.0.0.1", "mtime": 1548878750, "port": 8002}, "fileA.txt": {"ip": "127.0.0.1", "mtime": 1548878750, "port": 8001}}	A peer requests new or modified files from other peers

Table 2 Message Format between peers

File Request Message	From **peer to peer:** Request a new or more up-to-date file on the peer.	**Plain text** of the **file name**	Peer responds with the content of the file.
File Response Message	From **peer to peer**: Send the content of the requested file	**Content of the file**, it can be binary or text	Peer saves the received file locally.

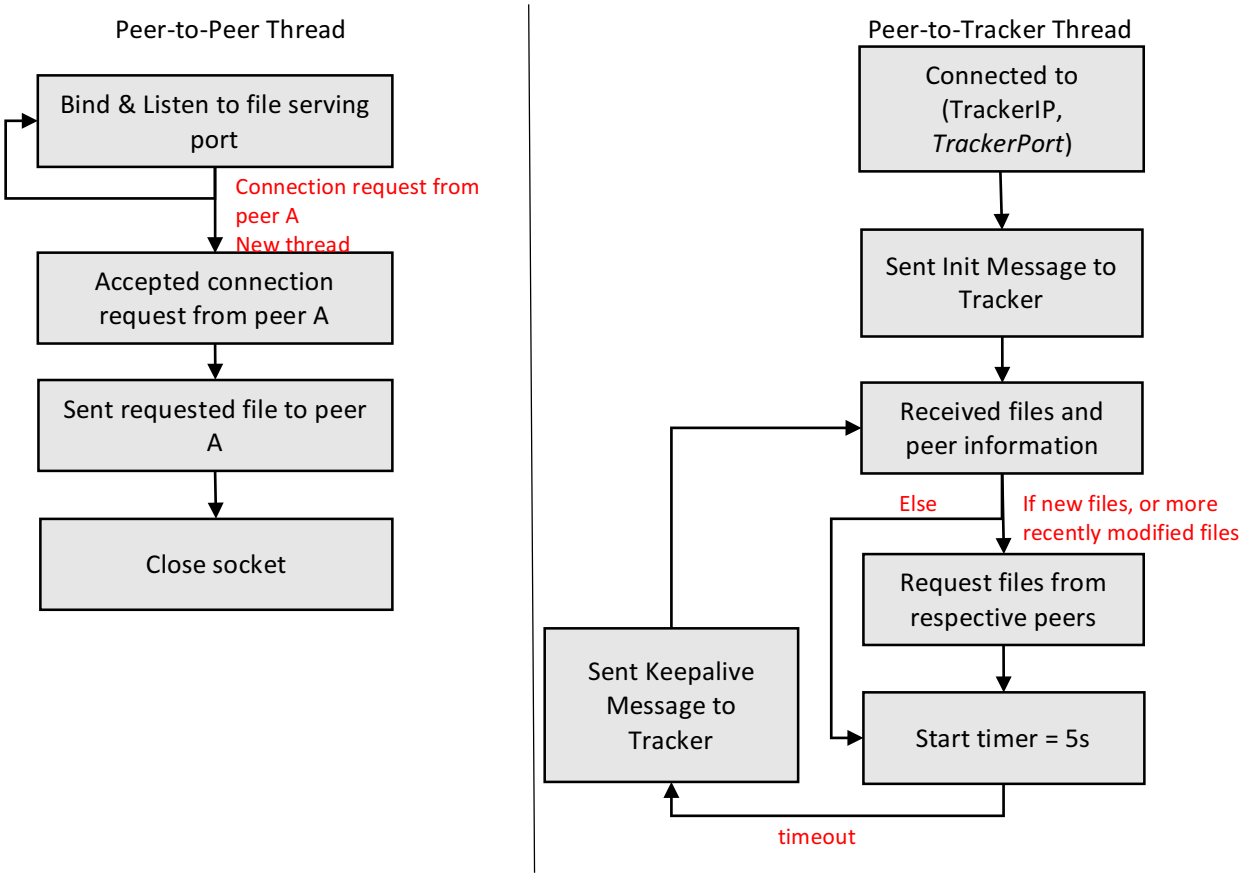

Figure 2 State Diagram of the file synchronizer.

For simplicity,
1) We only consider files within the same folder where the 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 **last** modified timestamp will be kept by the tracker.
4) Only one TCP connection is needed between a peer and a tracker.

Suggested Steps:

1. Read this document carefully to understand the logic.
2. Decide which platform you will use for development and testing, use your own computer if possible. **If you cannot find a binary tracker (from tracker_linux.zip, tracker_mac.zip, tracker.windows.zip) that can be successfully executed in your** computer**, ssh to mills.cas.mcmaster.ca with your CAS credentials**, upload and run the tracker code there. (you need to put a copy of the binary tracker on mills using *scp*)
3. Follow **Appendix I** to setup the programming environments if you use mills.
4. Enjoy! Any questions please discuss through piazza or go to lab sessions.

Note:

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

Submission:

1. Your code should be named fileSychronizer.py
2. Document your test cases and results in 'report.pdf'.
3. Put YourStudentNo.py and report.pdf in a single Zip file with name 'YourStudentNo.zip' and upload to Avenue dropbox.

Appendix I: Step-by-step guide to test your implementation

(The below example is done in Mac. A similar procedure can be followed in Linux. For windows, use the corresponding command for creating folders and copying files).

1. Prepare 3 folders named Server, Test1, Test2:

```
➜  P2Psharing mkdir Server Test1 Test2
➜  P2Psharing ls -lh
total 5056
drwxr-xr-x@ 6 yongyongwei  staff   204B 11 Jan 15:15 BIN
drwxr-xr-x@ 2 yongyongwei  staff    68B 28 Jan 09:51 Server
-rw-r--r--@ 1 yongyongwei  staff   6.9K 11 Jan 20:34 Skeleton.py
drwxr-xr-x@ 2 yongyongwei  staff    68B 28 Jan 09:51 Test1
drwxr-xr-x@ 2 yongyongwei  staff    68B 28 Jan 09:51 Test2
-rw-r--r--@ 1 yongyongwei  staff   2.5M 11 Jan 19:10 instructions.docx
drwxr-xr-x@ 4 yongyongwei  staff   136B 11 Jan 20:34 instructor-only
-rw-r--r--@ 1 yongyongwei  staff   162B 28 Jan 09:26 ~$structions.docx
```

Now find the appropriate binary format server from the BIN folder and copy to the Server folder and unzip it:

```
➜  P2Psharing cp BIN/tracker_mac.zip Server
➜  P2Psharing ls
BIN                 Skeleton.py         Test2               instructor-only
Server              Test1               instructions.docx   ~$structions.docx
➜  P2Psharing unzip Server/tracker_mac.zip -d Server/
Archive:  Server/tracker_mac.zip
```

Make sure you can run the tracker:

```
➜  P2Psharing ./Server/tracker/tracker 127.0.0.1 8899
Waiting for connections on port 8899
```

2. Open another console window and navigate to the Test1 folder, and copy the Skeleton.py to this folder and rename it to fileSynchronizer.py, then finish the 'YOUR CODE' part (you can use any text editing tools):

```
➜  Test1 cp ../Skeleton.py fileSynchronizer.py
➜  Test1 vim fileSynchronizer.py
```

3. When you finished the code, create a file to test synchronization:

```
➜  Test1 echo "hello, this is from client1">t1.txt
➜  Test1 ls
fileSynchronizer.py t1.txt
➜  Test1
```

Also, open another console window in the Test2 folder and copy your code to that folder. Similarly create another file.

```
➜  Test2 cp ../Test1/fileSynchronizer.py .
➜  Test2 echo "hello, this is from client 2">t2.txt
➜  Test2 ls -lh
total 24
-rw-r--r--@ 1 yongyongwei  staff   6.3K 28 Jan 10:07 fileSynchronizer.py
-rw-r--r--@ 1 yongyongwei  staff    29B 28 Jan 10:07 t2.txt
➜  Test2
```

4. Now start your file synchronizer at the two test folders, and check if the two files are synchronized (you may use another window to check or stop the current running fileSynchronizer).

```
➜  Test1 ~/anaconda2/envs/py37/bin/python fileSynchronizer.py 127.0.0.1 8899
Waiting for connections on port 8001
('connect to:127.0.0.1', 8899)
('connect to:127.0.0.1', 8899)
received file name: t1.txt
('connect to:127.0.0.1', 8899)
t2.txt
```

```
➜  Test2 ~/anaconda2/envs/py37/bin/python fileSynchronizer.py 127.0.0.1 8899
Waiting for connections on port 8002
('connect to:127.0.0.1', 8899)
t1.txt
Receiving...
```

(note the output may be different, depending what you want to show by yourself)

```
➜  Test1 ls
fileSynchronizer.py t1.txt                    t2.txt
➜  Test1
```

```
➜  Test2 ls
fileSynchronizer.py t1.txt                    t2.txt
➜  Test2
```

5. Restart from the above steps for debugging purposes. To restart the program use 'ctrl' + '\'

Appendix II: Step-by-step guide to test your implementation based on the mills server.

1. To get started, login to the mills server with any SSH tools (note you need to user VPN first, for windows use MobaXterm as the SSH tool):

```
➜  ~ ssh weiy49@mills.cas.mcmaster.ca
weiy49@mills.cas.mcmaster.ca's password:
Last login: Mon Jan 11 16:18:01 2021 from 172.18.216.78
weiy49@mills ~$
```

2. Then you need to upload tracker_mills.zip to your home directory (use scp in Linux or Mac, if in windows, you can drag files from Desktop to the server, please Google MobaXterm transfer files). In the case of scp, open another console window, execute the following command:

```
➜  BIN ls -lh
total 78528
-rw-r--r--@ 1 yongyongwei  staff   7.1M 13 Feb  2018 tracker_linux.zip
-rw-r--r--@ 1 yongyongwei  staff   4.6M 13 Feb  2018 tracker_mac.zip
-rw-r--r--  1 yongyongwei  staff   5.3M 11 Jan 15:15 tracker_mills.zip
-rw-r--r--@ 1 yongyongwei  staff    21M 13 Feb  2018 tracker_windows.zip
➜  BIN scp tracker_mills.zip weiy49@mills.cas.mcmaster.ca:~
weiy49@mills.cas.mcmaster.ca's password:
tracker_mills.zip                             100% 5470KB   1.3MB/s   00:04
```

(note you need to make sure you are in the BIN directory for the command to work)

3. Similarly, upload Skeleton.py to the server like below:

```
➜  P2Psharing ls -lh
total 104
drwxr-xr-x  6 yongyongwei  staff   204B 11 Jan 15:15 BIN
-rw-r--r--  1 yongyongwei  staff   5.6K 11 Jan  2019 Skeleton.py
-rw-r--r--@ 1 yongyongwei  staff    40K 30 Jan  2019 instructions.docx
drwxr-xr-x  4 yongyongwei  staff   136B 11 Jan 15:27 instructor-only
-rw-r--r--@ 1 yongyongwei  staff   162B 11 Jan 15:27 ~$structions.docx
➜  P2Psharing scp Skeleton.py weiy49@mills.cas.mcmaster.ca:~
weiy49@mills.cas.mcmaster.ca's password:
Skeleton.py                                   100% 5728   250.9KB/s   00:00
```

4. Now, open another two console windows and SSH to the mills server. In total, you need to maintain 3 SSH windows, one for the tracker to run, and two SSH windows simulating two clients.

In the 1st console window, unzip tracker_mills.zip and start the tracker:

```
weiy49@mills ~$unzip tracker_mills.zip
Archive:  tracker_mills.zip
   creating: tracker_mills/
  inflating: tracker_mills/tracker
```

```
weiy49@mills ~$tracker_mills/tracker 127.0.0.1 8877
Waiting for connections on port 8877
```

<Please use a large random port number to avoid conflicts with other students>

5. In the 2nd console window:

Create a test folder, copy the skeleton code into the folder and rename it like fileSynchronizer.py, and then create a test file to demo synchronize:

```
weiy49@mills ~$mkdir test1
weiy49@mills ~$cp Skeleton.py test1/fileSynchronizer.py
weiy49@mills ~$echo "this is content of f1.txt" > test1/f1.txt
weiy49@mills ~$ls test1
f1.txt  fileSynchronizer.py
```

Now you can edit fileSynchronizer.py by open it with vim or other text editing tools in Linux. You need to finish the YOUR CODE section.

You can test your implementation by running it (make sure the tracker is already started):

```
weiy49@mills test1$vim fileSynchronizer.py
weiy49@mills test1$python fileSynchronizer.py  127.0.0.1 8877
Waiting for connections on port 8001
('connect to:127.0.0.1', 8877)
received message from tracker: {"f1.txt": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610407343}, "c
ore.24641": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610408296}}
('connect to:127.0.0.1', 8877)
received message from tracker: {"f1.txt": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610407343}, "c
ore.24641": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610408296}}
('connect to:127.0.0.1', 8877)
```

(note the output may be different since it is your own implementation and you can decide to print whatever information)

6. When you finish the code, in the 3rd console window simulates another client:

```
weiy49@mills ~$mkdir test2
weiy49@mills ~$cp test1/fileSynchronizer.py test2/
weiy49@mills ~$echo "this is from f2.txt" > test2/f2.txt
weiy49@mills ~$cd test2
weiy49@mills test2$ls
f2.txt  fileSynchronizer.py
weiy49@mills test2$vim fileSynchronizer.py
weiy49@mills test2$python fileSynchronizer.py 127.0.0.1 8877
Waiting for connections on port 8002
('connect to:127.0.0.1', 8877)
received message from tracker: {"f1.txt": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610407343}, "c
ore.24641": {"ip": "127.0.0.1", "port": 8001, "mtime": 1610408296}, "f2.txt": {"ip": "127.0.0.1", "p
```

Now you can open a fourth console window and check if f1.txt and f2.txt has been synchronized in the two folders:

```
weiy49@mills test1$ls -lh
total 16K
-rw-r--r-- 1 weiy49 grad   26 Jan 11 18:22 f1.txt
-rw-r--r-- 1 weiy49 grad   20 Jan 11 18:46 f2.txt
-rw-r--r-- 1 weiy49 grad 6.4K Jan 11 18:39 fileSynchronizer.py
```

```
weiy49@mills test2$ls -lh
total 16K
-rw-r--r-- 1 weiy49 grad   26 Jan 11 18:22 f1.txt
-rw-r--r-- 1 weiy49 grad   20 Jan 11 18:46 f2.txt
-rw-r--r-- 1 weiy49 grad 6.4K Jan 11 18:47 fileSynchronizer.py
```

If you want to restart the server or the client after editing the code, use 'ctrl' + '\' to terminate a running tracker or peer. (It may generate some core.* files in mills, you can either ignore or delete them)

Reference:

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