Programming Assignment 2: Peer-to-Peer (P2P) File Synchronization (Part II)

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

Early submission will receive 1.5% bonus points per 24hr ahead of the deadline. For example, if you submit on March 9nd, you will get 10.5% bonus over the mark you would've gotten otherwise.

In this programming assignment, you will continue the development of 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 `Tracker` in this project, is needed to facilitate the "discovery" of peers. In Part I, you have developed the server program for *Tracker*. In Part II, you will develop the `FileSynchronizer` program for the peers.

In the project folder, you will find

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

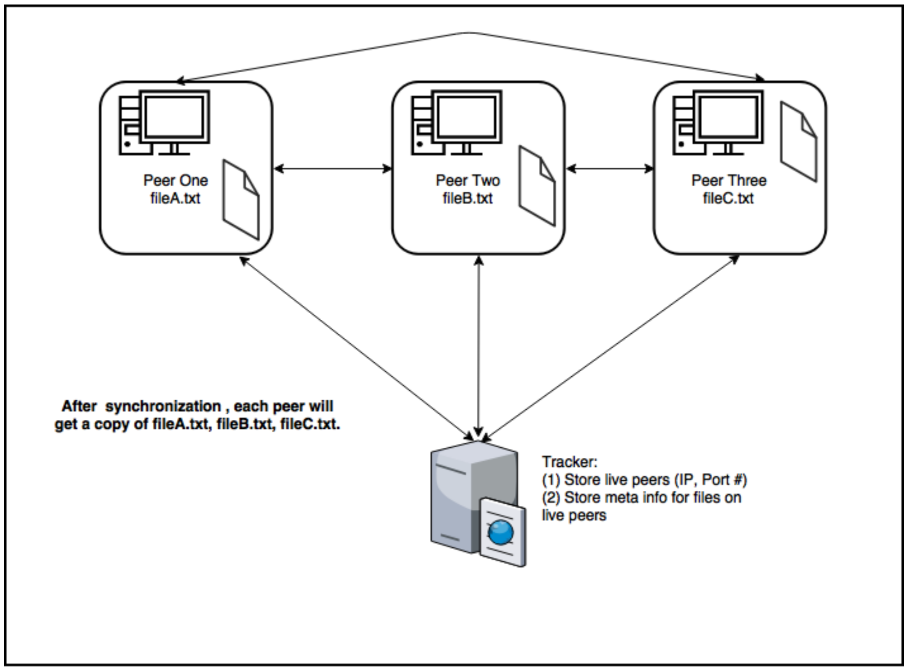
Peer One
fileA.txt
Peer Two
fileB.txt
Peer Three
fileC.txt
After synchronization , each peer will get a copy of fileA.txt, fileB.txt, fileC.txt.
Tracker:
(1) Store live peers (IP, Port #)
(2) Store meta info for files on live peers

Figure 1 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:

- `FileSynchronizer` takes as command line arguments the IP address/hostname and Port number of Tracker (is running on, e.g., `FileSynchronizer localhost 8080`
- `FileSynchronizer` will chose an available port to bind and listen to (Hint: use the `check_port_avaliable()` method provided). This port (*FileSynchronizerServerPort*) will be used to accept incoming connection requests and file transfer from other `FileSynchronizer`.
- `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
- Upon receiving a `Directory Response Message` from the `Tracker,` `FileSynchronizer` parses the message and identifies files that are either new or more up-to-date than its own copy by comparing with the modified time of its local files. If such a file is found, it connects the corresponding peer at the respective IP address and Port number contained in the message, and send a `FileRequest` message including the name of the file. The received file shall be stored in the same directory that `FileSynchronizer` runs. `FileSynchronizer` repeats this process for each new or more up-to-date files contained in the Directory Response Message.
- Upon receiving a `FileRequest` message from a peer, `FileSynchronizer` responds with the content of the file (for simplicity, there is no header information and no error handling).
- `FileSynchronizer` should close *client* sockets that are not in use to other peers.

As clear from the specification, `FileSynchronizer` contains 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 `FileSynchronizer` is given in Figure 1. (Application-layer) Messages exchanged between Tracker and `FileSynchronizer` are encoded in *JSON* specified in Table 1. (Application-layer) Messages exchanged between `FileSynchronizer` peers are listed in Table 2.

Table 1 Message Format between Tracker and FileSynchronizer

Message Type	**Purpose**	**Action**	**Message Format (key, value)**
***Init* Message**	From FileSynchronizer to Tracker. • Upload meta file information and *FileSynchronizerServerPort* • 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	{‘port’:int}

		2. Tracker responds with a directory response message	
Directory Response Message	From Tracker to FileSynchronizer. Return the directory at the tracker.	Peer retrieve new or modified files from other peers	{filename:{‘ip’:string,’port’:int,’mtime’:long}

Table 2 Message Format between FileSynchronizer peers

FileRequest Message	From FileSynchronizer to FileSynchronizer. Request a new or more up-to-date file on the peer.	Peer responds with the content of the file.	filename (filename being the name of the file in string)
FileResponse Message	From FileSynchronizer to FileSynchronizer. Send a file.	Peer saves the received file locally.	Content of the file

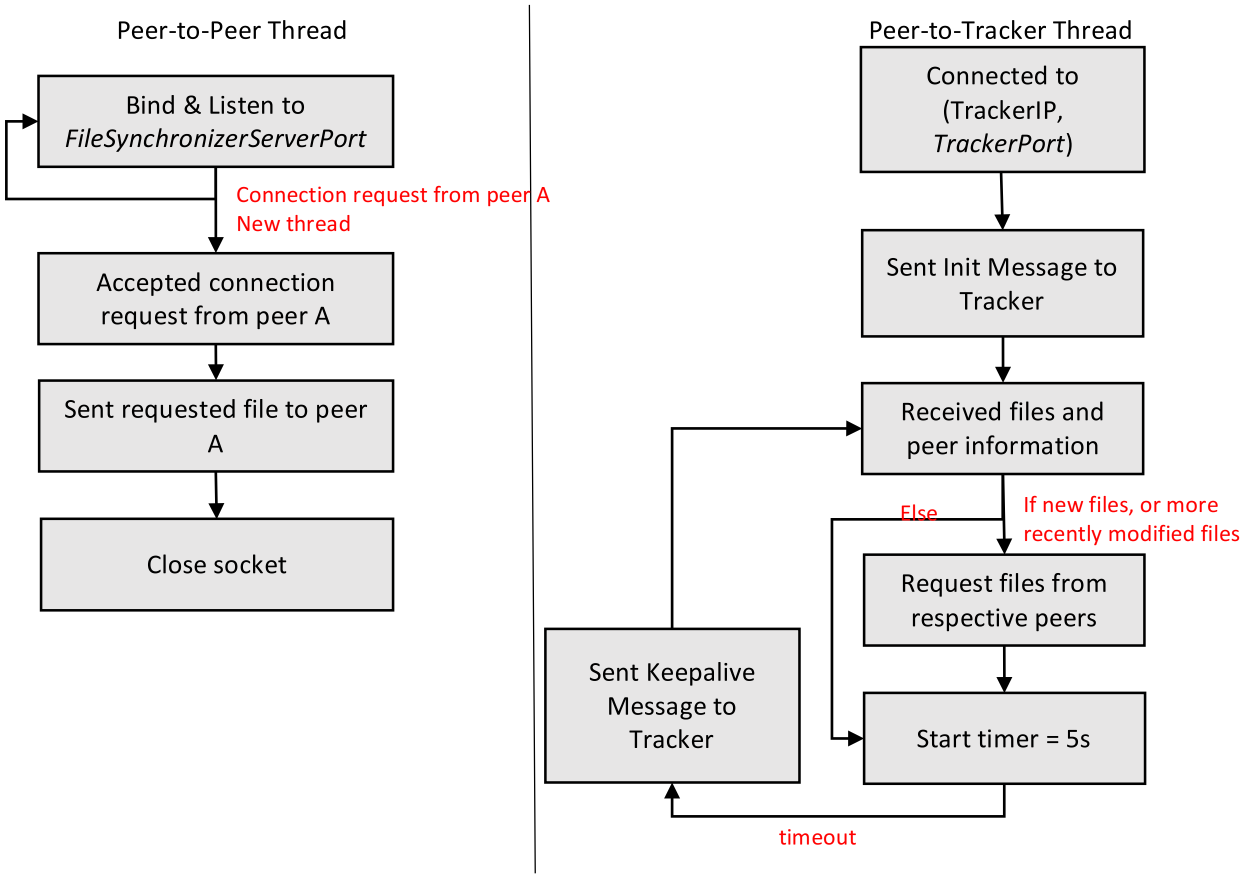
Peer-to-Peer Thread
Bind & Listen to *FileSynchronizerServerPort*
Connection request from peer A
New thread
Accepted connection request from peer A
Sent requested file to peer A
Close socket
Peer-to-Tracker Thread
Connected to (TrackerIP, *TrackerPort*)
Sent Init Message to Tracker
Received files and peer information
Else
If new files, or more recently modified files
Request files from respective peers
Start timer = 5s
Sent Keepalive Message to Tracker
timeout

Figure 2 State Diagram of `FileSynchronizer`.

For simplicity, 1) we only consider files in the same directory 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; 4) there is no error handling, namely, we assume peers always have the files available upon requests.

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

Instructions:

1. Your implementation should be based on Python 2.7 for compatibility
2. 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 code in a SINGLE file called ‘filesynchronizer.py’ to ‘PA2/src’ folder in your gitlab project.
2. Document your test cases/procedures in PA2/report