Design Documentation of P2P System

By Vishant Bhole (A20411225) , Saptarshi Chatterjee(A20413922)

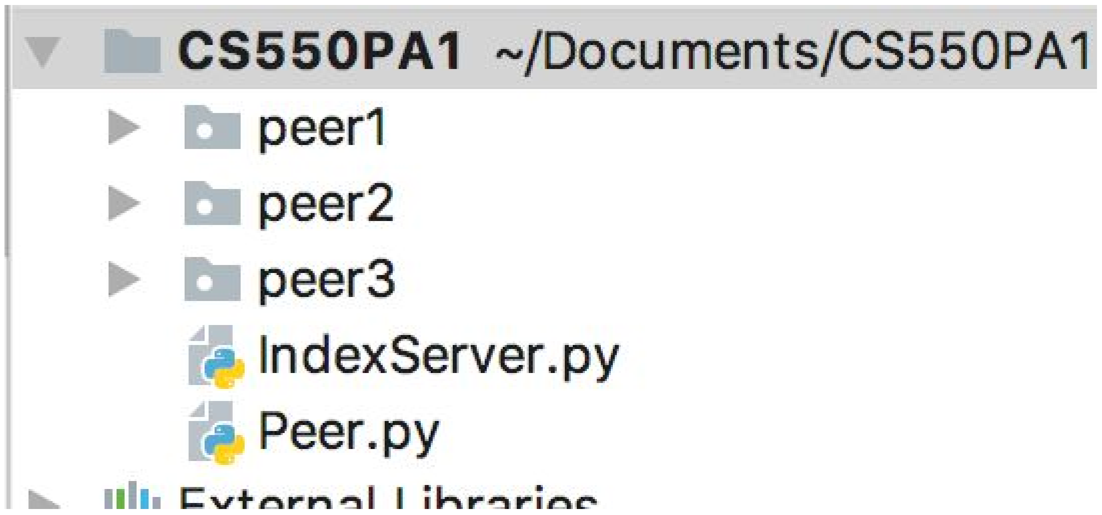
Project Structure -

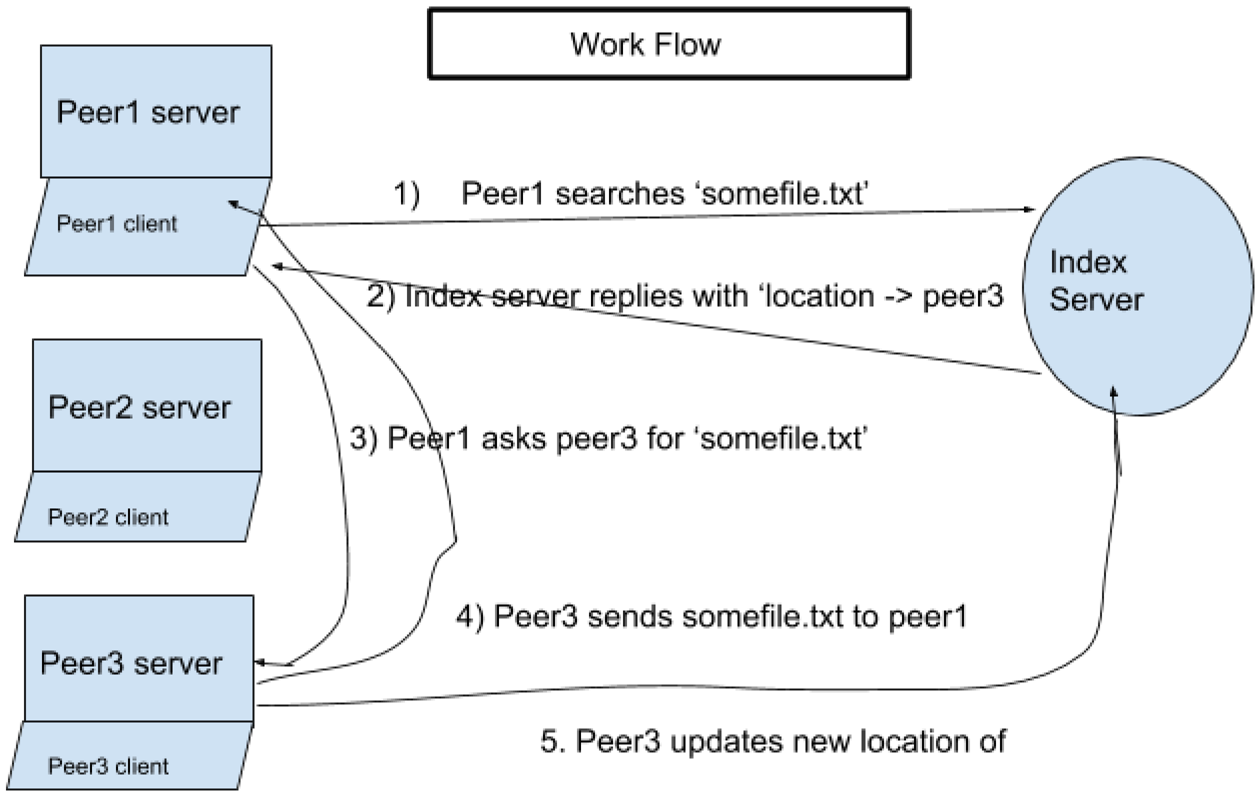
We have implemented the P2P system using Python3.6 , and used socket module which is a Low-level networking interface. This module provides access to the BSD *socket* interface

Application got 2 files and 3 storage folders -

1) IndexServer.py - Responsible for Keeping an registry of all the files in the distributed system, and serving result for file searches.
2) Peer.py - Includes both Client and Server Modules codes.
    a) When ran in client mode, it is responsible for taking filename as a user input from console, searching file from IndexServer, Requesting the server for file, and storing the file to Local Disk, and once downloaded updates the IndexServer.

    b) When ran in server mode, it is responsible for accepting connection from client and send the requested file. It is also responsible for updating the Index Server with it's local file locations.
3) The 3 folders 'peer1', 'peer2' and 'peer3' just have files of different sizes in them . And abstracts local disk for peers. E.g. Peer1 can only serve files from 'peer1' directory and so on. We tested and documented with 3 peers , hence only 3 directories. If you want to test with more than 3 clients you will have to create more directories with the same name as peer.




Work Flow
Peer1 server
Peer1 client
Peer2 server
Peer2 client
Peer3 server
Peer3 client
Index Server
1) Peer1 searches 'somefile.txt'
2) Index server replies with 'location -> peer3
3) Peer1 asks peer3 for 'somefile.txt'
4) Peer3 sends somefile.txt to peer1
5. Peer3 updates new location of

High Level Design -

ServerModule

When the argument 'server' is passed with the peerID, ip and the port number, then the receiveConnection module is called. In that module we have used the socket tcpsock where we have to select the domain and the type. So we need to specify the domain to AF_INET which is used for the IPv4 internet protocol and the type to SOCK_STREAM which Provides sequenced, reliable, two-way, connection based byte streams. Then for setting the socket we need to specify the SOL_SOCKET to manipulate the socket-level options described in this section and SO_REUSEADDR controls whether bind should permit reuse of local addresses for this socket inside the setsockopt().Then we need to use bind() which binds the socket to the address and port number specified in address.
Then in the loop we need to listen() which puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection and accept() which extracts the first connection request on the queue of pending connections for the listening socket and creates new connected socket, and returns a new file descriptor referring to that socket, where connection is established between client and server, and they are ready to transfer data. Now when the client and server are ready to send and receive the data we can receive the filename from the client using recv() and we need to decode() it as it was encoded while sending from client, there we print 'receiving request' with ip address, port number and

‘new sender started client’ with ip address, port number from where the actual file is to be received.
Then initialize new thread object as a newthread where we will call the ServerModule class for the actual sending and calls newthread.start() method for starting this thread which can be called only once, then we just append the newly created thread using append() method. And when the start() method is called it invokes run() method in a separate thread of control and invokes the callable object passed to the object’s constructor as the target arguments.
Before doing anything if the subclass overrides the constructor, it must make sure to invoke the base class constructor __init__() before doing anything else to the thread which are defined in the ServerModule class. Inside the definition of the run() method we use the fileName which we received earlier to reads the content of the file in small parts. We read these small parts in filepart with the BUFFER_SIZE of 4096. Then we send this file part by part till the filepart is null. When the filepart is empty we just close the file and the socket and print file transfer successfully.
And at the end of the receiveConnection we just call the join() method this will prevent Main thread to terminate before child threads are terminated.

ClientModule

Client Module waits for user-input from terminal , Once input is received It looks into own local directory , if the file already exists , it does nothing and just waits for next search.

If file is not found on local system , then it searches index server for the file , and index server sends a valid peer , It opens a socket connection with the peer , and asks for file. It receives files in chunks and keep storing it into local. Once the entire file is received it updates the index server.

IndexServer
  A threaded server running on port 1024 . It keeps listening for connection and accepts message in 2 predefined protocol . For get requests -> *get|filename* and for set requests -> *set|filename|host|port .* It maintains a dictionary in main memory in following format

```
{
'onekb.txt': {'localhost|peer1|1025','localhost|peer2|1026' }
'tenkb.txt': {'localhost|peer2|1026','localhost|peer2|1026' }
}
```

And for a get request send the file location (If multiple peer holds the file , it randomly selects 1, thus load balance) , and if it’s a set request , it updates the dictionary

<u>Limitations</u>

1) If Index server goes down then entire system halts. We should to have multiple load balanced Index server to solve this issue
2) Indexes are stored in main memory , so When it goes down it loses all the data . We should serialize (python pickle) the dictionary that holds all index in regular interval and store into disk . So that once index server is back up it can get back it's previous state.
3) Once we delete a file in filesystem it doesn't update the Index server automatically - Solution is very easy just add a file system watcher , and on change call the existing method talkToIndexServer to update it. Didn't implement it as I think it's trivial .

<u>CONTRIBUTIONS</u>

1) Architecture and System Design - Saptarshi
2) Peer Server Module - Saptarshi
3) Peer Client Module - Saptarshi
4) Design Doc - Saptarshi
5) Index Server design and implementation - Vishant
6) Verification - Vishant
7) User Manual -Vishant
8) Performance Result - -Vishant