CS 39006: Assignment 5
Implementation of P2P File Transfer
Date: 14-Mar-2022
Deadline: 30-March-2022 2:00 PM

In this assignment, we will simulate a peer-to-peer (P2P) network. You are quite familiar with client server architecture by now. In addition to the client-server architecture, P2P forms another popular networking architecture paradigm. Let us first go through some theory. In P2P, there are no specific systems identified separately as a client or a server. Rather all the computers connected in a P2P type network are treated as equal peers with similar functional and operational status (i.e no system is considered to be the master over any other system or that no system is considered superior to some other system with respect to computational and functional capabilities. All nodes are equal peers, as stated above). A basic diagram of P2P architecture is given below, shown with only four nodes for simplicity. All the four systems are equal peers within the network and all of them are connected with each other (mesh topology) via a carrier network (such as the Internet). Any system can request any other system, or request all the other systems simultaneously, or request a subset of the other systems present in P2P for some service or file/data transfer. The requested systems respond accordingly based on whether they can actually provide the service or not or whether they actually contain the requested data/file or not. In this P2P architecture, no system can be identified separately as exclusively a client or a server. In fact we can consider it in the following way. Each system in a P2P network acts as both the client and server based on what action it is currently performing. If one entity is requesting some data/service/file, then that entity acts as the client for that particular request session and the entities that are providing the requested services/data/information acts as the server(s) for that session.

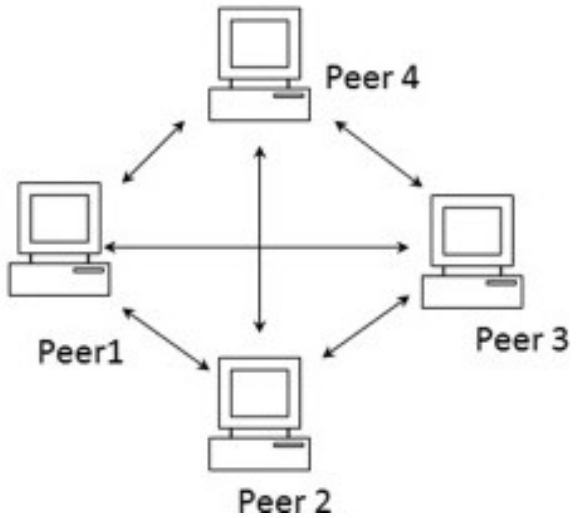

Fig. 1: A simple P2P overlay network with four peers.
(Source: https://www.sciencedirect.com/science/article/abs/pii/S1574013712000123)

One interesting aspect of any P2P network is that the network architecture topology (again consider the figure above) is considered as what is called an 'overlay network', i.e the connections shown between pairs of participant peers of the network are actually not actual physical connections, but logical ones. So the shown P2P network with four network and their interconnections shows the logical topology among the participant peers. The peers themselves may be separated physically by large distances and the data transfer is handled by a carrier transport network (like the Internet). In a P2P architecture, new nodes can join as fresh peers to the network and existing nodes may leave the network. Thus the architecture of a P2P network is dynamic, depending on the currently active peer nodes.

Programmatically how will you build such a P2P architecture? You can do so by considering that each participant peer computer of the P2P network runs BOTH the client code and the server code within the same system. When the system acts as the client (such as the requester of a file), the client code is one responsible for data transfer and connecting with other server peers. When the same system acts as the server, then the server code handles the incoming requests and responds to them. Since each computer in the P2P network runs both the type of codes (client and server), the port number on which the server code binds for that peer is also the port number that is globally known by other peers for that peer. In fact the tuple {IP_address:port_number} (considering IP type network and AF_INET socket) that is associated with the server code of each peer, is also the global unique identifier of that peer among the pool of peers.

Peer-to-peer file sharing:

P2P file sharing over the Internet is one of the most popular applications of a peer-to-peer network. The real P2P file sharing protocols that are used are, as usual, very complex with a lot of features. Instead, let us consider a simplified P2P file sharing model. Suppose there are 'n' peers currently active within the network, each peer knows about the existence of every other peer. Now one peer wishes to receive a file that is not present with itself. This peer acts as the client then. It sends out a request to all the other peers to know which among them has the requested file. The other peers respond with a success code based on whether they possess the requested file or not. The other peers act as servers. Upon response from the server peers, the client peer now chooses any one among the server peers to send the file to itself. Alternatively, the client peer may also send out a request to all other peers simultaneously to send the requested file part by part (for example if there are 4 server peers each having the requested file of size 1000 characters, then the client may ask server 1 to send the first 250 characters, server 2 the next 250 characters, server 3 the next 250 characters and likewise, until it receives the full file).

One more issue to solve is how each peer knows about the existence and the unique identifier (i.e IP address and port number) of every other peer. This is handled by a variety of means. Among all the peer nodes, there may be a central special node whose identifier is public and is known to everyone. This central node contains the list of other peers in the network. A new peer that joins the network will have to first register itself with this central node. The central node then shares the

identifiers of the newly joined node with all existing nodes. Similarly when a node leaves the network willingly, it notifies the central node, which then removes the leaving node from the list and also informs the same to other nodes. Another approach is network flooding, whereby a newly joined peer floods the network using a broadcast message with the information that it has newly joined. Other existing nodes then respond to the flood request with their identifiers. This second approach is, however, not suitable for big networks across MANs/WANs; it can be implemented across LANs (like campus networks).

As mentioned, we will now simulate only a simplified P2P file sharing model, as described above in this assignment. Let us come to the exact details of what you have to do.

Assignment specifications:

You will simulate the P2P file sharing with 5 peers, writing both the client code and server code **for each** peer. Since most probably you will be writing and testing all the codes in your local system and will use localhost:127.0.0.1 as the default IP (or INADDR_ANY), we will simulate the 5 different peers through 5 separate pairs of client-server programs, each pair being executed from a separate folder and each pair being associated with a different port number, so that the tuple {IP_address:port_number} is unique to all. For all the codes we will be using UDP sockets only for communication, the reason being that in a P2P network, there are a lot of peers. The list of steps to follow are as below:

1. First create 5 separate folders within your system, name the folders as **peer1**, **peer2**, **peer3**, **peer4** and **peer5.** The folders can be located anywhere within the file system.
2. Create another folder **master_peer** which will contain the common central peer whose identifiers are known publicly to all.
3. Inside **master_peer** write a server code **masterp2pserver.c**, which is a plain udp server. This server is known publicly to all and is responsible for sharing the information to newly joined peers about other existing peers. Bind this server to listen to any port number, say UDP port 40000. This server holds a database that contains the identifier of all other peers of the network. The identifiers are in the form of the tuple {IP_address:port}, that uniquely identifies each other peer of the network. You can implement this database as a structure, array of pointers, string pointers, etc in any way you feel. But implement this database in a separate function named **peer_database()** so that it is easier for us to identify where you are implementing the database. The return type and input parameters of this function is up to you. Run this server and make it continuously listen on port 40000 for newly joined peers.
4. **For all the steps listed below, replicate them exactly for each of the five peer folders that you have created**. For readability, let us consider only the folder **peer1**, but do the same for the remaining four peers as well. Of course make sure that the file names are changed accordingly.

a. Insider **peer1** folder, create two C programs, a client **peer1client.c** and a server **peer1server.c.** Bind the server peer code to the mentioned UDP port (the port numbers for each server peer are given at the end). Run the server and make the server go to the listening state. Run the client also simultaneously and open a console for the user to interact with this client.
b. Inside the peer1 folder, also keep some text files, each file containing any number of ASCII characters (similar to the file you considered for Assignment 1). Make sure that you keep different files with different names in all the 5 peer folders.
c. This peer1 will now work in the following way. The moment the server and the client part of peer1 are up and running, the client displays a message on its console that requests the user to press 'R' to register this peer1 with the master peer. Upon the appropriate user input (i.e 'R'), the client of peer1 sends a joining registration request to **masterp2pserver.c.** In the joining request message the client sends the tuple {IP_Address:port_number} of that peer (Note the port number is the port on which the server of peer1 is running). The **masterp2pserver.c** on receiving this joining request saves the joining peer's identifier on its local database and responds with a success code (Let us say 200 for successful registration and 500 for unsuccessful registration). The client displays the message "Registration to P2P network successful" or not successful based on the received code.
d. After successful registration the client console will display the message "Welcome to P2P file sharing. This is Peer 1. Enter the file name to start download" and wait for user input. DO NOT make the client connect to any other peers at this point, the connection will start based on the user choice, since at this point the client does not know which peer to contact regarding the file.
e. Once the initial registration is complete (and let us consider you did the same for the remaining 4 peers also), we will start with the file sharing. At this point, all 5 peer folders will have their corresponding server and client codes running along with 5 client consoles, each waiting for user input for the file name.

5. Open the client console of any one peer and enter a filename that this peer wants to download from the P2P network. Let us assume that although the five peer folders can be situated anywhere within the file system, for simplicity, the text files for all the 5 peers are kept within the local folder of that respective peer. After the user enters the name of the file (in <name_of_file>.txt format), the client first checks whether the file is present locally within that peer itself. If yes, it prints a message "File present locally, no need of file transfer". Otherwise, this peer now again inquires the **masterp2pserver.c,** but this time to know about the existence of other

peers in the network (Note the first connection to **masterp2pserver.c** was to register this peer). The master server responds with the identifier tuple of all other peers that it currently knows of. The master peer sends the identifiers of all the other registered peers, except the calling peer, as a single string with a symbol, say '@' acting as a delimiter. It looks like this <peer1_identifier>@<peer2_identifier>@<peer3_identifier>@... The requesting peer client upon receiving this parses the identifier accordingly. If there are currently no other peers registered with the master (except the calling peer of course), the master peer responds "No Peer Available".

6. The client, upon receiving the identifiers of all peers, **iteratively** inquires each other peer about the file that the user requested. Note that the client sends this inquiry request to the servers of the corresponding peers. The server codes of the other peers receives this request and checks whether it has the requested file. If no, then it responds with a message "FileError". If yes, then the server responds with the message "FileFound", along with the size of the file with respect to the number of characters of the file. The other peer serves thus needs to calculate the size of the requested file (if it has the file) by number of characters before sending the response to the calling peer.
7. The client makes a list of all the peers that have the file. It also knows the file size. Now in order to expedite the file transfer, the client does not take the entire file from a single peer, rather it forks into a number of child processes equal to the number of peers that has the file. Each child process then makes a new connection to one peer server and **concurrently** requests a chunk of the file from that peer. The chunk size is calculated as the file size (in number of characters) divided by the number of peers that have the file. In addition to requesting the chunk from the peer servers, the respective client child process also mentions the starting character number and ending character number of the chunk from that peer server. The starting and ending character numbers should be with respect to the number of characters of the original file. To which peer server will the requesting client peer request what portion (i.e chunk) of the original file is upto your implementation.
8. The respective peer servers accordingly respond to the client request and send the respective file chunk to the requesting child process. Each child process writes the chunk it received into a temporary text file and returns to its parent client.
9. The parent client then finally collects these temporary files according to the correct order and compiles them into the final complete file. While doing this, the parent must be aware of which child has written what portion of the original file in the temporary files. This can be done by, say, the child clients writing the starting character number and the ending character number of its own chunk with respect to the original file as the first line of the temporary file. The parent can then open each temporary file and knowing what portion of the original file it contains, accordingly

compile the entire file, and deletes the temporary files that were created by the child clients. This finishes the file transfer. The exact same process should be replicated for any other peer acting as client also.

Use the following UDP port numbers for the 5 peer servers:

1. Peer 1 - 4050
2. Peer 2 - 4060
3. Peer 3 - 4070
4. Peer 4 - 4080
5. Peer 5 - 4090

NOTE:- Take special care with respect to the various sockets that you declare in the respective client and server codes. Remember that the client code especially needs to handle multiple sockets along with forking. So take great care.

Submission criteria:

You have to submit the five peer folders and the master peer folder along with the client and server codes and the text files that you have considered within each peer folder. Put all the folders into a master folder, zip it, name it <Your_roll_number_A5>.zip, and submit on Moodle. BUT PLEASE NOTE THAT WE WILL CHECK THE CODES WITH OUR OWN TEST FILES, SO ENSURE THAT YOUR CODE RUNS FOR ANY TEXT FILE. AS ALREADY MENTIONED, YOU ASSUME THAT THE TEXT FILES ARE SIMPLE AND WILL CONTAIN PLAIN ASCII CHARACTERS OF ARBITRARY LENGTH AND THE FILES WILL BE KEPT WITHIN THE LOCAL FOLDERS OF THE RESPECTIVE PEERS.