Advanced Computer Networks LAB
Spring Semester 2026
Assignment 2

Name – Soham Chakraborty
Roll No – 22CS02002

Note:- Throughout Assignment_2, I could have used HashMap or Set from STL in many cases for O(logN) searching, but I didn't because it was use-case dependent. Also, unnecessary usage of Heavy-Data-Structures might cause extra overload compared to simpler ones.

Part 1 –

1. **Problem:** How to ensure that taking input can be done along with reading and writing to client i.e. without using another thread and not blocking ?
   Solution: Put STDIN = 0 in the fd_readset.

2. **Problem:** How to check which fds are ready in the fd_readset efficiently without having to manually loop thorugh all of them ?
   Solution: Create a list of all_fds which we need to check and iterate through that loop only. This list is updated upon each new Peer entry or exit. Consider the case – Initially we have all_fds = {0, 3}. Then, peer with fds 4…10 connect and then 4…9 disconnect. Finally we have fd_set = {0, 3, 10}.
   So, ideally we need to check only 3 fds, but using a manual loop we have to go through all 11 fds from 0-10. Similarly, I have a **max_fd** integer which tracks the max_fd seen till now, and updates upon each new peer addition or deletion. This prevents the select() from having to go through all FD_SETSIZE+1 Integers.

3. **Problem:** How to uniquely identify each peer, so that I know from whom a certain message is coming ?
   Solution: Using pair peer **{ip, listening_port}** I am assigning a **uniqueID** to each client. {ip, port} combination will always be globally unique for each Peer. [Note that uniqueID for a Peer is unique w.r.t. someone, but for that same Peer the

uniqueID might be different w.r.t someone else. That is, it's locally consistent not globally.]. I am using a HashMap to map Peer {ip,port} → uniqueID.

4. **Problem:** If I am sending a message to someone who is not my neighbor using flooding. Then how to ensure that the receiver knows that he is the target destination ?
   Solution: Create a **header** and a separate payload. In the header section add **destination_ip** and **port**. If that matches with receiver's ip and port then this is the target.

5. **Problem:** How to ensure that a certain sent message doesn't keep on looping forever ?
   Solution: Create a message_id, which is a 32-bit random signed integer. Make a set of **seen_message**_ids. If a message is already seen then don't read it again. This is somewhat similar to visited array in dfs.
   Along with this there is also a **TTL** field in header, which reduces upon each hop. If TTL is < 0 drop that message (Initially, TTL set to 10).

Final Message Format – ***message_id | source_ip | source_port | destination_ip | destination_port | TTL | payload***

6. **Problem:** 2 simultaneous messages are colliding (clustered message) when Sending from same source→destination.
   Solution: I put a 1 sec sleep timer as a timeout after sending each message so that this is avoided. Another option is to use message end delimiters, but this one was easier to implement.

7. **Problem:** Broadcasting a message without increasing header size.
   Solution: Putting all Peer {ip,port} into the message header seemed like an inefficient solution. So I put an empty string in the destination_ip section and set destination_port = 0 (for General Broadcasts).
   Now, on receiver's end if I receive a message with empty ip, then I know that

this is a broadcasted message, in that case I read the message as well as forward it again, until it reaches all Peers.

8. **Problem:** If a new Peer is connected, then how to inform all other connected Peers about the details of the new Peer ?

   Solution: First, we do a somewhat handshaking protocol. If say *a* and *b* connect to each other, they exchange their name, ip and port with each other. After this *a* and *b* exchange their Peer lists with each other and broadcast this message to the remaining part of their previous graph.
   If *a* was part of G(a) and *b* was part of G(b) then, List(b) is broadcasted within G(a) and List(a) is broadcasted within G(b). Now, this ensures that everybody in G(a) U G(b) knows about all the others. We can prove that this is correct using Mathematical Induction on the Exchange List Process.

   Note:- When a new connection arrives I am not fully connecting that to all other nodes. I am just broadcasting their Peer List.
   We are treating this a Special Broadcast Message by using destination_port = -1 (instead of 0) and keeping destination_ip an empty string.

9. **Problem:** What if 2 peers who connected were already previously connected to each other via some other path (might not be neighbors previously) ?

   Solution: I am checking that using if the Peer is present in List of the other connection. Then I am not broadcasting this message, and instead just simply connecting the links between those Peers. Since, all other Peers already know about that Peer.

10. **Problem:** Broadcasting disconnection among all Peers.

    Solution: Just before directly doing close(sockfd) I am Broadcasting to all other Peers, the message that I disconnected. Using a Special Broadcast Message i.e. destination_port = -2. On the receiving side I am removing that Peer from the List and if it was a neighbor then removing from the neighbor_fd set and close(sockfd).

11. **IMPORTANT Unsolved Problem:** Efficiently Updating respective Peer Lists for each Peer if the disconnected Peer was an **Articulation Point/Cut-Vertex** in the graph.

Probable Solution: I couldn't come up with any ***O(n)*** solution to solve this problem. So the only solution was that from Each Peer I send a NewConnectionUpdate() message to every other Peer saying that "I am still connected". In which case the Peer List for all Peers is calculated afresh. This method will take ***$O(n^2)$*** time complexity. All other Operations till now were taking ***O(n) or O(diameter of Graph)*** time.
Take Example: (1-2-3-4-5). Now if 3 disconnects, there is no way for (1-2) to know that (4-5) can no longer be reached and vice-versa. So, my solution is, From each of the remaining Peers broadcast a message to everyone else and whoever can read that message is still reachable. In this case, Broadcast from 1 only reaches 2 and Broadcast from 2 only reaches 1. Thus, they update their Peer List and remove (4,5) from the List which were previously present. Same goes for (4-5).

12. **Problem:** DEBUGGING.
    Solution: Commenting or uncommenting print statements each time was a tedious task. So I made a debug() function, where I put the debug message. If I run the code normally using - ./a.out port_no. Then no, debug() is printed.
    It is only printed when I run it as - ./a.out port_no 1. Putting second argument as 1, enables all debug() within the code and prints them.

The problems given above are only the significant issues that I faced and remembered. There, were many other Problems and Bugs in the code when actually running and debugging them.

Part 2 –

Most of the problems were solved in Part 1. This was just a layering fully based on Part 1. This also has some minor bug fixes to Part 1.

1. **Problem**: Distinguishing if a file is already owned by someone else previously.
   Solution: For each File after someone acquires is create a corresponding .owners file for it, where it stores the number of owners. If owners>0 then this is private file. .owners file is updated upon each file acquisition or release.

2. **Problem**: Separating files for each Peer within the same PC.
   Solution: For each ***/addfile*** that the Peer does create a separate myfiles-<IP>-<port> folder where all his files are stored, without affecting the GLOBAL copies.

3. **Problem**: Distinguishing various private message requests send by one Peer to another.
   Solution: Earlier I was using destination.port as the identifier for broadcasts. But since for private messages I can't do that, I made a another section in previous header for message_type.

4. **Unsolved Problem:** Globally searching for a File using ***/search*** in an unstructured network.
   Probable Solution: Similar to Problem 11 from Part 1 due to same reasons, I had no ***$O(n)$*** solution to it, and the only solution was ***$O(n^2)$.***

Part 3 –

1. **Problem:** I
   Solution: I