Chitty-Chat

1. We originally tried using an RPC without any streams, we made this work such that a client could send a message to the server and the server could send a response to that client. We had a lot of trouble getting to work with other clients, so we chose to change it to work with a bidirectional stream. This made it possible for the server to keep a list of client stream which it can broadcast messages over.

2. We make use of a Server-Client system architecture. This is done such that clients can join and leave at any time without other clients losing connection. That is achieved by having the server control the connected clients and how their message-, join- and leave-requests are handled.

3. We have implemented 3 different RPC's.

    a. We have implemented a "Join" RPC, which takes a requestMessage and returns a responseMessage. When the RPC is called the server will broadcast "X has joined" to all open streams and returns "X has joined" to the client that send the request.

    b. We have implemented a "Leave" RPC, which takes a requestMessage and returns a responseMessage. When the RPC is called the server will broadcast "X has left" to all open streams and returns "X has left" to the client that send the request.

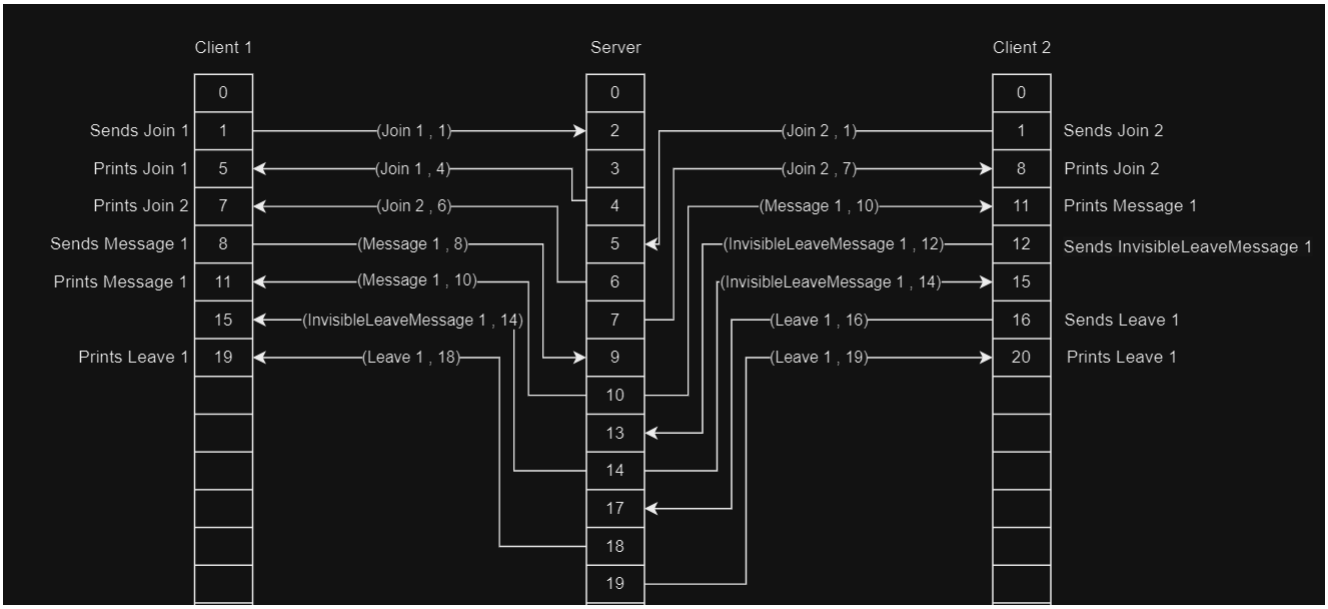
    c. We have implemented a "Publish" RPC, which takes a stream of publishMessages and returns a stream of broadcastMessages. When the RPC is called, the server will broadcast "X: <message" to all open streams.

4. We have chosen to implement Lamport timestamp, to do this we have implemented two functions to calculate the correct timestamp.

    a. When send a message, we need to increment by 1. We have created a function called "incrementLamport". This is done like the following: "lamport = lamport + 1"

    b. When receiving a message, we need to check whether the current or received Lamport is biggest and then increment by 1. We have created a function called "incrementMaxLamport". This is done like the following: "lamport = max(lamport, receivedLamport) + 1".

5. We have chosen the following sequence of interactions: Client 1 joins, Client 2 joins, Client 1 Publishes, Client 2 Leaves.



6. The following link is to our Git repository, where our source code is located: https://github.com/ITU-DISYS23-GROUPPandemicHalfPals/Hand-in3

7.

   a. To show that we are using go look at image 1, you can see that we run the “go run .” command.

   b. To show that we deploy client in different processes look at image 1, you can see that the clients are running in two different terminals.

   c. To show that we use the log package look at image 1, you can see that the timestamp is followed by the message which is how the log package work.

   d. To show that it is possible to have at least 3 clients look at image 2, you can see that three different processes are running at the same time and communicating with each other.

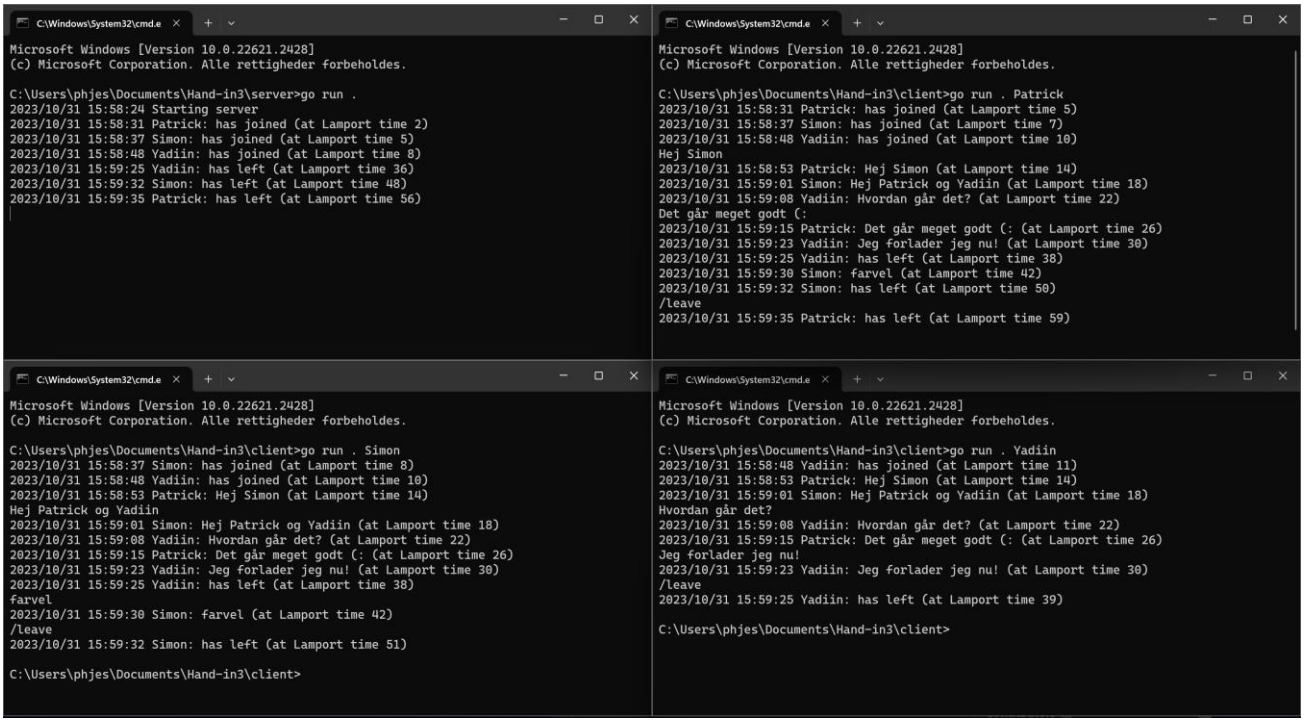
   e. To show that it is possible to join and leave the system at any time, look at image 1 and 2, you can see that the clients produce join and leave messages without crashing the server or other clients.

Appendix

(Image 1 – Sequence of interactions)

This shows that the sequence of interactions we chose in the diagram, matches the Lamport timestamp we expect.

(Image 2 – Multiple processes running)



This shows that multiple clients can run at the same time.