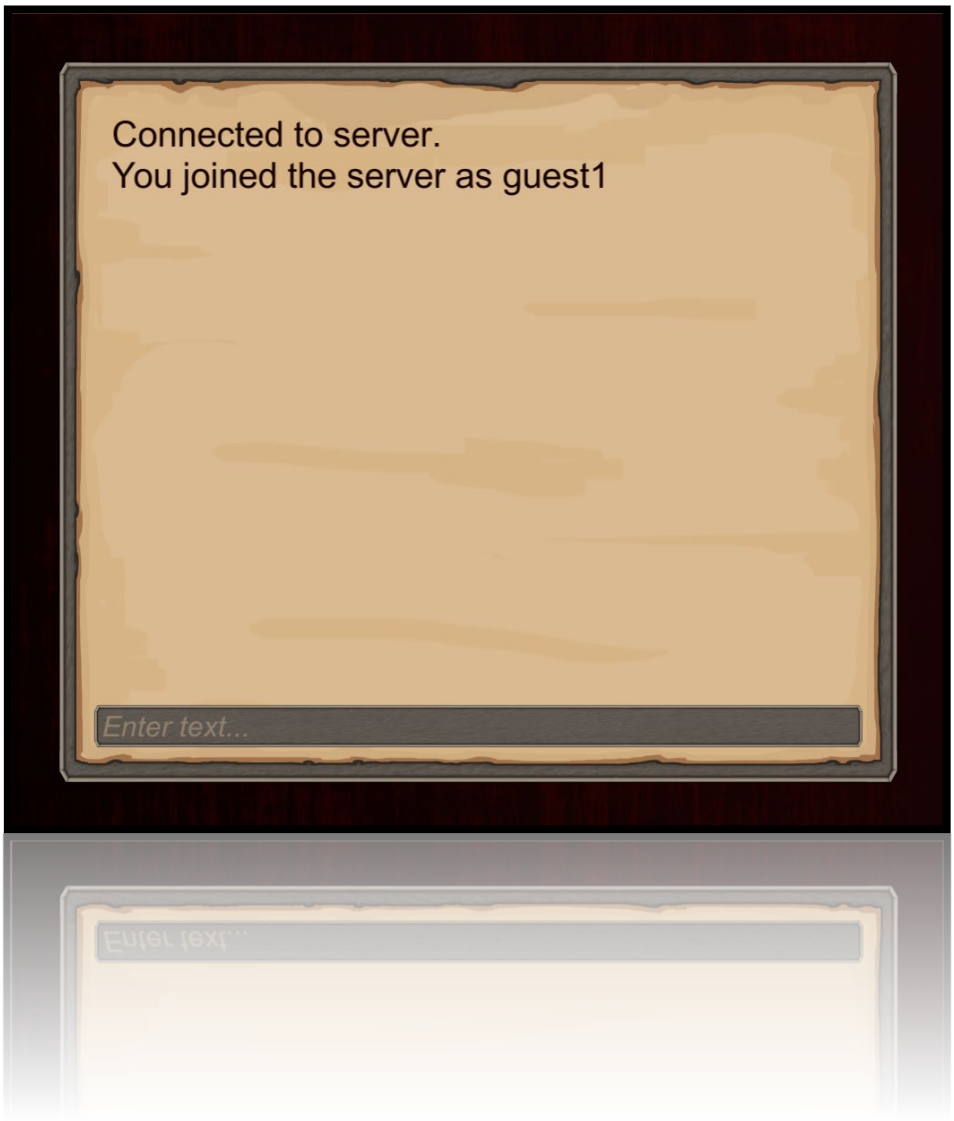
Networking - Assignment 2 - Implementing a chatbox



Getting started

In this second assignment you have to implement a chat client/server setup, *using the starting code* provided in the assignment zip file. Before you start on the actual assignment, follow the steps below to get acquainted with the setup.

Testing the setup with a single client

1. Open both the Unity client project and the command line server project.
2. Open and play the TcpChatClient scene in Unity and notice the error message.
3. Start the server, restart the client and notice how the client now prints 'Connected to server'.
4. Enter some text and notice how it ends up in the chatpanel. This is basically the same TCP Echo client server setup demonstrated during lecture 1, but now implemented in Unity.

Testing the setup with multiple clients

There are several ways to test this setup with multiple clients:

1. You can build the client and start it multiple times (but this costs a fair amount of time).
2. You can clone your project using ParrelSync (https://github.com/VeriorPies/ParrelSync).
3. You can duplicate the "client" within the current project (unfortunately this is not always possible).

For this assignment we are lucky and we can use the third option, open the TcpChatClientsQuad and test it out.
The biggest change with respect to the command line TCP echo client server demo, apart from the visuals, is that ***all*** 4 clients are responding: the server is serving these 4 clients "concurrently": there is *no* interaction between clients yet, but the server *is* responding to *all* of them and not putting any of them on hold while it loops to server another.

Getting to know the code

1. The client is implemented using a TCPChatClient script on the ChatPanel gameobject/prefab.
   This prefab is located under the main canvas in each scene. Each ChatPanel instance basically acts as a single client.

   Open the TCPChatClient script and see how everything you type is actually send to the server,
   echoed by the server and then read back by the client and printed to the chatpanel using the StreamUtil class.

   In other words, all the text you type is not being printed to the chatpanel directly, instead the reply from the server determines what you see in the chatpanel. Make sure you inspect the StreamUtil class and ask questions if anything is unclear. If you want you can also inspect the PanelWrapper class, but it is nothing more than a UI helper class.
2. Inspect the server to see how the 'concurrency' has been implemented using polling operations.
   Ask questions during lab about any part of the code that you don't understand.
3. Comment out the reply from the server, save all your work and run the client again.
   Terminate and restart Unity :).
4. Fix the client so that it no longer crashes if the server does not send a reply, by applying what you learned during lecture 2 about polling. What is the best place to implement this? Don't forget to save your work each time before testing, since Unity might have to be restarted again if you did it wrong.
5. After fixing the previous point, enable the reply from the server again and try to modify it in several ways. For example you could:
   a. Send the reply in reverse: Hello --> olleH
      (Every programmer should learn to reverse a string sometime right?)
   b. Add a timestamp on the server
   c. Add information about the sending client (e.g. remote IP address and port)
6. Last but not least, note how the code is shared between client and server: the shared scripts are located in the Unity project in the Scripts/shared folder and then linked into the server project. There are other ways that will be demonstrated in lectures to come, but for now this is the easiest way.

Now that you've worked through these steps, you are all set to start working on the actual assignment on the next pages.

Sufficient requirements (basic chat):

The biggest change in the code for this assignment, is that clients are no longer an island, but instead they can communicate with each other. Almost all work for this has to be done by the server.

Pass this part of the assignment, by fulfilling the requirements below:

- Clients can chat with all other clients.
  This means you must gather messages from every client and echo them back to all clients (including the sender).
- The server automatically generates a nickname for a joining client (Guest1, Guest2 etc).
- Appropriate feedback is provided where needed.
  - You joined the server as ...
  - Client ... joined the server.
- Every client message is automatically prepended by *the server* with the client's nickname (in other words the server is authoritative, the client does ***not pass*** its own nickname).

Tip 1 - A TcpClient uniquely identifies a client, use a Dictionary to map it to other data.
Tip 2 - Structure your server code like this:

```
private void run()
{
    Console.WriteLine("Server started on port 55555");

    listener = new TcpListener(IPAddress.Any, 55555);
    listener.Start();

    while (true)
    {
        processNewClients();
        processExistingClients();
        cleanupFaultyClients();

        //Although technically not required, now that we are no longer blocking,
        //it is good to cut your CPU some slack
        Thread.Sleep(100);
    }

}
```

Try to finish this assignment before the next lecture and work on the good, very good and excellent requirements after (see next page).

Good requirements (nickname administration):

- A client can ask the server to change its' nickname using a '/setname ....' command (or '/sn ....')
    - Nicknames must be unique, non-empty and (autoconverted to) lowercase
    - Appropriate feedback is provided by the server where needed:
        - This nickname is already taken etc
        - Client ... changed the nickname to ...
- If a client sends a '/list' command, the server replies with a list of names of all the connected clients.
- If a client sends a '/help' command, the server replies with information about all possible chat commands.

Very good requirements (whispering):

- Clients can whisper each other using '/whisper nickname message' (or '/w nickname ...'):
    - If the target exists, a message is sent:
        - to the target: <client...> whispers ...
        - to the sender (you): You whisper to <client...> ...
    - If the target does not exist, a message is only sent back to you:
        - Target ... does not exist

Excellent requirements (rooms):

- Clients can type '/join <roomname>' to automatically create and join a room.
- Clients only see messages from people in the same room
- Every client starts out in the room 'general'
- If a client sends a '/listrooms' command, the server replies with a list of all active rooms
  (there are no private rooms, rooms cannot be hidden, users cannot be kept out).
- If a client sends a '/listroom' command, the server replies with a list of all users in the current client's room
- Empty rooms do not *have* to be deleted.

Optional (as an additional challenge, but **not** mandatory for this part):

- You have a 2d array of rooms, each room has a different description (Hallway, entrance etc)
- Clients navigate from room to room by using 'north', 'south', 'east', 'west' instead of '/join'