When we started this course, I was interested in how multiplayer games are built to different uses/ends. Turns out, there is A LOT to cover, even more the deeper into the subject you go.

For the game, I didn't want my lack of experience in Unity to deter me from trying experimenting. So, I ended up with a very basic setup:
The player is a triangle that can move on one axis, that fires "bullets" in one direction. The goal is to hit a rectangle that goes horizontally back and forth.

We're using a client-host server model. The game follows an authoritative server model. This translates to major game events (in this case scoring) is validated by the host. This includes the position of the "goal", the rectangle the players are aiming to hit.

Player positions, and the goal, are synchronized with Unitys "Netcode for gameobjects" package, i.e. Network Object and Network Transform.

I wanted to try having the instantiated bullets be handled by clients through RPCs. The client that fires a bullet has a separate bullet that updates the score when it hits the goal. The client sends an RPC that creates a bullet without the scoring logic on each other client. The idea was that I want to avoid having the host fire the score-managing logic multiple times each time a bullet hit the goal. In this tiny project we might not notice much delay, but on a much larger scale the principle of having each client tell the host when to reevaluate score could create delays (especially if other game events did the same). This solution does, of course, create room for cheating. More on how I handled that issue later.

For my chat I settled for having messages sent as RPCs and then displayed on all clients.

A list of active players is kept, along with another list containing PlayerInfo objects that stores each player's score. When a client connects/disconnects, they're added/removed as intended. The host updates, verifies and syncs these scores across clients. I wanted to have active scorings listed on each client aswell, but my inexperience with Unity (specifically UI) forced me to drop that feature. I left it in, since I had the clients add themselves through the "ScoreBoard UI" script.

My biggest hurdle to tackle in this project was the way certain Networked objects were affecting my scripts timing in Start. At that time, all managers were null when initialized as Singletons in OnEnable(). This was most apparent when tracking players connecting through the client connect callback. I made an ugly solution to this by having affected scripts wait for the singletons to not be null until they proceeded. There must be a better way to deal with this other than using Coroutines.

The other big problem I had was with how the client's position was handled by the server. Basically, the movement command clashed with the server's currently held position of the client, leading to the client telling the server its model moved, and the server retroactively snapping that model back. To fix this I had clients keep a predicted position of themselves, which is used when instantiating bullets, and to lerp towards when there's no input from the client on a set frequency (I landed on every .01 seconds in the end).

Something we had drilled into our skulls from when the first course in Unity was key throughout building this system: Make sure to debug whatever you expect to have a value. In this project, it was interesting to see the way the host and connected clients communicated.

Since my game is very light on features, I didn't get to solve any desynchronization- or performance-issues. I did, however, in a way handle desynchronization of scores through the to-be anti-cheat measure:

This is where I put the bulk of my time into. I made a consensus-based verification system, where the server crosschecks the score of whichever client that just claimed it had scored. After the score is increased on each client, it is then crosschecked between the score held on each client. If the score isn't the same on all clients, it is set to the value held by the majority, and then updated as such on each client.

```
private void VerifyAndSetScore(string playerName)
{
    if (!_scoreReports.ContainsKey(playerName)) return;

    List<int> reportedScores = _scoreReports[playerName];
    int verifiedScore = reportedScores.GroupBy(s :int => s) // IEnumerable<IGrouping<...>>
        .OrderByDescending(g :IGrouping<int,int> => g.Count()).First().Key;

    PlayerInfo player = PlayerScores.Find( match: p :PlayerInfo => p.playerName == playerName);
    if (player != null && player.score != verifiedScore)
    {
        Debug.Log($"Score discrepancy detected for {playerName}. Setting score to {verifiedScore}");
        player.score = verifiedScore;
        UpdateScoreClientRpc(playerName, verifiedScore);
    }
}
```

After this project, I've learned so much about how client-server traffic works. But the biggest take-away I have from this is optimization is everything. I had the naive notion that everything going on in a multiplayer game was synchronized, and that good netcode could offset the serverload that would apply. It is only after working with netcode myself that I understand that there are multiple ways of making sure everyone playing the same game sees the same things on their screen. After this course, I've taken notice of how different studios have handled latency (I tried lowering my connection speed briefly to see what happened when playing different titles. Both

League of legends and World of tanks are using extrapolation. Escape from Tarkov, on the other end, cuts any and all movement).
Since I didn’t manage that many different gameobjects or players, I’ve not tried to optimize/handle mass movement, network interruptions and so on. This is something I one day definitely will try out on my own time.