CMP303 Presentation

3D Arena Battler

Architecture

- I went with a server-client architecture.
- One client hosts the game and their client gets authority over the game, runs all checks and dictates positions of all clients.
- Clients send their positions to the server, server sends back all other positions and corrects their own which may be required if network conditions are bad.
- Preferable to peer to peer in this case as I wanted the number of players to be scalable.

Network Protocol

- I wrote the program in C# using the .NET sockets API.
- I went with UDP only.
- TCP has some advantages, but the disadvantages of each packet requiring arrival and processing when I only need the most recent information for position updates made it a burden.
- I could have used it for game state changes, but I didn’t want to go through the effort of setting up an additional connection when I could achieve the same result through the existing one.
- So just using the unreliable UDP, I had to solve 2 big problems, no guarantee of arrival, and insuring idempotency.

Acknowledgement Protocol

UDP doesn’t handle packet loss, so an acknowledgement protocol was required.

The code displayed here is a task that runs every time a packet that requires an acknowledgement is sent.

It checks the list of acknowledgement tokens the UDP socket has received and if any of them match the token of the packet, it ends the task.

Otherwise, it keeps sending the packet until it does.

The mutex is required to protect the list object, as the receive function needs to add tokens to it as well, which could very well be happening at the same time, due to the asynchronous aspects.

```
public void AcknowledgementPacketSender(short token, byte[] stream, string ip, int port) {
        bool acknowlodgementArrived = false;
        double timeSinceStreamLastSent = timeElapsed;

        while (!acknowlodgementArrived) {
            foreach (var tokens in acknowledgementTokens.ToList()) {
                if (tokens == token) {
                    //Protect list before removing token.
                    tokenListMutex.WaitOne();
                    acknowledgementTokens.Remove(tokens);
                    tokenListMutex.ReleaseMutex();
                    acknowlodgementArrived = true;
                }
            }

            //If Token didn't arrive before acknolodgementTimeToResendPacket passed, send again.
            if (timeElapsed > timeSinceStreamLastSent + acknolodgementTimeToResendPacket) {
                GD.Print("Acknolodgement not arrived, resending packet: " +
Packet.GetPacketTypeFromStream(stream));
                udpClient.Send(stream, stream.Length, ip, port);
                timeSinceStreamLastSent = timeElapsed;
            }
        }
    }
```

Idempotency

Idempotency means only applying an operation once no matter how many times it's called. This is important with the risk of packet duplication UDP doesn't inherently handle.

Each packet I send has a type, and depending on the type, I can vary how it's handled.

Most of my idempotency revolves around time. I create a timer and sync that timer across each client every few seconds, both to insure a synced time but also to make sure the connection is still active.

If the timestamp on a packet is higher than the highest timestamp received previously, I know it's the most recently sent one, and can be safely process.

```
TimeSyncServerPacket receivedPacket = new TimeSyncServerPacket();
receivedPacket.Deserialise(receivedResults.Buffer);

float currentTimestamp = (DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond) * 0.001f;
timeElapsed = receivedPacket.serverTime + (currentTimestamp - receivedPacket.serverTimestamp);
```

```
if (gameStarted) {
    ServerPositionUpdateClientPacket serverPositionUpdateClientPacket = new
ServerPositionUpdateClientPacket();
    serverPositionUpdateClientPacket.Deserialise(receivedResults.Buffer);

    Player clientPlayer = players[serverPositionUpdateClientPacket.playerId - 1];

    try {
        if (clientPlayer.timeLastPositionReceived < timeElapsed) {
            clientPlayer.timeLastPositionReceived = timeElapsed;
            clientPlayer.timeSinceLastPositionUpdate = 0;
            clientPlayer.mostRecentReceivedPosition = serverPositionUpdateClientPacket.position;
            clientPlayer.mostRecentReceivedVelocity = serverPositionUpdateClientPacket.velocity;
        }
    } catch (NullReferenceException e) {

    }
}
```

Receiving Packets

For receiving packets, I used an asynchronous method. C# has the async and await keywords that streamline the process of setting up multithreaded operations.

When I create my UDP client, I also start this ‘Receive’ task. The task calls ‘udpClient.ReceiveAsync()’ with the await keyword, which means the method itself will be stalled until it completes so I can process the packet afterwards safely, but it does not stall the thread, so it can run other operations.

My initial tests with this allowed me to receive 10k+ packets a second when I had one instance send the other packets in an infinite loop.

```
public async void Receive() {
    while (udpClient != null)
    {
        UdpReceiveResult receivedResults;
        try {
            receivedResults = await udpClient.ReceiveAsync();
        } catch (ObjectDisposedException e) {
            GD.Print(e.Message);
            continue;
        } catch (SocketException e) {
            GD.Print(e.Message);
            continue;
        } catch (NullReferenceException e) {
            GD.Print(e.Message);
            continue;
        }

      //Proceed to process packet....

    }
}
```

Prediction/Linear Interpolation

I used Godot for creating the game, and due to not finding a more elegant way of synchronising the positions of the player physics bodies, I decided to just modify their transforms.

The last column represents their positions, which I perform linear prediction and interpolate that in one line of code.

GlobalTransform.origin represents the bodies current position.

The arguments of the LinearInterpolate function represents where I predict them to be, and .5f to get the halfway between the 2 points.

```
private void MoveCharacterNetworked(float delta)
    {
        timeSinceLastPositionUpdate += delta;

        //Prediction + Linear Interpolation
        GlobalTransform = new Transform(
            GlobalTransform.basis.Column0,
            GlobalTransform.basis.Column1,
            GlobalTransform.basis.Column2,
            GlobalTransform.origin.LinearInterpolate(mostRecentReceivedPosition +
(mostRecentReceivedVelocity * (float)timeSinceLastPositionUpdate), 0.5f)
    );
}
```

Timestamp As Trigger For Event

In addition to using the timestamp as a check for insuring idempotency, I also use it as a trigger for starting the game to make sure each client begins at the same time.

This is achieved through a task. The server sends each client the start time, and they all create their own tasks to load the level at that time. If the timers are synced, they'll all start the game at the same time.

Idempotency in this case is insured by storing the task itself as a reference, so if a duplicate packet arrives, the program simply checks if the task exists already.

```
public void ServerSyncStartGame() {
    ServerStartGamePacket serverStartGamePacket = new ServerStartGamePacket();
    serverStartGamePacket.acknowledgementToken = (short)rnd.Next(short.MaxValue);
    serverStartGamePacket.startTime = timeElapsed + 5;
    serverStartGamePacket.networkId = networkId;

    SendToAllConnections(serverStartGamePacket.Serialise());
    SendAssignPlayerId();
    BeginStartGameTask(serverStartGamePacket.startTime);
}
```

```
public void StartGame(double startTime) {
    while (timeElapsed < startTime) {
        //Do nothing, wait
    }

    sceneSwitcher.LoadGame();

    //Create player objects.
    players = new Player[playerCount];

    for (int i = 0; i < playerCount; i++) {
        players[i] = playerScene.Instance() as Player;
        players[i]._Ready();
        AddChild(players[i]);

          players[i].Translate(new Vector3(i * 5, 0, 0));

        players[i].playerId = i + 1;

        if (players[i].playerId == playerId) {
            players[i].CameraVisible(true);
            players[i].canMove = true;
        } else {
            players[i].CameraVisible(false);
            players[i].canMove = false;
        }
    }

    gameStartedMutex.WaitOne();
    gameStarted = true;
    gameStartedMutex.ReleaseMutex();

    gameStartTask.Dispose();
    gameStartTask = null;
}
```

Clumsy Tests

- The demo video I created showed the program being demonstrated with Clumsy.
- There was visible latency with some of the collisions, but no failures since the launching packets require acknowledgements.
- Position handling worked really well, with minimal rubber banding since only the most recent position packet is processed.
- The program ran very well with fairly tough parameters (Outbound and Inbound):
  - Lag: 100ms
  - Drop: 10%
  - Throttle - Timeframe: 30ms, Chance: 10%
  - Duplication: 10%
  - Out of order: 10%

Time Sync Analysis

I didn't like the way I synchronised the timers since it relied on the times on both machines being identical, which was never a guarantee. Here's how I would have changed it.

How I synced the timers

- Send a packet with the server time and current time to the client.
- When the client receives it, check the current time, and compare it to the received time, then add that to the server time and use that time as the new time.

How I'd change it

- Start timer on client side.
- Send message to server.
- Have server send back its time.
- Accept servers time and set client time to servers time plus half of timer.
- Iterate off of this idea by instead first creating an average latency, adding that to the server timer instead.