Advanced Games Engineering Report

Alfie Thomasson
40312683

Lag Compensation

The Software Delivered

The software created is a demonstration of lag compensation techniques, built in Unity and using the Mirror API to manage networking. The software features the following things:

- The ability to host a dedicated server with which other users can connect to, from both inside and outside the local network.
- A lobby system that clients load into initially, and a ready up command that starts the game automatically once all players are ready to play.
- Customisable name tags to identify each player, which displays both in the lobby and in game.
- A simple sandbox environment that players can move around in, and see the movements other players make.
- Simple first person shooter abilities, the ability to shoot and damage other players and to reload your weapon.
- Interpolation, a toggleable lag compensation technique that smooths player movement.
- Extrapolation (or Dead Reckoning), a toggleable lag compensation technique that predicts the next movement of a player in order to smooth movement.
- An automatic mode which will turn Interpolation and Extrapolation on depending on/off the latency of the connected player.
- Two different rewind techniques, Frame based and Latency based, that aim to aid hit detection using a server authoritative method.

How to use the Software

Loading

In order to test the software properly, 3 instances of it must be opened and ran at the same time. One of these will act as the dedicated server, and the other two will be the players. Players can connect from outside the local network, although this will not work in the submitted build as it requires port forwarding.

Name Input

On software start, enter the desired player name and continue. If testing this with multiple instances on the same computer, please be aware that the name tags will only use the last inputted name due to it reading from Unity player preferences, this will work correctly with players connecting on other machines however. If you wish to host the server, pressing Host Server will begin this which will allow other instances

to connect to it, this may ask for a firewall popup to allow this. Currently, it is set to host to the *localhost* address, as otherwise it would require port forwarding. On the other clients, if they wish to join the game they can press Join Lobby and it will ask for an IP Address input. If connecting to a locally hosted server, this will be *localhost*, and from outside the local network it will be your public IPv4 address.

Lobby

Once successfully connected, the players will see a lobby screen with simple game instructions, the other players connected, and a ready up button which will tell the server they are ready to begin the game. Once the minimum number of players have connected (currently set to 2) and readied up, the server will begin a countdown that is displayed on screen, and when that reaches 0 the game will begin. At any point a player can unready and the countdown will stop.

Basic Game

Inside the game, players will load in and should be able to move around, look around, and have these movements shown for all other players as well. Name tags should also be shown above players heads to identify who's who. All players should be able to fire their weapon with left click, which will display a ray and a hit marker for all other players as well. Each player's weapon has an ammo count, and players can reload their weapon with the R key. If they choose to shoot other players, they can deal damage to others and once they have taken enough damage, other players will die, have their health reset, and deaths and kills incremented for the targeted player and shooter respectively. On death, a player should respawn, although this doesn't work all the time in the current build and sometimes they may respawn in the same place they have died, or be shown to respawn somewhere else but have them displayed incorrectly, although this is fixed by them moving.

Lag Compensation

Each player alongside the server should have a selection of tick boxes that represent the lag compensation. On the software that is hosting the server, you can tick these boxes to enable them across all the players. It is recommended not to use Historical Lerp or Rewind Hit Detection at the moment, as these are not working as intended. The others, Interpolation, Extrapolation, Frame Rewind, and Latency Rewind, will be covered in more detail later, but can be toggled on or off to see results. The automatic switch may also be enabled to enable/disable interpolation and extrapolation based on the connected player's latency. The capsules represent the following thing when using the frame rewind/latency rewind:
Red - Where the shot player was on the shooter screen at time of firing.
Yellow - Where the server has calculated the shot player was at time of firing.
Blue - Where the player was most recently on the server.

In order to test lag on the local network, I used the Clumsy application. The fork I was using that worked well for me was this one here: https://github.com/IntouchHealth/clumsy/releases

How It Works

Synced Movement

For the movement and rotation syncing, each player has a simple Character Controller script attached which will allow them to move normally on their client, without the server checking if it's a legal move. Each player has an additional script attached, known as *SyncPosition*. This has a synced variable known as *SyncPos*, which is synced across all clients. When a player moves, it sends its updated position to the server. The option to add a threshold to check if a player has moved more than a specified amount is here as well, which may be used if the network is particularly strained to minimise network updates. This is currently disabled in the demonstration software for more fluid movement. If a player has moved, it sends a command to the server to update the *SyncPos* variable with the player's new position, which will then update that variable on other instances of the player object on the other clients. In order to have players movements reflected on other clients, on each other instance of the game except for the local players, they are moving players locally towards their synced position variable, which reflects the movements players make. The same principle applies to the rotation, which works in a near identical fashion to the movement. The code to represent this is shown in Appendix 1.

Interpolation

The interpolation lag compensation technique is one of the most effective in smoothing player movement over the network, and is one that I implemented in my software. While simple, it has significantly effective results. If all players and networked objects only rendered whenever they received an update from the server that they have moved, they would look extremely choppy and jump between positions statically. A solution to this is interpolation, whenever a player receives an update from the server that a player has moved, it uses the Vector3.Lerp function to interpolate between the position of the player currently on screen, and the new position received from the server. This smooths player movement overall and removes most of the choppiness from movement, making movement for networked entities appear far more fluid. The code to represent this is shown in Appendix 2.

Extrapolation (Dead Reckoning)

This technique is in essence, predicting the future movement of players to help smooth movement. It works by tracking the previous movement a player made, and using that along with the new position updates to create a directional vector that the

player is projected to move along. Using this, the player is moved to their new position plus an estimate of where they will be next frame. This additionally helps smooth player movement due to the amount of server ticks in between player directional changes. The code to represent this is shown in Appendix 3.

Automatic

The automatic toggle is simple, but it is essentially constantly checking the local players ping value and if it exceeds a threshold (usually set to 100), it automatically enables extrapolation, and disables it without. This is because extrapolation tends to work more efficiently when a player has a higher latency due to it guessing positions to make up for lost updates. The code to represent this is shown in Appendix 4.

Rewind Time

In my program, I have partially implemented two different types of Rewind Time, latency and frame rewind, which I will explain individually. The concept of rewind time is the same for both of these however, which will be explained here.

Essentially, when a player moves and sends an updated position to the server, which then sends it to each other client, there is a delay in transition as a result. What this means is that other players see everyone else slightly in the past, and most up to date positions are stored on the server. In a server-authoritative system where the server checks if a player has successfully shot another player, this can cause issues. One solution to this is to use a rewind system, which works like this: When a player fires its weapon, it sends a command to the server that it has fired, along with its position, and where it was aiming. It also sends information on its current state, which differs for latency and frame rewind, as this will be used to calculate how far back the player is perceiving the environment in relation to what is currently on the server.

Once the server has received the required information, it calculates how far back the shooting player was in relation to the most recent server state, and moves each player back to where they would have been on the local players screen, runs the hit detection raycast, moves the players back to where they were, and outputs the hit information back to the player.

Sadly, in its current form the hit detection on the server is still not working accurately due to the way the movement syncing has been implemented, which is something that will be discussed in the critical evaluation section. However the rewind can still be viewed, using three coloured capsules in the scene. When one of latency rewind or frame rewind is enabled, a red capsule, yellow capsule, and blue capsule represent the following things:

Red - Where the shot player was on the shooters screen at the time they fired.

Yellow - Where the server has rewinded the shot player to in order to calculate hits on the server.

Blue - Where the shot player is on the server, e.g it's most recently updated position.

Latency Rewind

For the latency rewind, when the player shoots and sends information on its position and where it's aiming, it also sends information on its current latency and ping. This is used to calculate where the player would have been perceiving other players when it shoots. When this is enabled, the server starts tracking the positions of each player on every server tick for two seconds of time, which it can use to rewind back to where players were, using a List of a custom class that has each player with a list of its own to track its positions over the time. It essentially takes the current latency time, and attempts to predict how far back the last positional update would have been sent to the shooting player, and based on this rewinds other players back to where they should have been for the shooting player, calculates hits, and moves them back. In my program, this does not work quite as well as intended, and while it does track back in time, the calculation does not quite rewind far enough and is only partially working. The code to represent this is shown in Appendix 5.

Frame Rewind

The frame rewind technique works similarly to the latency, in that it tracks a list of previous positions and rewinds to a calculated point. However this does work differently, in the way it tracks positions and calculates rewinds. On the server, each time a positional update is received, it calls the *UpdateFrameData* function on each player, which adds their current position to a dictionary and uses the current frame count of the server as the key. It then sends this key to each player along with the movement data. What this means is that when a player shoots, it sends the current frame key it is on to the server alongside the shot direction, as previously described. The server uses this frame key to retrieve the positional data for all objects at the time the player shot, and moves all players back to where they were at the time, runs the raycast, and moves them back. This does work significantly better than the latency rewind, but is still not 100% accurate. The code to represent this is shown in Appendix 6.

Lobby System

The lobby system in the software is another element that is worth discussing. It uses a custom network manager which uses Mirror's base network manager and overrides some of the methods. The server can be started by simply calling *StartServer* on the network manager, and can be connected to from there. How the lobby works in essence is as follows:

- The player inputs the desired ip address, and presses join.
- In the network manager, *OnClientConnect* is called on the client.
- On the server, *OnServerConnect* is called which checks if the server is full and rejects the player if so, *OnServerAddPlayer* is called, which creates a *NetworkRoomPlayerLobby* object on both server and clients, this is the lobby

menu that the player will see and becomes the local player, meaning they have authority over it.

- When this starts, it is added to a list that will be the list of players on both the server and client network managers, in addition it also sets the players name on the server.
- When other players join, the *UpdateDisplay* function is called on each player, which keeps the UI up to date and displays the correct player names. Whenever other players change their ready status as well with the button on screen, the server will check if all players are ready in *StartGame,* and if so, begin the countdown until game start, which is done in the *Update* function, which only runs on the server.
- Once the countdown has ended, the server calls the *ServerChangeScene* function, that loops through all players and replaces the game object that represents them with the new prefab *NetworkGamePlayerLobby*. This is a placeholder object that will serve as the player while it spawns the actual player objects.
- On scene change, *OnServerSceneChanged* is also called, which will spawn the *SpawnSystem*, this will be used to spawn our player objects once everyone has loaded into the game.
- When a player has successfully changed a scene, *OnServerReady* is called on the network manager, which invokes the SpawnPlayer action that has had the *SpawnPlayer* function added to it on the player spawn object.
- In the *SpawnPlayer* function, the server creates a player game object for each player, and assigns it as the local player. From here, the player should be able to play the game normally.

The code to represent this is shown in Appendix 7.

Critical Evaluation

Initial Thoughts

Now that I am at the end of the project, I can safely say that it has been a real learning experience. At first, I thought this would be a challenge to take in, however I didn't expect the initial learning curve to be so steep. As I was, for the most part, totally new to networking in general, learning how Mirror worked and discovering functions or methods that were previously unknown, was a lot more work than anticipated. My initial research into the proposed techniques used to mitigate lag, while definitely useful, I do feel was a bit too general and could have used more depth into the implementations using c# and specifically mirror. With regards to the

project itself, I do think that I could have improved my overall time management considerably, as I spent a lot of time unsure of my next steps.

Relating back to the initial research perhaps not being as strong or relevant as it could have been, I do think this is reflected in the code I have written, as a lot of it is not optimal and could probably be significantly refined. In particular, I think my project suffered mainly from the system I used for transmitting movement and syncing player positions, as I created it before I implemented any rewind-time or server authoritative methods. This resulted in major problems when I came around to implement rewind time, as it interfered with the hit detection. For implementing the client-side techniques interpolation and extrapolation, it actually worked fairly well, and was easy enough to introduce these techniques to. These techniques I also was happy with, and while it took awhile to figure out a working solution, the final result is simple and effective.

Aims

As part of my evaluation, it's important to relate back to my initial aims and how I feel I have accomplished them, or not quite lived up to them. The initial aims were as follows:

The general aim was to gain an understanding of and implement techniques to compensate for high latency and lag in video games.

- Research different methods of lag compensation and topics surrounding the subject.
- Implement client-side lag compensation techniques, such as Interpolation and Extrapolation.
- Implement server-side techniques such as rewind time to mitigate lag.
- Create a functioning server and game to test quality of lag compensation techniques by inducing lag, either through external software or manual code.

My initial aim to gain an understanding of and implement lag compensation techniques, I actually feel on the whole I am pretty happy with my learning from the project, and have ended up with a piece of work I am happy with and that reflects a lot of my learning. I have managed to implement most of my initial plans of lag compensation techniques in some form, and with varying success. The research i've ended up conducting has been an extremely interesting and informative insight into various methods of lag compensation, and I can safely say my number one takeaway is that although a lot of common techniques exist, it seems that there are so many different ways of implementing them. This is actually something that I didn't anticipate as much as I should have done, and did pose an issue as I found it challenging to move some techniques and ways of implementing methods into my code, due to differences in the network frameworks.

On the implementation of client-side lag compensation techniques, overall I was happy with this as previously mentioned, although my implementation is far from perfect. The server side techniques again, as previously mentioned, I did think I fell down on, and if I were to re-do this project or work on it again in the future, I would definitely start out by ensuring the setup was in place to work with a server-authoritative framework. With regards to creating a functioning server and a game software to test the quality, I am fairly happy with how I achieved this. Using the clumsy application to test lag also worked well, and I had no issues using the application. The game and server function well and can host players from outside the local network, and I feel this is a strong point of the software. In addition, I implemented the lobby system and hit markers to aid gameplay and improve the software, and this is something I am very happy with.

Useful Sources

Here I will link a few of the sources I found useful and/or aided me in writing the code for this project.

- https://www.construct.net/en/tutorials/multiplayer-tutorial-concepts-579/lag-compensation-8
- https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization
- https://en.wikipedia.org/wiki/Lag
- https://www.youtube.com/watch?v=2kIgbvl7FRs&ab_channel=FullstackAcademy
- https://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking
- https://www.gamasutra.com/view/feature/131781/the_internet_sucks_or_what_i_.php?page=4
- https://www.gabrielgambetta.com/lag-compensation.html
- https://www.youtube.com/watch?v=Nn5cG8IXg3k&ab_channel=ShrineWars
- https://www.youtube.com/watch?v=Fx8efi2MNz0&ab_channel=DapperDino
- https://www.youtube.com/watch?v=KwyTPIRKJHY&ab_channel=SolidJuho
- https://valllar.wordpress.com/2018/09/28/45/
- https://learn.unity.com/tutorial/let-s-try-shooting-with-raycasts#5c7f8528edbc2a002053b469
- https://twoten.dev/lag-compensation-in-unity.html
- http://beej.us/guide/bgnet/pdf/bgnet_a4_bw_2.pdf
- https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

Appendix

1 - Synced Movement

```
//Set new position
curTransform.position = syncPos;
```

```
//Called on server when player has moved
[Command]
2 references
void CmdSendPosition(Vector3 pos)
{
    //Updates synced position variable, updates on clients as well as it is a syncvar
    syncPos = pos;
    //Updates the past move based on movement
    pastMove = syncPos - curTransform.position;
    //Updates all frames in hit tracker
    hitTracker.UpdateAllFrames();
}
```

```
//Checks for movement on player and updates others if so
[ClientCallback]
1 reference
void RpcSendToClient()
{
    if(isLocalPlayer) // If this is local player
    {
    if(!useDeadReckoning)
    {
    if(Vector3.Distance(curTransform.position, lastPos) > threshold) // If threshold is exceeded, can add a threshold for minimum movement if network is strained
    {
        //Send position to the server
        CmdSendPosition(curTransform.position);
        //Updates last position to current position
        lastPos = curTransform.position;
    }
    }
    else
    {
        CmdSendPosition(curTransform.position);
        lastPos = curTransform.position;
    }
    }
}
```

2 - Interpolation

```
//Lerp normally to new position
curTransform.position = Vector3.Lerp(curTransform.position, syncPos, Time.deltaTime * lerpRate);
```

3 - Extrapolation

```
//Lerp position, using the estimation of next position as well
curTransform.position = Vector3.Lerp(curTransform.position, syncPos + (pastMove * moveSpeed * Time.deltaTime), Time.deltaTime * lerpRate);
//Updates previous move to equal the new move
pastMove += (pastMove * moveSpeed * Time.deltaTime);
```

4 - Automatic

```
//If automatic is selected
1 reference
void CheckPing()
{
    //Check if ping is over 100
    if((NetworkTime.rtt * 1000) > 100)
    {
        //If it is, turn extrapolation on
       if(!useDeadReckoning)
       {
           CmdChangeExtrapolation(true);
       }
    }
    else if(NetworkTime.rtt * 1000 < 100) // If ping below 100
    {
        if(useDeadReckoning) // Then turn extrapolation off
        {
            CmdChangeExtrapolation(false);
        }
    }
}
```

5 - Latency Rewind

```
//This gets the position in list of gameobjects that the player should have been viewing at the time they fired
//Currently does not work as intended
int calculatedPosition = (int)Mathf.Floor((float)(latency * 4) / (float)serverTickRate);
//If higher latency than tracked, reset to max.
if(calculatedPosition > 119) {calculatedPosition = 119;}
//Subtract calculated position from 119, this is because the oldest values in the list are at list[0]
calculatedPosition = 119 - calculatedPosition;
Debug.Log("Calculated Position = " + (int)calculatedPosition);

//Loops through all tracked players
foreach(TrackedPlayer player in trackedPlayers)
{
    //And moves them to calculated position for shot detection
    player.playerBody.transform.position = player.positions[calculatedPosition];
}
RaycastHit hit;
//Shoots ray
Physics.Raycast(rayOrigin,rayForward, out hit, 500.0f);
```

6 - Frame Rewind

```
//Calls function SetNewTransform for player, should move it back to where it was when player shot
player.playerBody.GetComponent<SyncPosition>().SetNewTransform((int)frameTime);
//Shoots ray
Physics.Raycast(rayOrigin,rayForward, out hit, 500.0f);
```

```
//Resets player back to where they were
1 reference
public void ResetTransform()
{
    //Sets position to saved position
    curTransform.position = savedPos;
    //Disable override
    overRide = false;
}
```

```
//Called on the server to update frame data for use in rewind time
//Tracks posiions of players in a dictionary and uses the framecount on the server as a key
[Server]
1 reference
public void UpdateFrameData()
{
    if(!overRide) // If not overriding
    {
        if(Keys.Count > 120) // Check if Keys list is full
        {
            //If so, remove oldest variables from lists
            int key = Keys[0];
            Keys.RemoveAt(0);
            FrameData.Remove(key);
        }
        //Tests if value already exists
        Vector3 test;
        if(FrameData.TryGetValue(Time.frameCount, out test))
        {
            FrameData[Time.frameCount] = curTransform.position; // Updates dictionary
        }
        else
        {
            FrameData.Add(Time.frameCount, curTransform.position); // Adds to dicitonary if doesnt exist
        }
        //Adds the key to the list
        Keys.Add(Time.frameCount);
        //Sends the updated frame id to all clients
        RpcUpdateFrameId(Time.frameCount);
    }
}
```

```
//Sets the transform of player to the point in the framedata dictionary
//Used for rewind time, setting player back to where they were at a point
1 reference
public void SetNewTransform(int frameid)
{
    overRide = true; // Sets override to true
    //Override is used to help consistency of shooting and raycasts (wip), still does not always work
    //Updates the saved position to return to afterwards
    savedPos = curTransform.position;
    //Sets position to rewinded pos
    curTransform.position = FrameData[frameid];
}
```

7 - Lobby System

```
0 references
public void HostLobby()
{
    //Start server
    networkManager.StartServer();
}

0 references
public void StopLobby()
{
    //Stop Server
    networkManager.StopServer();
}
```

```
//When client trys to conncet
0 references
public override void OnClientConnect(NetworkConnection connection)
{
    Debug.Log("Client Attempting to Connect");
    //Call base function
    base.OnClientConnect(connection);

    OnClientConnected?.Invoke();
}
```

```
//Called when something connects to server
0 references
public override void OnServerConnect(NetworkConnection connection)
{
    //If server full
    if(numPlayers >= maxConnections)
    {
        //Disconnect attempted connector
        connection.Disconnect();
        return;
    }
}
```

```
 //When server succesfully adds a player
0 references
public override void OnServerAddPlayer(NetworkConnection connection)
 {
    Debug.Log("Should spawn!");

    //Assign first joined player as lobby leader
    bool isLeader = RoomPlayers.Count == 0;

    //Instantiate room player object
    NetworkRoomPlayerLobby roomPlayerInstance = Instantiate(roomPlayerPrefab);

    //Sets this player as leader or not leader
    roomPlayerInstance.IsLeader = isLeader;

    //Adds instantiated object to the connection, it is now the player for that connection
    NetworkServer.AddPlayerForConnection(connection, roomPlayerInstance.gameObject);
 }
```

```
//Updates the display of tthe lobby
5 references
private void UpdateDisplay()
{
    if(!hasAuthority) // If this does not have authority, recursively check for the one that does
    {
        foreach(var player in Room.RoomPlayers)
        {
            if(player.hasAuthority)
            {
                player.UpdateDisplay(); //When player with authority is found, call this again
                break;
            }
        }
        return;
    }

    //Loop through all players
    for(int i = 0; i < Room.RoomPlayers.Count; i++)
    {
        //Updates player names to equal their names
        playerNames[i].text = Room.RoomPlayers[i].DisplayName;
        //Updates ready status
        playersReady[i].text = Room.RoomPlayers[i].IsReady ?
        "<color=green>Ready</color>" : "<color=red>Not Ready</color>";
    }
}
```

```
//Start game function to begin game starting
1 reference
public void StartGame()
{
    if(SceneManager.GetActiveScene().name == menuScene) // If scene is the menuscene
    {
        if(!IsReadyStart()) //Check if players are ready to start
        {
            //If not ready, countdown is not active
            countdownActive = false;
            countdownTime = defaultCountDown;
            for(int i = 0; i < RoomPlayers.Count; i++)
            {
                RoomPlayers[i].countdownActive = false;
            }
            return;
        }
        else
        {
            Debug.Log("Ready to start!");
            //Set countdown to active!
            countdownActive = true;
            for(int i = 0; i < RoomPlayers.Count; i++)
            {
                RoomPlayers[i].countdownActive = true;
            }
        }
    }
}
```

```
//Only runs update on the server
[ServerCallback]
0 references
void Update()
{
    if(countdownActive && SceneManager.GetActiveScene().name == menuScene) // Checks if countdown is active and scene is menu
    {
        //Decrements countdown time
        countdownTime -= Time.deltaTime;

        //Updates all clients with countdown time
        for(int i = 0; i < RoomPlayers.Count; i++)
        {
            RoomPlayers[i].countdownTime = (int)countdownTime;
        }

        //If countdown has ended
        if(countdownTime < 0)
        {
            //Change the scene
            ServerChangeScene("GameScene");
            countdownActive = false;
        }
    }
}
```

```
//Begin Scene change!
1 reference
public override void ServerChangeScene(string newScene)
{
    //If going from menu to game
    if(SceneManager.GetActiveScene().name == menuScene && newScene == "GameScene")
    {
        //Stop Countdown
        countdownActive = false;
        //Loop through all players backwards to stop errors occuring
        for(int i = RoomPlayers.Count - 1; i >= 0; i--)
        {
            //Get players connection
            var connection = RoomPlayers[i].connectionToClient;

            //Spawn intermediatery game prefab!
            var gameplayerInstance = Instantiate(gamePlayerPrefab);
            //Set display name to player name
            gameplayerInstance.SetDisplayName(RoomPlayers[i].DisplayName);

            //Destroy current game object attached to player
            NetworkServer.Destroy(connection.identity.gameObject);

            //And replace the player object with the new one!
            NetworkServer.ReplacePlayerForConnection(connection, gameplayerInstance.gameObject, true);
        }

        //Call base change scene
        base.ServerChangeScene(newScene);
    }
}
```

```
//When scene is changed, this is called ons erver
0 references
public override void OnServerSceneChanged(string sceneName)
{
    if(sceneName == menuScene) // check if current scene equals the menu
    {
        //Spawn the spawn system used for creating the proper player prefabs
        GameObject playerSpawnSystem = Instantiate(spawnSystem);
        //Spawns on the network server which spawns on clients as well
        NetworkServer.Spawn(playerSpawnSystem);
    }
}
```

```
//Called on server when client is ready
0 references
public override void OnServerReady(NetworkConnection connection)
{
    //Base method
    base.OnServerReady(connection);

    OnServerReadied?.Invoke(connection);
}
```