NETWORKING LAB 02

HANDS-ON

In the hands-on networking unit the student will start off looking at a HLAPI (High-Level API) which falls within layers 4 and 5 on the TCP/IP generic model. The student will be mostly protected from networking errors, and will get a grasp on how connections can be created between computers. The foundational knowledge of a server authorative networking state is also established, as well as client-side prediction and latency compensation.

The student will then transition into a LLAPI (Low-Level API) which resides within layer 2 and 3 on the TCP/IP generic model. The student will lose a significant portion of the protection provided from the HLAPI as they begin to work directly with socketing and commands.

Section Two: Taking It Into Your Own Hands!

In this section of the networking unit, the student will implement a client-side prediction setup, create their own character controller, and synchronize animations over the network. This will allow:

- Customizing character controllers over a network
- Smoothing out latency spikes
- Animation synchronization

The basic knowledge gained in this unit will be the foundation of the students learning as the students begins to work in a LLAPI (Low-Level API) where the student will directly handle sockets and other networking tasks.

The student should not hesitate to refer back to their notes from in-class discussions when creating the client-side prediction and latency compensation algorithms for this lab.

STEP ONE: SETUP

For the setup of this lab you have been provided with a template game. The game includes two scenes: A menu, and the main game world.

You are also supplied with multiple scripts: Lab02a_PlayerControl, and Lab02_DisableComponents.

There is a model (named UnityGuy) with the animations and transitions already setup and ready to use.

The Player prefab has been updated to reflect the new player that you will create.

Although the template game will run, you will have no control over the character if you attempt to run the game at this point in time (this is due to Lab02a_PlayerControl script being empty!)

Familiarize yourself with the animation controller setup (Static Assets > Prefabs > Common > Player V2 > UnityGuy).

STEP TWO: GETTING A “MOVE” ON

The first thing you will do is recreate a very simple character movement system and network it (instead of relying on the character controller that unity is built with!).

Open the Lab02a_PlayerControl script.

Add the using UnityEngine.Networking; statement, and change the extension from a MonoBehaviour to a NetworkBehaviour.

PSEUDOCODE

```
Create a state that will hold player position and rotation.
Create a syncvar with a local variable that will hold that state.

Start
	Initialize the state to default values
	Sync the player character to reflect the state the player is in.

Server Command - Initialize
	Initialize the state to default values

Update
	If the current object is the local player
		Create an array for every key the player can press (and get a result)
		Loop through that array, checking to see if the player is pressing the key
			If the player is pressing the key, command the server to move the player
	Sync the player character to reflect the state the player is in

SyncState
	Set the gameobjects position to whatever position is in the state
	Set the gameobjects rotation to whatever rotation is in the state

Command - Move on the server
	set the state to the result of the keypress

Move
	Keep track of delta x, y, z, and rotation y
		(This means the change in x, y, z, rotation y)
	Switch on the different types of keypresses that can happen
		If player pressed Q
			Decrease delta x by some speed
		If player pressed S
			Decrease delta z by some speed
		If player pressed E
			Increase delta x by some speed
		If player pressed W
			Increase delta z by some speed
		If player pressed A
			Decrease delta rotation y by some amount
		If player pressed D
			Increase delta rotation y by some amount

	return a new player state that reflects these changes
```

STATES!

If you remember from the in-class lecture, part of the idea behind network synchronization and client-side prediction is the idea behind the client sending to the server it’s input values. The server can then authenticate the client’s values, and send them back to the client to update the client. You will simulate a very basic form of this communication. To start, you will need to create a struct that can hold very basic information about the player such as the player’s position and rotation.

Inside of this struct you want to stick to basic types. This means that you don’t want to store a Vector3! Instead, store 3 floats.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
public class Lab02a_PlayerControl : NetworkBehaviour {
    struct PlayerState
    {
        public float posX, posY, posZ;
        public float rotX, rotY, rotZ;
    }

       // Use this for initialization
       void Start () {
```

```
        }

        // Update is called once per frame
        void Update () {

        }
}
```

Next, you will want the player to be able to store this information in a local variable. Declare a variable to store this struct. This variable will be what controls how the server (and other people connected to the server) see your character. In order to update this variable every single time it gets changed, you can add the [SyncVar] attribute to it. Again, the SyncVar attribute will broadcast changes to this variable to the server and connected clients.

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
public class Lab02a_PlayerControl : NetworkBehaviour {
    struct PlayerState
    {
        public float posX, posY, posZ;
        public float rotX, rotY, rotZ;
    }

    [SyncVar]
    PlayerState state;
}
```

INITIALIZING AND SYNCING

Next, you will want to initialize the state to something (preferably when the player first connects to the server!) You can break this initialization into its own method for readability, but more importantly so you can ensure that the server remains authorative over this information. What this means is that you want the **server** to initialize the player's state to some value and then the server can inform everyone of what those values are. If you allowed the client to initialize the value and then tell the server what the values are, then you are opening your game up to cheating.

Create the new function (InitState) to setup the default values of the player state. These values were hand-picked to work with the Main World scene.

```
    void InitState()
    {
        state = new PlayerState
        {
            posX = -119f,
            posY = 165.08f,
            posZ = -924f,
            rotX = 0f,
            rotY = 0f,
            rotZ = 0f
        };
    }
```

Now in order to ensure that the Server is the one that makes this call, you can add the [Server] attribute above the method. The [Server] attribute marks a function that is only allowed to be called on the server. This means that if a client attempts to run this function, nothing will happen! (Well – unity will give you a warning, but it won't run the function or crash the game).

The next method that you can add will be the method used to sync the player's actual model to the values stored inside of a PlayerState (in other words, set the players position and rotation to the values inside of a PlayerState).

You can call this method SyncState, and it is a simple method that imposes values onto the transform of the gameObject that the script is running on.

```
    void SyncState()
    {
        transform.position = new Vector3(state.posX, state.posY, state.posZ);
        transform.rotation = Quaternion.Euler(state.rotX, state.rotY, state.rotZ);

    }
```

Remember: The script has a local variable named state that contains the position and rotation information about the player!

Now that you have a way to initialize the state of the player, and then sync the player's game object with that state, add the Start function to call these two methods.

```
    void Start()
    {
        InitState();
        SyncState();
    }
```

MOVEMENT ALGORITHM

The next task is the most difficult in this step. You will need to create a movement method that will update the values of the players state. This method should accept (as a parameter) the player's previous state (the state the player is moving FROM), and the keyboard input from the player (what key the player is pressing). With that information, you will be able to determine how to adjust the CURRENT state of the player.

```
PlayerState Move(PlayerState previous, KeyCode newKey)
{

}
```

Inside, you will want to switch on the KeyCodes to adjust the player's theoretical position and rotation. Keep in mind, at no point in this method are we actually going to MOVE the player. Since this method will be used to update both clients AND the servers view of the player, you want to maintain strict control over all information. That is why you will be using a previous player state instead of just requesting transform.position from a gameobject.

In order to facilite this will you need a few variables to store temporary information. This temporary information will represent the CHANGE in position/rotation of the player. Generally when dealing with physics this type of change is called **delta**. Therefore, it would be most acceptable to name your variables with the delta signature.

```
    PlayerState Move(PlayerState previous, KeyCode newKey)
    {
        float deltaX = 0, deltaY = 0, deltaZ = 0;
        float deltaRotationY = 0;
    }
```

Now with some temporary variables to store information, you can switch on the KeyCode and begin to add some "physics" to your character! The physics at this point will just to adding a small amount of movement in whichever axis is nessessary. For the purpose of the walkthrough on this lab, you will not be

concerning yourself with information such as which way is forward. You will just assume that forward is positive Z, backwards is negative Z, left is negative X, and right is positive X. As far as rotation is concerned, left is negative Y and right is positive Y:

```
PlayerState Move(PlayerState previous, KeyCode newKey)
{
    float deltaX = 0, deltaY = 0, deltaZ = 0;
    float deltaRotationY = 0;

    switch (newKey)
    {
        case KeyCode.Q:
            deltaX = -0.5f;
            break;
        case KeyCode.S:
            deltaZ = -0.5f;
            break;
        case KeyCode.E:
            deltaX = 0.5f;
            break;
        case KeyCode.W:
            deltaZ = 0.5f;
            break;
        case KeyCode.A:
            deltaRotationY = -1f;
            break;
        case KeyCode.D:
            deltaRotationY = 1f;
            break;
    }
}
```

Finally, you can apply these small adjustments to a new player state, and return it out of the method. Because we are tracking adjustments (like +0.5 in the positive Z direction) you cannot just say that the Z position in your new state is deltaZ. This would cause your player to “teleport” to 4 direction, with 2 rotations and not be able to move from those.

For example, if I set my new PlayerState’s posZ to deltaZ everytime the player presses the ‘W’ key, the character would just tellport to 0.5Z in the world every time. Instead what you want to do is ADD the adjustments to the previous known position.

```
PlayerState Move(PlayerState previous, KeyCode newKey)
{
    float deltaX = 0, deltaY = 0, deltaZ = 0;
    float deltaRotationY = 0;

    switch (newKey)
    {
        case KeyCode.Q:
            deltaX = -0.5f;
            break;
        case KeyCode.S:
            deltaZ = -0.5f;
            break;
        case KeyCode.E:
            deltaX = 0.5f;
            break;
        case KeyCode.W:
            deltaZ = 0.5f;
            break;
        case KeyCode.A:
            deltaRotationY = -1f;
            break;
        case KeyCode.D:
            deltaRotationY = 1f;
            break;
    }

    return new PlayerState
    {
        posX = deltaX + previous.posX,
        posY = deltaY + previous.posY,
        posZ = deltaZ + previous.posZ,
        rotX = previous.rotX,
        rotY = deltaRotationY + previous.rotY,
        rotZ = previous.rotZ
    };
}
```

Notice that at this point in time there is no change to rotation X and rotation Z. This is because there is currently no keyboard commands that would adjust this information.

APPLYING MOVEMENT

With the movement method complete, you can now apply this movement.

You are going to want to attempt to apply movements every Update() (mainly because unity handles input very poorly outside of the update function). You are also going to want to make sure that you only apply movements if the player that the script is attached to is YOUR player (that way you don’t end up controlling every connected player!).

```
void Update()
{
    if(isLocalPlayer)
    {

    }
}
```

An important note about isLocalPlayer

- The script that is utilizing isLocalPlayer MUST extend a NetworkBehaviour
- isLocalPlayer will only return true if it is called from a script that is attached to a game object that has a NetworkIdentity component and Local Player Authority enabled
- isLocalPlayer will **always** return false if it is called in the Awake() function

Now that you know that you are working with a local player, you can begin to capture key inputs. Instead of having six if statements to check for KeyCodes, you can use an array of KeyCodes and compare against it.

```
void Update()
{
    if(isLocalPlayer)
    {
        KeyCode[] possibleKeys = { KeyCode.A, KeyCode.S, KeyCode.D, KeyCode.W, KeyCode.Q, KeyCode.E, KeyCode.Space };
        foreach (KeyCode possibleKey in possibleKeys)
        {
            if (!Input.GetKey(possibleKey)) //If the currently observed key code is not pressed
                continue;                   //Then do nothing!

        }
```

```
        }
    }
```

This allows you to condense a large amount of code (however, this method has its own drawbacks as you will see later!).

What you want to do if the user is pressing a key, is inform the SERVER that a key is being pressed and that the SERVER needs to simulate the movement and tell the client where it's new position is (remember: this is 100% server authoritative). In order to ask the server to run a function from a client, you can use the [Command] attribute. The [Command] attribute is called from a client, and is sent to the server for the server to run the function. Again, with this setup the client does not run the function, only the server does! The difference between [Command] and [Server] is that [Server] is only called and only ran from the server. [Command] is called on a client and ran on the sever.

Create a new method (CmdMoveOnServer) that will update the player's state based on a key input.

```
    [Command]
    void CmdMoveOnServer(KeyCode pressedKey)
    {
        state = Move(state, pressedKey);
    }
```

A few important notes about [Command]s:

- A command will only run if isLocalPlayer would return true
- Any method that is tied to a Command MUST start with the Cmd prefix

Finally in your update function call the command and then sync the model to the new player state.

```
    void Update()
    {
        if(isLocalPlayer)
        {
            KeyCode[] possibleKeys = { KeyCode.A, KeyCode.S, KeyCode.D, KeyCode.W, KeyCode.Q, KeyCode.E, KeyCode.Space };
            foreach (KeyCode possibleKey in possibleKeys)
            {
                if (!Input.GetKey(possibleKey)) //If the currently observed key code is not pressed
                    continue;               //Then do nothing!

                CmdMoveOnServer(possibleKey);
            }
        }

        SyncState();
    }
```

CHECKPOINT!

You can verify that your networking solution is working by:

1. Go to the Player V2 prefab, and assign it the Lab02a_PlayerControl script
2. Run the game
3. Setup Server
4. Start Host
5. If you press the W key your player should move forward
6. If you hold the D key your player should rotate right
7. If you press the W key again, your player should move to the left
    a. This is because you coded in a "static" forward direction! When the player presses the W key they will always move towards the mountains instead of moving in the direction the character is facing
8. Stop the game
9. Build the game
10. On the standalone, start a host
11. In the unity editor, connect to the server
12. Observe the client/server movement

    At this point the movement should be very choppy and not smooth. This is because you are still running a 100% authoritative server with no client-side prediction or latency compensation!

STEP THREE: PREDICTING THE PAST?

SETUP

Create a new script named Lab02b_PlayerControlPrediction

Duplicate the Player V2 prefab, and rename it to Player V3.

Remove the Lab02a_PlayerControl script from the prefab.

Add the Lab02b_PlayerControlPrediction script to the prefab.

Apply the changes to the prefab.

Open Lab02b_PlayerControlPrediction.

Copy the Lab02a_PlayerControl information into the Lab02b_PlayerControlPrediction script.

Rename the class to Lab02b_PlayerControlPrediction (at the top of the class)

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
public class Lab02b_PlayerControlPrediction : NetworkBehaviour
{
    struct PlayerState
    {
```

PSEUDOCODE

```
Create a state that will hold player position and rotation for the server (Actual)
       Sync this variable and add a Hook to it
Create a state that will hold player position and rotation for the client (Prediction)
Create a queue to hold the movements of the player

Start
       Initialize the state to default values
       Set the clients state to equal the servers state
       If this is the local player
              Initialize the queue
              update the predicted state of the client
```

```
        Sync the player character to reflect the state the player is in.

Server Command - Initialize
        Initialize the state to default values

Update
        If the current object is the local player
                Create an array for every key the player can press (and get a result)
                Loop through that array, checking to see if the player is pressing the key
                        If the player is pressing a key
                                add the key to the queue
                                update the predicted state of the client
                                command the server to move

        Sync the player character to reflect the state the player is in

SyncState
        determine if you should be rendering a predicted state (client) or the server's state (remote clients)
        Set the gameobjects position to whatever position is in the state
        Set the gameobjects rotation to whatever rotation is in the state

Command - Move on the server
        set the state to the result of the keypress

Move
        Keep track of delta x, y, z, and rotation y
                (This means the change in x, y, z, rotation y)
        Switch on the different types of keypresses that can happen
                If player pressed Q
                        Decrease delta x by some speed
                If player pressed S
                        Decrease delta z by some speed
                If player pressed E
                        Increase delta x by some speed
                If player pressed W
                        Increase delta z by some speed
                If player pressed A
                        Decrease delta rotation y by some amount
                If player pressed D
                        Increase delta rotation y by some amount

        return a new player state that reflects these changes

Declare the hook
        set the server state to a new state
        if there is a queue
                while the server has not processed all of the client movements
                        remove keys from the queue
                update the predicted state on the client

UpdatePredictedState
        set the predicted state to the servers state
        for every movement left in the queue
                predict the movement
```

CREATING THE VARIOUS STATES

When tackling the idea of client-prediction you need to remember that the server remains in control of everything. This means that if the client predicts something, and the server disagrees with the prediction, the server will always win.

In order to keep track of what the client is predicting and what the server is actually simulating you will use two states: one to represent the prediction and one to represent the server's actual version.

In this line of thinking, you only need to sync the server's actual version of the player's state. You don't want to sync the prediction, because it is just a local prediction so the client feels like everything is very responsive.

```
    [SyncVar]
    PlayerState serverState;        //This will represent the state of the player on the server

    PlayerState predictedState;     //This will represent the state of the player as predicted on
                                    //Client only!
```

Now, normally when a SyncVar'd variable is changed, the server just blasts this change out to all of the clients. In this case, what that will do is just change the various variables inside of the state (such as posX, posY, posZ, ect.).

Let's say the client starts out at posX = 0, posY = 0, posZ = 0.

They then press some buttons and end up at posX = 10, posY = 1, posZ = 0. This took the client one frame to complete. The client sends the command to the server to update their state (approx. 50ms on a perfect connect), the server updates the state, and then the server blasts the changes out to all connected player (another 50ms). This whole exchange takes 100ms. In that time, the player has moved again and is now at posX = 20, posY = -10, posZ = 5.

When the end clients get this updated version of the SyncVar and the Move() method is called to render the changes, it will appear that the player jumped from 0,0,0 to 20,-10, 5. This is because it is just a raw update of the variable! The game is not re-rendering the position of the player when the variable updates, it is just changing some numbers.

What you want to do instead is change the numbers AND update the position of the gameObject (sync the game object to the state's locations). To do this, you can use something called a hook. Adding a hook (which is a method) to a SyncVar will call the hook everytime the SyncVar is updated.

In this case, you will add a hook into a method named OnServerStateChange

```
    [SyncVar(hook = "OnServerStateChanged")]
    PlayerState serverState;        //This will represent the state of the player on the server

    PlayerState predictedState;     //This will represent the state of the player as predicted on
                                    //Client only!
```

START!

Inside of the start function you will need to do some more setting up. Since you are now going to be working with a locally predicted location of a client (on the client), *and* the actual state of the client (on the server) you need to make sure that your variables are in order.

When the client starts, you want them to start at the server's position. At the start of the game the client has not had a chance to predict his/her own movement yet, so unless they are cheating they will be where the server says they are!

```
    void Start () {
        InitState();
        predictedState = serverState;
```

Remember: InitState is a Server function that sets up the player's position on the server!

Next you will want to setup your queue, but only if you are the local player. A queue is a simple data structure similar to a List. The only difference is that it is optimized to be a first-in first-out data structure. This means when you place something into the queue it goes at the end of the list [count-1], and when you remove something from the queue it removes it from the front of the list [0]. Because it is like a list, you don't need to set a size for it. And don't forget to important System.Collections.Generic!

The Queue will represent the keys that the player has pressed on the keyboard. This will allow the server to catch up on anything it missed while it was processing other data. This means if the player presses 5 buttons between server updates, the server will know what those 5 buttons were and can simulate them (in order!) to create the illusion of seamless movement! It also means that after the server simulates those 5 buttons, it can tell the client where that result is. If the client predicted a different location, then the server can correct that prediction before the client gets too far ahead of itself.

After the queue is created, you can update the client's prediction, and finally sync the client's state.

```
Queue<KeyCode> pendingMoves;    //This will represent the moves that the player is attempting
                                //That have not been acknowledged by the server yet!
                                //Remember: Queue is a first-in first-out list
void Start () {
    InitState();
    predictedState = serverState;
    if(isLocalPlayer)
    {
        pendingMoves = new Queue<KeyCode>();
        UpdatePredictedState();
    }
    SyncState();

   }
```

ACCOUNTING FOR LATENCY

Inside of your PlayerState you need to make a small change. At this point, you are queue'ing up the player's movement keys to keep the client and the server in sync. However, and important thing to remember is how many movements the player has made! Although you can use the Queue.Count to determine how many key presses are in the queue, that will not tell you how many key presses the client has currently run. This becomes important because the server will only get updates every so often.

In order to keep track of the number of keys the user has pressed, you can simply add a movementNumber variable to your player state struct!

```
struct PlayerState
{
    public int movementNumber;
    public float posX, posY, posZ;
    public float rotX, rotY, rotZ;
}
```

Now you need to initialize this in the InitState() method. While you are in the InitState() method you will need to change state to serverState as well (you will use serverState because this method is called on the server by the server! This is the *actual* state of the player on the servers simulation).

```
void InitState()
{
    serverState = new PlayerState
    {
        movementNumber = 0,
        posX = -119f,
        posY = 165.08f,
        posZ = -924f,
        rotX = 0f,
        rotY = 0f,
        rotZ = 0f
    };
}
```

UPDATING INFORMATION

Your update function needs to account for this new architecture. Luckily you can reuse a lot of the code you already wrote in your update function!

Your goal is to make sure that you are queueing the player's input. You are already checking if the player is local, you are already checking for pressed keys, so all you need to do is add they key to the que!

After you have added the key to the que (which will eventually make its way to the server so the server can update the *actual* simulation of your character) you can tell your client to update it's prediction of your movement. This will give you the instant movement that players have come to enjoy (no input lag!).

And finally you will again send the command to the server to update its simulation, and sync the state of the object with the actual gameobject.

```
void Update()
{
    if (isLocalPlayer)
    {
        KeyCode[] possibleKeys = { KeyCode.A, KeyCode.S, KeyCode.D, KeyCode.W, KeyCode.Q, KeyCode.E, KeyCode.Space };
        foreach (KeyCode possibleKey in possibleKeys)
        {
            if (!Input.GetKey(possibleKey)) //If the currently observed key code is not pressed
                continue;                   //Then do nothing!

            pendingMoves.Enqueue(possibleKeys);
            UpdatePredictedState();
            CmdMoveOnServer(possibleKey);
        }
    }

    SyncState();
}
```

CMDMOVEONSERVER

You will need to adjust the CmdMoveOnServer slightly to set the serverState of the player, and pass in the serverState of the player (instead of using state like you did in the last script). Again, this is because this command will be run on the SERVER to update the *actual* simulation of the player instead of the predicted state of the player.

```
[Command]
void CmdMoveOnServer(KeyCode pressedKey)
{
    serverState = Move(serverState, pressedKey);
}
```

SYNCSTATE

Your SyncState method will also need a little bit of work. On the last sync, you were assuming that the state of the player was up to date. Now you need to determine if you want to sync the gameobject to the predicted state, or to the servers actual state. If you only update the gameobject to the servers actual state, that you will still have the input lag that you had before (because you still have to wait for the queued keypressed to travel to the server, for the server to simulate the *actual* state of your player, and then send that state back). So what you will want to do is if the player is local, sync the predicted state. If the player is not local (it is a dummy, another player's character projected onto your game) then sync the server information.

```
void SyncState()
{
    PlayerState stateToRender = isLocalPlayer ? predictedState : serverState;

    transform.position = new Vector3(stateToRender.posX, stateToRender.posY, stateToRender.posZ);
    transform.rotation = Quaternion.Euler(stateToRender.rotX, stateToRender.rotY, stateToRender.rotZ);
}
```

MOVE

You will need to add a very small adjustment to your Move function. Hopefully at this point you can figure out what this adjustment is going to be.

It won't be on your movement algorithm.

```
    return new PlayerState
    {
        movementNumber= 1 + previous. movementNumber,
        posX = dx + previous.posX,
        posY = dy + previous.posY,
        posZ = dz + previous.posZ,
        rotX = previous.rotX,
        rotY = dRY + previous.rotY,
        rotZ = previous.rotZ
    };
```

HOOK!

You created a hook for the serverState sync var so that everytime the variable changes, the server will render every keystroke from the last time the server updated the *actual* simulation, and the current time.

This means if the last time the server ran the *actual* simulation, the player was at 0,0,0 and it was time = 0. The next time the server runs the *actual* simulation, it is time = 1 and the player is now at 10,10,10. The hook will allow the server to render the information that happened between going from 0,0,0 and 10,10,10. That way it doesn't look like the character just teleported! (This is why you are storing the keystrokes in a queue!)

You already setup the hook on the SyncVar when you declared that the hook="OnServerStateChanged".

Now you just need to create a method with the same name, and a parameter that is the same as the SyncVar (this will represent the change).

```
void OnServerStateChanged(PlayerState newState)
{
}
```

Inside of this method, you will set the server's current state to the new state that was received.

Then, you can check and see if there are any pending keystrokes. If there are pending keystrokes, you will want to start to remove them from the que until the que is updated to the point in time that the server is.

For example, if your last predicted state on your player was movement number 200, and the server's last *actual* update was a movement number 100, you will have at *least* 100 keystrokes in your queue.

You will want to begin removing keystrokes (oldest to newest) that the server has *actually* simulated (which would be movement's 1 to 100, but not 101 to 200).

After you are done doing this, you can update the predicted state of the player!

```
void OnServerStateChanged(PlayerState newState)
{
    serverState = newState;
    if (pendingMoves != null)
    {
        while (pendingMoves.Count > (predictedState.movementNumber- serverState.movementNumber))
        {
            pendingMoves.Dequeue();
        }
        UpdatePredictedState();
    }
}
```

PREDICTION IS KEY!

Finally you can create your prediction (which is responsible for predicting new client keystrokes after a server update).

Your prediction is rather simple, start off by setting your prediction state to the *actual* state of the player. You want to make sure that everytime you predict you start off accurate!

After that, you will go through each movement in the queue and apply them to your character.

```
void UpdatePredictedState()
{
    predictedState = serverState;
    foreach(KeyCode moveKey in pendingMoves)
    {
        predictedState = Move(predictedState, moveKey);
    }
}
```

CHECKPOINT!

You can verify that this is working by:

1. Adding a debug statement in your Update() > if(isLocalPlayer) function that prints the Count of the queue

```
void Update()
{
    if (isLocalPlayer)
    {
        Debug.Log("Pending moves: " + pendingMoves.Count);

        KeyCode[] possibleKeys = { KeyCode.A, KeyCode.S, KeyCode.D, KeyCode.W, KeyCode.Q, KeyCode.E, KeyCode.Space };
```

2. Verify that Player V 2 has the Lab02b_PlayerControlPrediction script
3. Verify that the network manager's player prefab is Player V 2
4. Run the game
5. Setup Server
6. Start Server
7. Run around and watch the queue count!
    a. Note: You will notice lag when the queue count reaches zero during a movement!
8. Build the game
9. In the standalone, setup a server
10. Start the server
11. In the Unity Player connect to the server
12. Run around on both!
    a. NOTE: the unity player has BUITL-IN lag. You can see the effect most clearly by moving the client around in the unity player, but watching the client move on the standalone. If you watch the unity player and move the standalone, there will still be some jitter because it is running in the editor.

STEP FOUR: ANIMATIONS

The last step is to add animation syncing. You already have the entire architecture created for this, so only a few small tweaks are needed.

In your Lab02b_PlayerControlPrediction script, create an enum named CharacterState. This will control what animation is player (you will sync this into mechanim like last semester).

```
public enum CharacterState
{
    Idle = 0,
    WalkingForward = 1,
    WalkingBackwards = 2,
    Jumping = 4
}
```

Inside of the PlayerState struct, add a variable for a CharacterState.

```
    struct PlayerState
    {
        public int movementNumber;
        public float posX, posY, posZ;
        public float rotX, rotY, rotZ;
        public CharacterState animationState;
    }
```

Add a local variable to store the current animation state.

```
    Queue<KeyCode> pendingMoves;    //This will represent the moves that the player is attempting
                                    //That have not been acknowledged by the server yet!
                                    //Remember: Queue is a first-in first-out list

    CharacterState characterAnimationState;
```

Lastly, add a public Animator value to store the animator controller so you can switch animations on the fly.

CAPTURING…NOTHING?

There are two pieces of information you need to know to determine animations to player: if the character is moving, or if he is not.

With the current system you have set up, it is rather easy to determine if the character is moving or not. You simply need to capture if the player has pressed a button this frame! (Or continued to hold down a button).

Luckily, you already basically have this information.

In your update function, you are checking if the local player is pressing a button. You can simply add a bool flag so that if any of they KeyCodes get processed, you set it to true. Otherwise, it is false.

After determining if a key was pressed, you can check if a key was NOT pressed. If a key was not pressed, then add a new keycode to the queue (something that you aren't already using … this example uses KeyCode.Alpha0 but you can use whatever you want!).

```
    void Update()
    {
        if (isLocalPlayer)
        {
            //Debug.Log("Pending moves: " + pendingMoves.Count);

            KeyCode[] movementKeys = { KeyCode.A, KeyCode.S, KeyCode.D, KeyCode.W, KeyCode.Q, KeyCode.E, KeyCode.Space };
            bool somethingPressed = false;

            foreach (KeyCode moveKey in movementKeys)
            {
                if (!Input.GetKey(moveKey))
                    continue;

                somethingPressed = true;
                pendingMoves.Enqueue(moveKey);
                UpdatePredictedState();
                CmdMoveOnServer(moveKey);
            }

            if (!somethingPressed)
            {
                pendingMoves.Enqueue(KeyCode.Alpha0);
                UpdatePredictedState();
                CmdMoveOnServer(KeyCode.Alpha0);
            }
        }
        SyncState();
    }
```

CHANGING ANIMATIONS

Inside of your Move() function, where you are assigning variables to the new PlayerState, assign the animationState to a new function (which will return a CharacterState).

```
        return new PlayerState
```

```
        {
            movementNumber = 1 + previous.movementNumber,
            posX = deltaX + previous.posX,
            posY = deltaY + previous.posY,
            posZ = deltaZ + previous.posZ,
            rotX = previous.rotX,
            rotY = deltaRotationY + previous.rotY,
            rotZ = previous.rotZ,
            animationState = CalcAnimation(deltaX, deltaY, deltaZ, deltaRotationY)

        };
```

Inside of your SyncState method you can simply change the animation by setting the CharacterState parameter set up in mechanim.

```
    void SyncState()
    {
        PlayerState stateToRender = isLocalPlayer ? predictedState : serverState;

        transform.position = new Vector3(stateToRender.posX, stateToRender.posY, stateToRender.posZ);
        transform.rotation = Quaternion.Euler(stateToRender.rotX, stateToRender.rotY, stateToRender.rotZ);
        controller.SetInteger("CharacterState", (int)stateToRender.animationState);
    }
```

DETERMINING THE NEW ANIMATION

Create the CalcAnimation function with the appropriate parameters.

```
    CharacterState CalcAnimation(float dX, float dY, float dZ, float dRY)
    {

    }
```

In here, you will want to calculate which animation to play. For this particular movement algorithm it is relatively easy: if any of the position deltas are greater than zero, then the animation should WalkForward, if they are negative, then the animation should WalkBackwards, if they are zero, then they animation should Idle.

```
    CharacterState CalcAnimation(float dx, float dy, float dz, float dRY)
    {
        if (dx == 0 && dy == 0 && dz == 0)
            return CharacterState.Idle;

        if (dx != 0 || dz != 0)
        {
            if (dx > 0 || dz > 0)
                return CharacterState.WalkingForward;
            else
                return CharacterState.WalkingBackwards;
        }

        return CharacterState.Idle;
    }
```

CHECKPOINT!

You can verify this is working by:

1. Assigning the UnityGuy controller (from the prefab) into the controller slot on the script
2. Opening UnityGuy controller (double click on the controller then look for the animator window)
3. Clicking on WalkForward
4. Clicking the transition from Idle_Ready to WalkForward (the white line with an arrow pointing towards WalkForward)
5. Unchecking HasExitTime (in the inspector)
6. Clicking the transition from WalkForward to Idle_Ready (the white line with an arrow pointing towards Idle_Ready)
7. Unchecking HasExitTime (in the inspector)
8. Saving
9. Playing
10. Moving around and observing the animation
    a. Suggest moving the speed down from 0.5 to something like 0.1

WRAP UP

You now have the tools to create a very basic networked game with a simple HLAPI, sync animations and a custom controller, and implement basic latency compensation and client-side prediction

THINK ABOUT IT!

1. What is the difference between [Client], [Server], and [Command]?
2. What is a hook and what is it used for?
3. What is client-side prediction?
4. What is latency compensation?
5. Why is client-side prediction and latency compensation used?

ON YOUR OWN!

1. Finish the jumping movement algorithm (don’t worry about animating it!).
2. Add a running motion to the movement algorithm (animate this!).
    i. The animations are already in mechanim, you just need to create the transitions and tie them into your state machine
3. Make the speeds that the player moves at designer-adjustable
4. Extra Credit: Change the movement algorithm to that it moves the player relative to the direction the player is facing