# CS447 Project – Group 4

**Barış Karaer**
**Alper Sayan**
**Taci Ata Küçükpınar**

## Project Description

Our project is to demonstrate 3 main network programming techniques (Client-Side Prediction, Server Reconciliation, Entity Interpolation) that used in multiplayer video games to reduce the negative effects of the network delay.

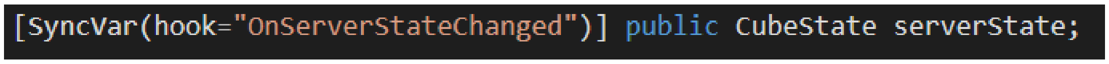
We followed Gabriel Gambetta's 5 part Fast-Paced Multiplayer tutorial. We tried to implement the real multiplayer version of the JavaScript live demo at the end of the tutorial.

We used the examples and algorithms from the Gabriel Gambetta's tutorial, Unity's Multiplayer Documentation and Valve's Source Multiplayer Networking articles. (see References.)

## Implementation

In Unity, every game object has scripts attached to it. Every game object in the scene runs its script. FixedUpdate method gets called in every 20ms.

### a.) Client-Side Prediction and Server Reconciliation



```
[SyncVar(hook="OnServerStateChanged")] public CubeState serverState;
```

With SyncVar keyword, when the server updates its serverState variable, it propagates this value to all clients. With hook keyword, we can call the OnServerStateChanged method whenever the serverState variable is updated.

```
if (Prediction)
{
    if (!isLocalPlayer)
    {
        stateHandler = (ICubeStateHandler)gameObject.AddComponent<CubePlayerObserved>();
        return;
    }

    stateHandler = (ICubeStateHandler)gameObject.AddComponent<CubePlayerPredicted>();
    gameObject.AddComponent<CubePlayerInput>();

}
else
{
    stateHandler = (ICubeStateHandler)gameObject.AddComponent<CubePlayerObserved>();
    gameObject.AddComponent<CubePlayerInput>();
}
```

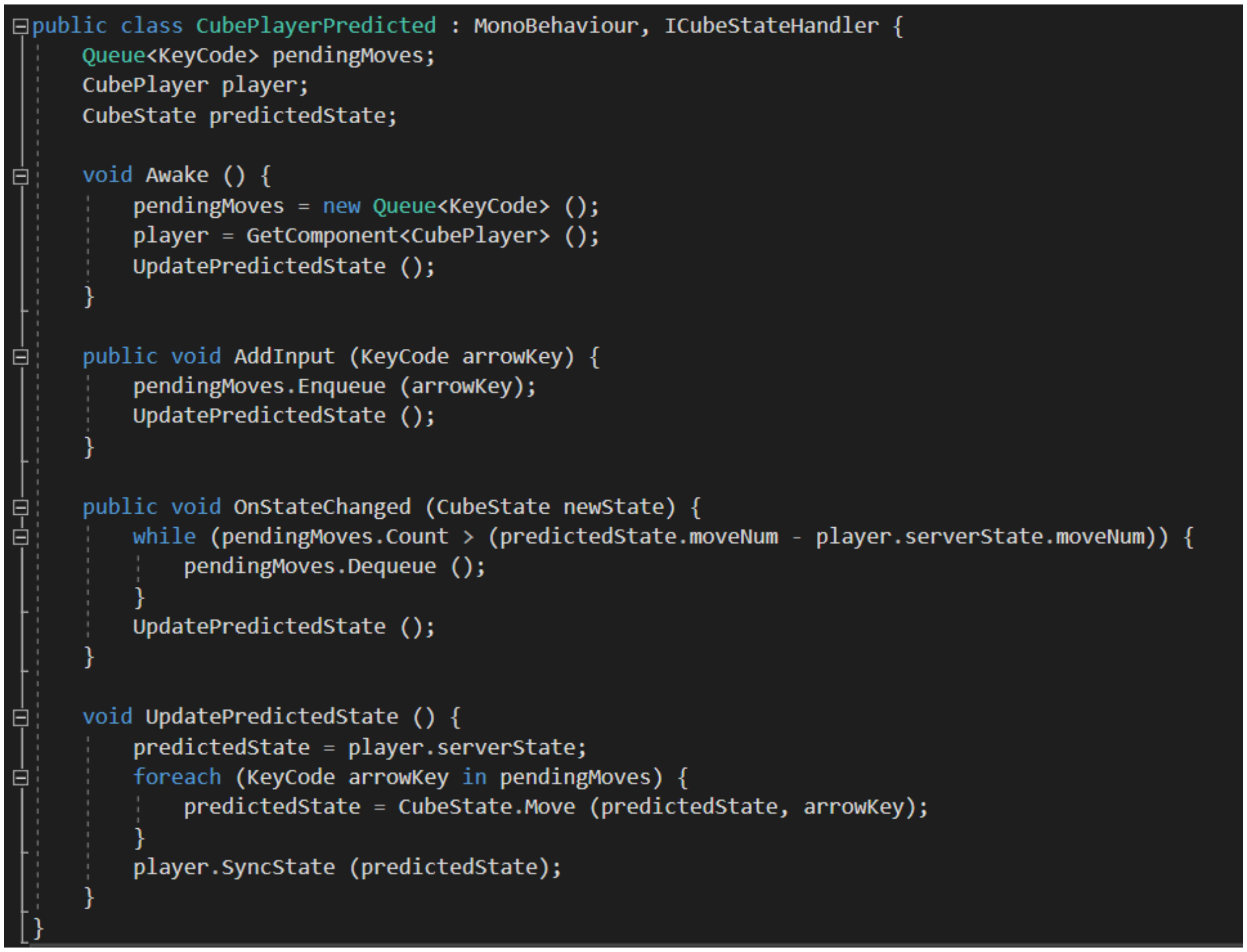
```
void OnServerStateChanged (CubeState newState) {
    serverState = newState;
    if (stateHandler == null) return;
    stateHandler.OnStateChanged (serverState);
}
```

We added a bool Prediction variable to show easily what happens if we close client-side prediction. isLocalPlayer is Unity's available bool value. It returns true if Cube instance belongs to the player on the local machine. So, if Cube belongs to us and Prediction is true, we attach the CubePlayerPredicted script to the Cube game object. Otherwise, we attach the CubePlayerObserved script.

```
public void OnStateChanged (CubeState newState) {
    player.SyncState (newState);
}
```

```
public void SyncState (CubeState stateToUse) {
    transform.position = stateToUse.x * Vector3.right + stateToUse.y * Vector3.up;
}
```

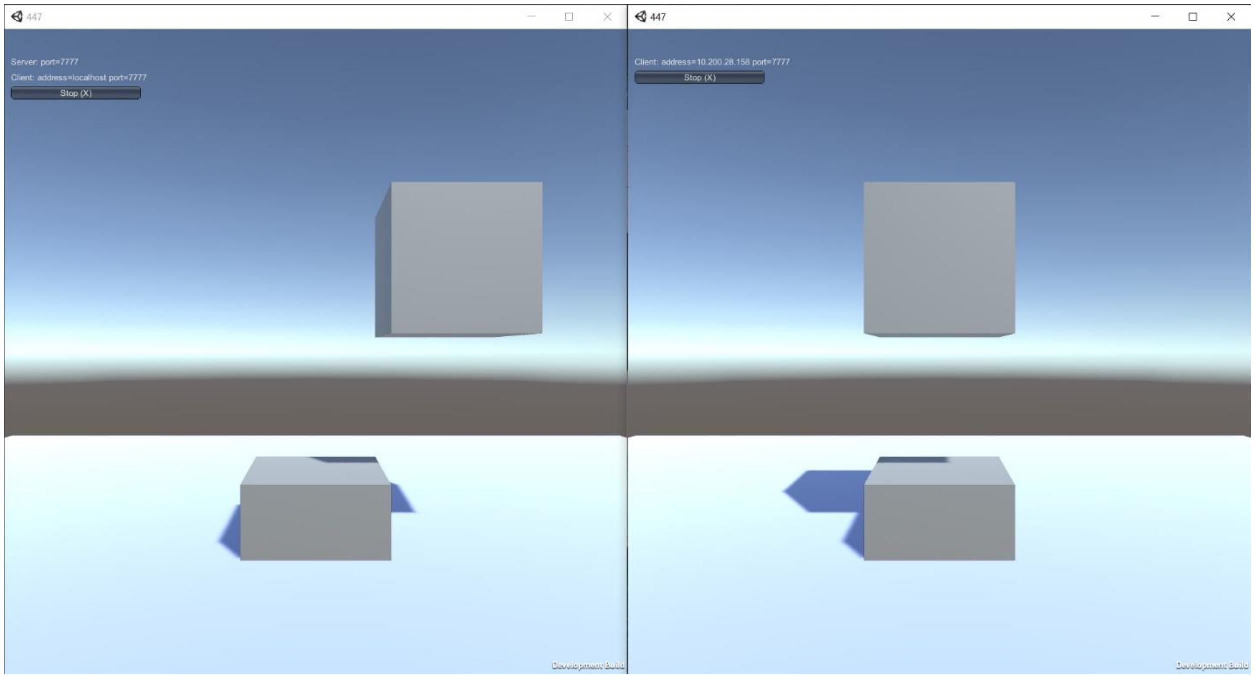
These CubePlayerPredicted and CubePlayerObserved scripts have their different OnStateChanged implementation. When CubePlayerObserved's OnStateChanged method is called, it just synchronizes object's state with serverState's position. Input will be sent to the server first. Then server will update object's position and propagate this to the clients and then we will see the synchronized position on our screen after the network delay.



```
public class CubePlayerPredicted : MonoBehaviour, ICubeStateHandler {
    Queue<KeyCode> pendingMoves;
    CubePlayer player;
    CubeState predictedState;

    void Awake () {
        pendingMoves = new Queue<KeyCode> ();
        player = GetComponent<CubePlayer> ();
        UpdatePredictedState ();
    }

    public void AddInput (KeyCode arrowKey) {
        pendingMoves.Enqueue (arrowKey);
        UpdatePredictedState ();
    }

    public void OnStateChanged (CubeState newState) {
        while (pendingMoves.Count > (predictedState.moveNum - player.serverState.moveNum)) {
            pendingMoves.Dequeue ();
        }
        UpdatePredictedState ();
    }

    void UpdatePredictedState () {
        predictedState = player.serverState;
        foreach (KeyCode arrowKey in pendingMoves) {
            predictedState = CubeState.Move (predictedState, arrowKey);
        }
        player.SyncState (predictedState);
    }
}
```

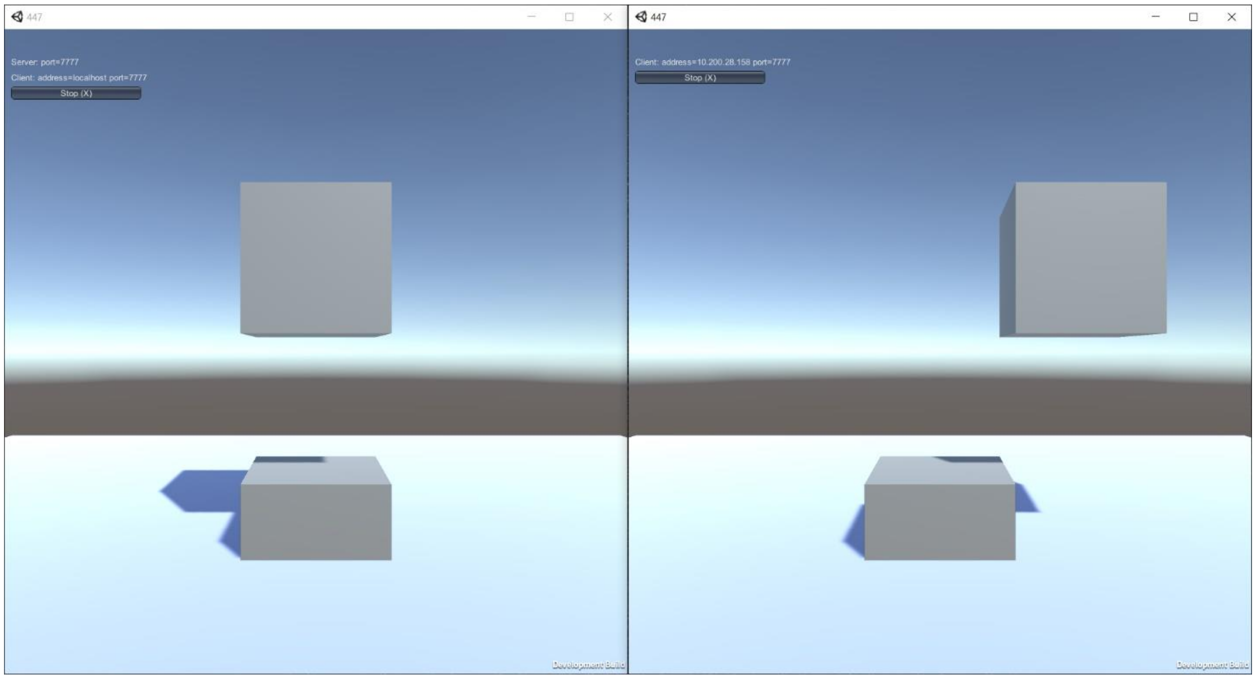
In CubePlayerPredicted script, we hold all the inputs in the queue and directly update our object's position without waiting for serverState's values from the server. Whenever the serverState's value arrives from the server which means OnStateChanged method is called, we delete this processed input from the queue.

## Demo #1 (Build1) with Client-Side Prediction and Server Reconciliation Disabled


447
Server: port=7777
Client: address=localhost port=7777
Stop (X)
447
Client: address=10.200.28.158 port=7777
Stop (X)
Development Build
Development Build

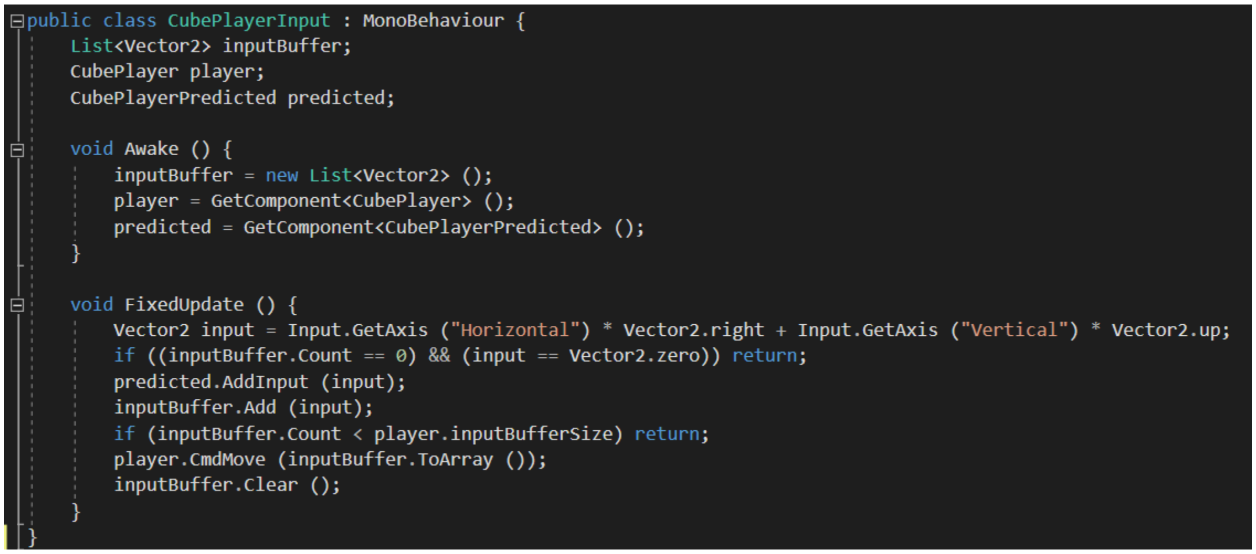
In this case, the server is on the left and the client is on the right. When you send move right input with the arrow keys, the server will move first and then the client will receive the update and it will follow. So, the client will feel the network delay while waiting for the server’s update.

**Demo #2 (Build2) with Client-Side Prediction and Server Reconciliation Enabled**


447
Server: port=7777
Client: address=localhost port=7777
Stop (X)
Development Build
447
Client: address=10.200.28.158 port=7777
Stop (X)
Development Build

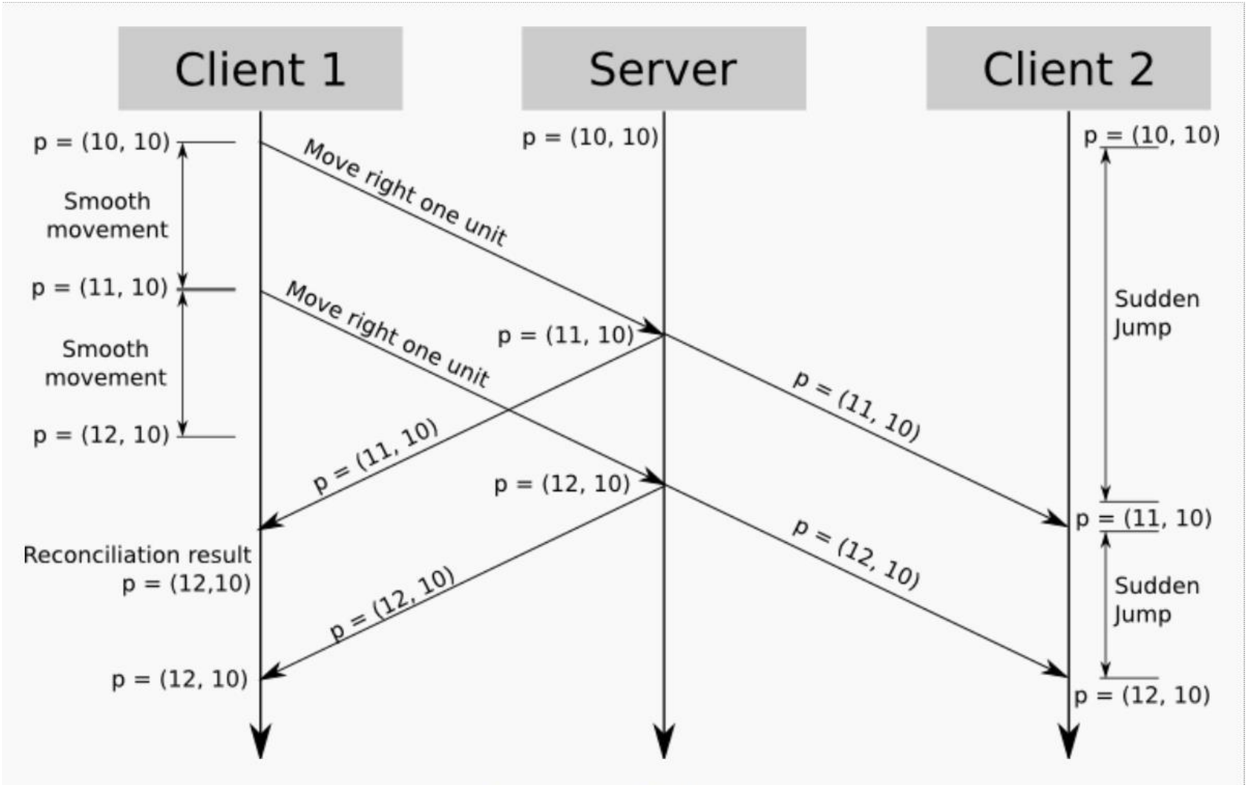
Again, the server is on the left and the client is on the right. In this case, for example, if you send move right input with the arrow keys, the client will process the input first and then it will send it to the server. So, the client will update its position without delay and the server will reconcile. As a result, client will get more responsive feedback from the inputs.

**b.) Entity Interpolation**

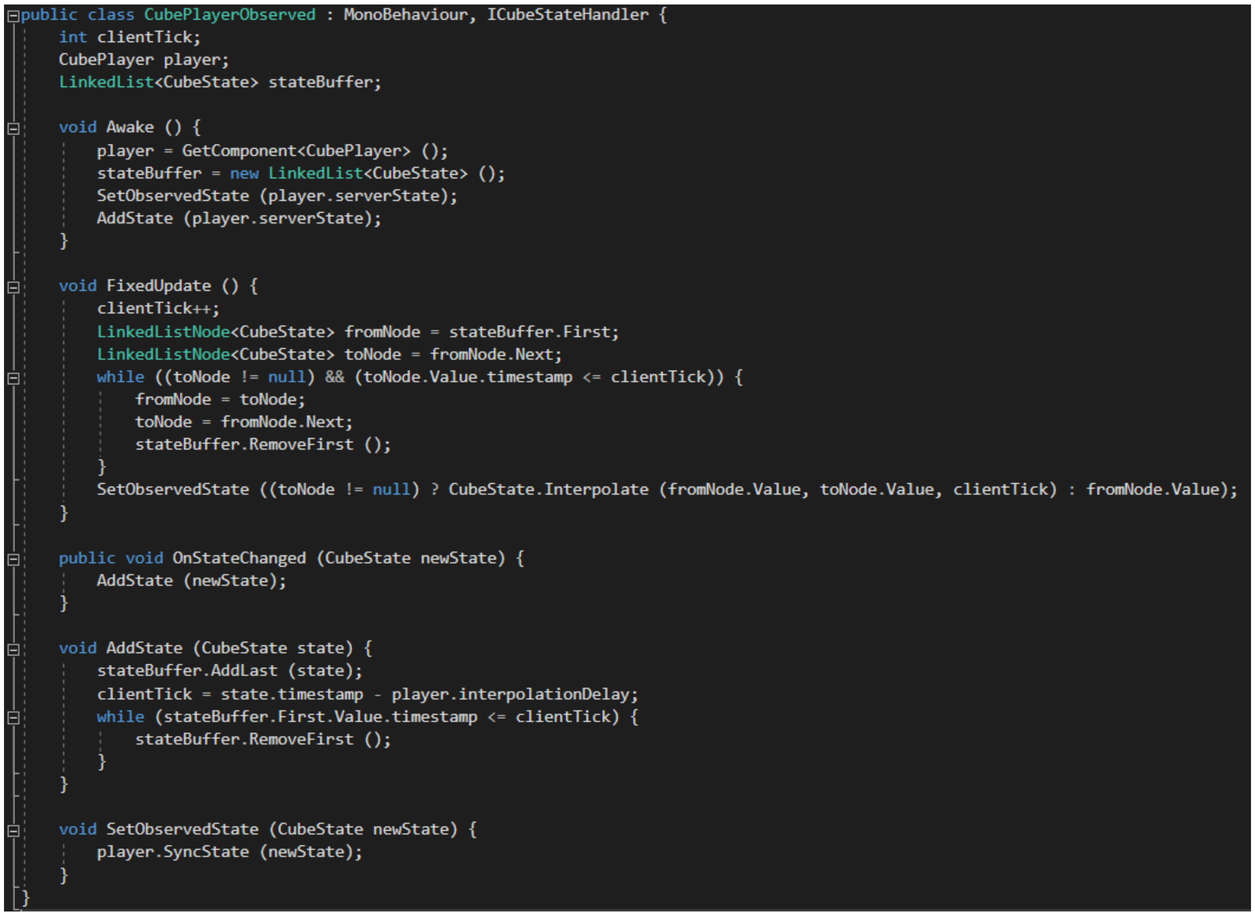


```
public class CubePlayerInput : MonoBehaviour {
    List<Vector2> inputBuffer;
    CubePlayer player;
    CubePlayerPredicted predicted;

    void Awake () {
        inputBuffer = new List<Vector2> ();
        player = GetComponent<CubePlayer> ();
        predicted = GetComponent<CubePlayerPredicted> ();
    }

    void FixedUpdate () {
        Vector2 input = Input.GetAxis ("Horizontal") * Vector2.right + Input.GetAxis ("Vertical") * Vector2.up;
        if ((inputBuffer.Count == 0) && (input == Vector2.zero)) return;
        predicted.AddInput (input);
        inputBuffer.Add (input);
        if (inputBuffer.Count < player.inputBufferSize) return;
        player.CmdMove (inputBuffer.ToArray ());
        inputBuffer.Clear ();
    }
}
```

In this build, we added velocity to have a smooth movement on key press. We also buffer the inputs in the list to not overwhelm the server with continuous commands. We hold inputs in the list and after the list gets full, we send these to the server. So, we will see a smooth movement in our object because of the prediction, but server and other clients will see sudden jumps like described in the picture below. Our aim is to see a smooth movement on the server even though we hold inputs in the buffer.

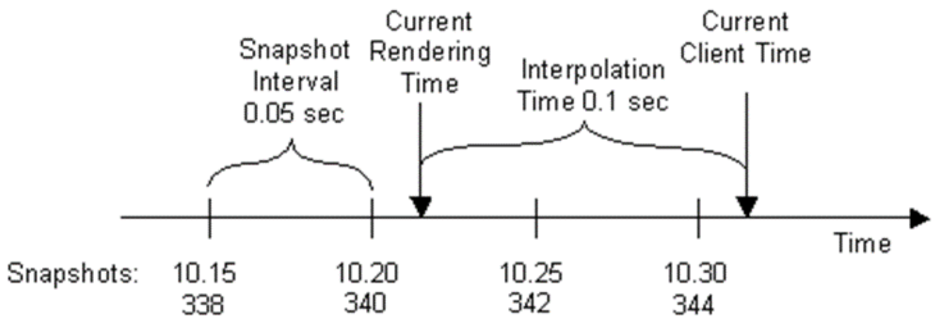


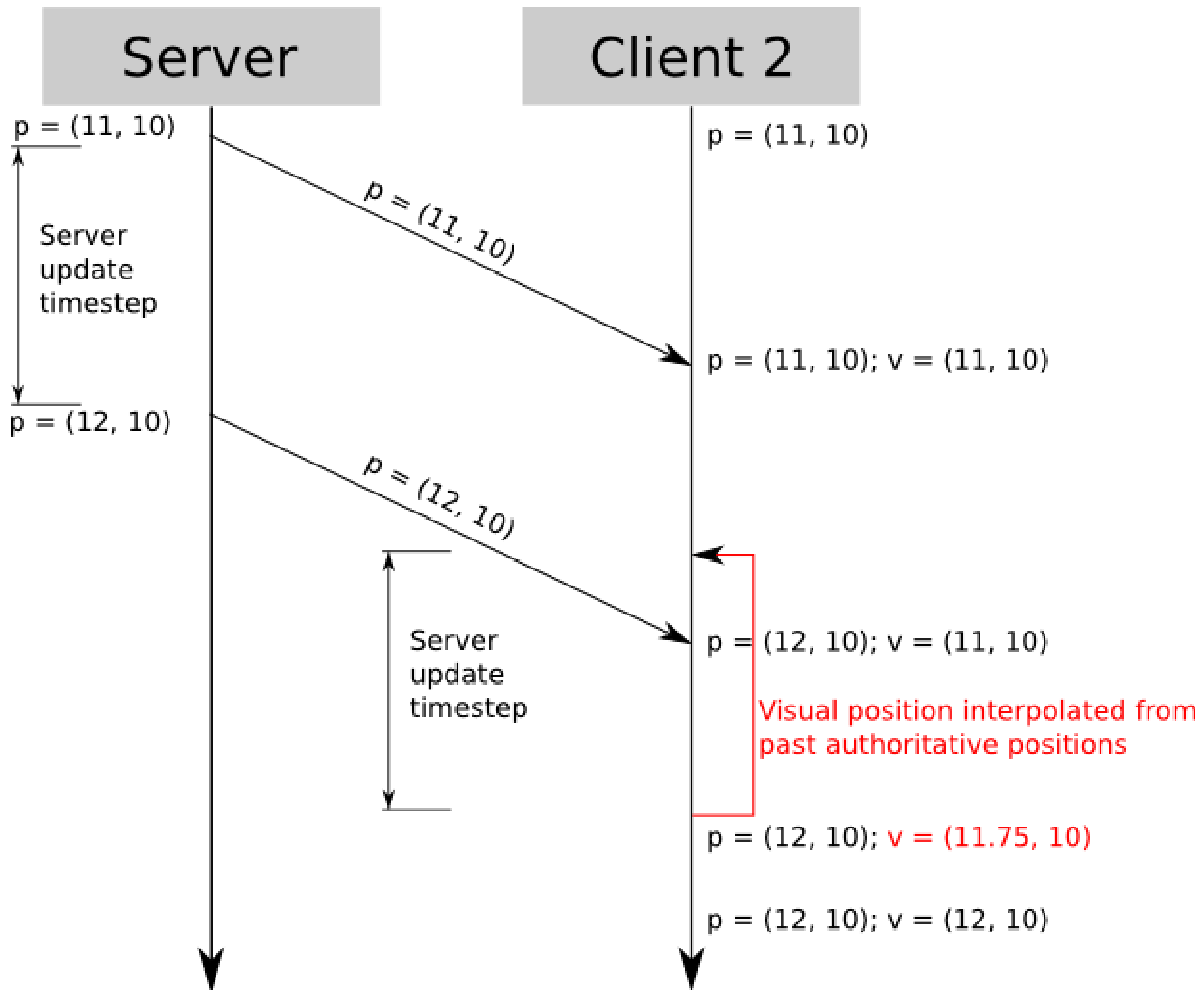
*Client 1 as seen by Client 2.*

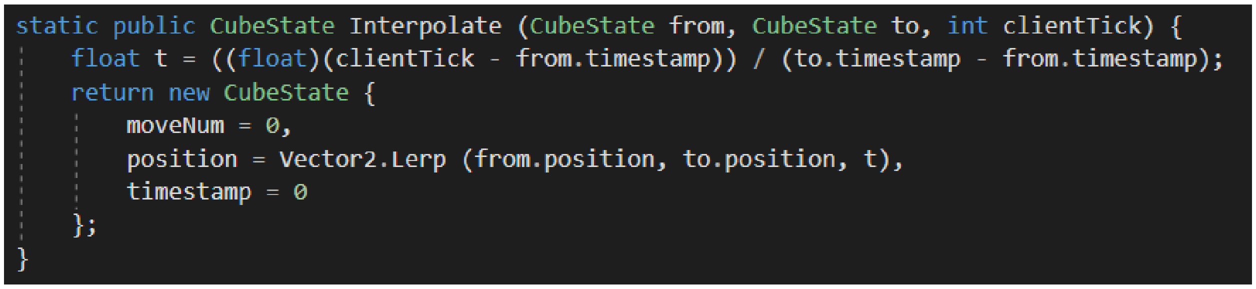


```
public class CubePlayerObserved : MonoBehaviour, ICubeStateHandler {
    int clientTick;
    CubePlayer player;
    LinkedList<CubeState> stateBuffer;

    void Awake () {
        player = GetComponent<CubePlayer> ();
        stateBuffer = new LinkedList<CubeState> ();
        SetObservedState (player.serverState);
        AddState (player.serverState);
    }

    void FixedUpdate () {
        clientTick++;
        LinkedListNode<CubeState> fromNode = stateBuffer.First;
        LinkedListNode<CubeState> toNode = fromNode.Next;
        while ((toNode != null) && (toNode.Value.timestamp <= clientTick)) {
            fromNode = toNode;
            toNode = fromNode.Next;
            stateBuffer.RemoveFirst ();
        }
        SetObservedState ((toNode != null) ? CubeState.Interpolate (fromNode.Value, toNode.Value, clientTick) : fromNode.Value);
    }

    public void OnStateChanged (CubeState newState) {
        AddState (newState);
    }

    void AddState (CubeState state) {
        stateBuffer.AddLast (state);
        clientTick = state.timestamp - player.interpolationDelay;
        while (stateBuffer.First.Value.timestamp <= clientTick) {
            stateBuffer.RemoveFirst ();
        }
    }

    void SetObservedState (CubeState newState) {
        player.SyncState (newState);
    }
}
```

We hold states in the linked list with their timestamp. We don’t update the position immediately. We interpolate it according to the past positions like shown in the second picture below. Our interpolation delay is 2 times the client tick rate similar to the example below. Because if we lose one packet, we still have 2 positions to interpolate between.




Server
Client 2
p = (11, 10)
Server update timestep
p = (12, 10)
p = (11, 10)
p = (12, 10)
p = (11, 10)
p = (11, 10); v = (11, 10)
Server update timestep
p = (12, 10); v = (11, 10)
Visual position interpolated from past authoritative positions
p = (12, 10); v = (11.75, 10)
p = (12, 10); v = (12, 10)



```
static public CubeState Interpolate (CubeState from, CubeState to, int clientTick) {
    float t = ((float)(clientTick - from.timestamp)) / (to.timestamp - from.timestamp);
    return new CubeState {
        moveNum = 0,
        position = Vector2.Lerp (from.position, to.position, t),
        timestamp = 0
    };
}
```

This is our linear interpolation algorithm. For example, if the clientTick is exactly in the middle of the two timestamps, t value will be equal to the 0.5. Then, when we give this t value and 2 positions as vector to the Lerp function, it will find us the middle point of these two vectors.

**Demo #3 (Build3) Entity Interpolation**

In this demo when we move our Cube on the client, we see a smooth movement on the server instead of a jumpy movement.

## User Manual

We have 3 builds for the Unity implementation. To run the demo:

- Open Build1 folder
- Run 447.exe
- Run it again to have 2 different windows
- Click to the ‘LAN HOST(H)’ button to run as a server
- Open the second window
- Write your local IP address next to the ‘LAN CLIENT(C)’ button
- Then click to the ‘LAN CLIENT(C)’ button
- Move the Cube with the arrow keys

Repeat the same steps with the Build2 and Build3.

## References

Gambetta, G. (2019). Fast-Paced Multiplayer. Retrieved from https://www.gabrielgambetta.com/client-server-game-architecture.html

Unity. (2019). Multiplayer and Networking. Retrieved from https://docs.unity3d.com/Manual/UNet.html

Valve. (2019). Source Multiplayer Networking. Retrieved from https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Entity_interpolation