Contents

[Changelog 2](#_Toc156825288)

[2024-01-09 – Alvin Tran 2](#_Toc156825289)

[Introduction 2](#_Toc156825290)

[Disclaimer and Resources 2](#_Toc156825291)

[Uncovered Topics: 2](#_Toc156825292)

[Prerequisites 2](#_Toc156825293)

[Additional Tools 3](#_Toc156825294)

[Getting Starting: Connecting and Starting Scene 4](#_Toc156825295)

[Some Terminology 4](#_Toc156825296)

[Necessary Packages 4](#_Toc156825297)

[Network Manager 5](#_Toc156825298)

[Network Transport 5](#_Toc156825299)

[Network Prefabs List 6](#_Toc156825300)

[Network Objects: Creating Player object 7](#_Toc156825301)

[Network Object Component 7](#_Toc156825302)

[Network Transform 8](#_Toc156825303)

[Test it out! 9](#_Toc156825304)

[Connecting to Server/Host 10](#_Toc156825305)

[Testing Multiple Screens 10](#_Toc156825306)

[Connecting 10](#_Toc156825307)

[Variables and Awake: 11](#_Toc156825308)

[Testing 13](#_Toc156825309)

[Changes to Scripting 16](#_Toc156825310)

[NetworkBehavior 16](#_Toc156825311)

[Useful Functions/Variables 16](#_Toc156825312)

[Useful Methods 16](#_Toc156825313)

[Camera 16](#_Toc156825314)

[Movement itself 16](#_Toc156825315)

[Client vs Server Authoritative network Transform 17](#_Toc156825316)

[Object Creation 17](#_Toc156825317)

[Example Bullet 17](#_Toc156825318)

[Spawning 17](#_Toc156825319)

[Remote Procedure Calls 19](#_Toc156825320)

[Pooling 19](#_Toc156825321)

[Data Management 19](#_Toc156825322)

[Network Variables 19](#_Toc156825323)

# Changelog

## 2024-01-09 – Alvin Tran

* Created

# Introduction

Hi, this is Alvin Tran, previous student of CSE 528 Game Design at UofL. I took it during Spring 2023. My team and I developed an online multiplayer game using Unity’s packages for online and Dr. Dar-Jen Chang requested that we make a document such that other people in the class can follow lead.

Also I’m assuming you have a basic idea of using Unity and navigating it, I’ll just be telling you what to add and where to look for things for most part. **This guide is more of a quickstart rather than a full tutorial**. I’ll be glossing over a few cardinal principals to consider when making games.

Theres a TLDR at the end to find things as a QuickStart and throw in.

## Disclaimer and Resources

I am no proficient writer, not am I a good game developer. You can get an equal understanding following a multitude of different tutorials available online for this kind of stuff. I like the first link below.

1. <https://www.youtube.com/watch?v=3yuBOB3VrCk>
   1. I would recommend this, it got me 99% of the way, and is basically what this document is going over.
2. Our final submission <https://github.com/ArcaneBow7258/CSE528_USPS>
3. Github for documentation:

## Uncovered Topics:

1. Syncing Animation
2. VOIP
3. Changing Scenes (We never did this, good luck)
4. Connections without Unity (always online, no local)

## Prerequisites

1. Netcode
   1. **Unity Netcode for Gameobjects (NGO)**
   2. FishNet
   3. Photon
   4. Mirror
2. Network / Relay
   1. **Unity Relay**
   2. Steam’s Facepunch
3. OPTIONAL: Unity Lobby

Here I’ll be using Unity Netcode for Gameobjects (NGO) and Unity Relay (Relay). There are other solutions that may be easier to work with, I read the other netcode alternatives are very good and more intuitive than Unity’s.

Unity Lobby is a solution to have players create/join lobbies and matchmake. I won’t cover it here in depth, but it makes getting Unity Relay codes online easier.

## Additional Tools

1. Parallel Sync (Debugging)
   1. Ability to run 2 of the same Unity instances at once

# Getting Starting: Connecting and Starting Scene

## Some Terminology

* Player ~ Client
* Game Instance = Connection
* Server -> Game running w/o player
* Host -> A player who is also a server

## Necessary Packages

Please install the packages below:

1. Unity Relay
   1. A close up of a number

      Description automatically generated
   2. You may need to connect to a Cloud Account on Unity Cloud Services (Remaking this project I was already connected)
2. Unity Netcode for Game Objects
   1. A grey background with white text

      Description automatically generated
3. <https://github.com/Unity-Technologies/com.unity.multiplayer.samples.coop.git?path=/Packages/com.unity.multiplayer.samples.coop#main>
   1. See Client Network Transform sections / google

## Network Manager

Create an empty GameObject to represent the Network Manager. **Network Manager** is a script attached to a singleton GameObject in a scene and by its name… controls networking. You can add it from the *Add Component* menu:

A screenshot of a computer program

Description automatically generated

There’s a lot of settings to look through but we’ll only go through the necessary ones:

### Network Transport

Add another component named **Unity Transport** and drag the object in the *Hierarchy* into the slot tat says Network Transport. If you’re taken the networks class, you can imagine what this is: protocol to control data flow.

A screenshot of a computer

Description automatically generated

We’ll begin with using the default *Protocol Type* as *Unity Transport* in **the *Unity Transport*** component.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Essentially what we’ve done thus far is set up the capability to create a connection between game instances.

### Network Prefabs List

An important setting to pay attention to is the network prefab list.

A screenshot of a computer

Description automatically generated

Essentially, the list holds registered prefabs to be shared amongst all connections. Objects that are not present will be able to be spawned on all instances.

1. **Anything you plan to instantiate should be added to this list**.
2. **Any object with data you wish to synchronize (HP, game progress**…)

You can add objects to the **Network Prefab List** through a file in your project:

A screenshot of a computer

Description automatically generated

For now, let’s focus on **Player Prefab.** This is the prefab that will be instantiated will be spawned when a player connects.

## Network Objects: Creating Player object

### Network Object Component

Let’s create a basic prefab with some shapes:

A cartoon face with arrows and red eyes

Description automatically generated

You should add some indication for direction / rotation of the object (nose, stick, arm, eyes, etc)

All GameObjects you intend to spawn or move on **ALL clients** require a *Network Object* Component.

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

This also will allow the prefab to be added to the **Network Manager’s** network prefab list we mentioned before.

You can look up what each of the settings mean, but important ones to look at:

1. Synchronize Transform – When you spawn the object into a scene, it will automatically transform the object to its correct position.
   1. You don’t need this on In-Scene [Network Objects](https://docs-multiplayer.unity3d.com/netcode/current/basics/scenemanagement/inscene-placed-networkobjects/), objects already in the scene or objects who you don’t care about transform (other managers)
2. Active Scene Synchronization – moves objects to new scene when necessary.
3. Don’t Destroy with owner – objects are mostly owned by the *Server*, but if owned by the client (player), and the client disconnects, you will lose the object unless this is checked.

For more info: <https://docs-multiplayer.unity3d.com/netcode/current/basics/networkobject/>

### Network Transform

Another component you should add is the **Network Transform**. This script is responsible for synchronizing the transform data to all connections, without it, when I move a object on my instance, it won’t move on yours (desync).

A screenshot of a computer program

Description automatically generated

You can choose what transform data to sync, and how accurate you want it to be. Generally, you want to uncheck as many syncing options as possible to conserve bandwidth. For example, a typical player will only be turning on the Y axis and moving on the X-Z plane with Y jumping, and there is not scaling present so:

A screenshot of a computer

Description automatically generated

Finally, you can add the GameObject into the **Network Prefab List** and **Player Prefab** in **network Manager**:

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

### Test it out!

1. Remove your Player Object from your scene (should just be maybe a platform to stand on or something)
2. Hit the Run Button:
   1. A screenshot of a video game

      Description automatically generated  
      Starting Scene, it is empty as no one is connected
3. NetworkManager object should have moved under **DontDestryOnLoad** in heirachy  
   Select it and on the script, pick **Start Host**A screenshot of a computer program

   Description automatically generated
4. A screenshot of a computer

   Description automatically generated  
   Our little guy should’ve spawned at (0,0,0)

## Connecting to Server/Host

Moving past starting a server, we need to get a client to connect as well.

### Testing Multiple Screens

You have two options:

1. Build the .exe file for your project and run that twice or run the .exe and the project debug (play button)
   1. Default: Ctrl + B or File -> Build and Run
2. Use a package like ParallelSync to copy your Unity project and debug on both.

### Connecting

This is where Unity Relay comes in handy. Relay allows you to easily make connections between Client and Server. We’ll set up an Empty GameObject and create a script named **RelayManager.cs**.

Theres a script in the GitHub with comments, but I’ll walk you through here as well:

#### Imports:

Self-Explanatory:

using UnityEngine;

using Unity.Services.Relay; // for relayService

using UnityEngine.Events; //For Unity events

using System.Threading.Tasks; //for Service

using Unity.Services.Relay.Models; // for allocation

using Unity.Networking.Transport.Relay; //RelayServerData

using Unity.Netcode.Transports.UTP; // for the transport

using Unity.Netcode; // To get NetworkManager Instance

using Unity.Services.Core; // get Initialization for Unity Cloud Services

using Unity.Services.Authentication; // for authenticating to Unity Cloud Services

### Variables and Awake:

1. Since this is a manager, we want only 1 instance of it to be accessible.
2. We create an event named *e\_relayDone*.
   1. If you haven’t used Unity Events, essentially you can bind functions to an “event” to “trigger” when you “invoke” the event. Quicker than calling many functions in a scrcipt, although things can get lose using binds. Completely optional.
   2. In *Awake()*, we delegate a function to swap our connection variable to true when we invoke the event later.
      1. playerId is useful in other scenarios but we won’t be using it in this tutorial.
3. Finally, we sign into Unity Services to be able to use Relay. You can make this your own function if you want to have logic for a single player game (i.e, async void *LogIn*()

public class RelayManager : MonoBehaviour

{

    public static RelayManager Instance; // ensures you only have 1 manager

    public bool connected = false;

    public UnityEvent e\_relayDone; //if you want triggers for when you connect

    public string joinCode;

    private String playerId;

    async void Awake(){

        if(Instance != null){

            Debug.Log("You have two relay managers");

        }

        else{

            Instance = this;

        }

        e\_relayDone.AddListener(delegate{connected = !connected;});

        try{

            // Need to connect to Unity Services to use their relay.

            await UnityServices.InitializeAsync();

            // Sign into unity services to use relay

            await AuthenticationService.Instance.SignInAnonymouslyAsync();

            playerId = AuthenticationService.Instance.PlayerId;

        }

        catch(RelayServiceException e){

            Debug.Log(e);

        }

    }

//………

#### Creating Relay:

For most part its just calling a series of functions to get the get allocation from Unity, connect that to our Network, and then starting the host/server/client:

//Relay Manager

public async Task<string> CreateRelay(int playerNumber = 4){

        try{

            Allocation alloc = await RelayService.Instance.CreateAllocationAsync(playerNumber);

// Create Relay Connection

            joinCode = await RelayService.Instance.GetJoinCodeAsync(alloc.AllocationId);

// get joincode for allocation we made

      // joinCode is the code needed to Join a session, you can either use this instance to get it

      // Ideally, you'll display this somewhere or find away to send it to client.

            Debug.Log("Relay created " + joinCode);

       //Idk how this works I followed a tutorial

            RelayServerData serverData = new RelayServerData(alloc, "dtls");

            //Connect our Transport to Relay Server

            NetworkManager.Singleton.GetComponent<UnityTransport>().SetRelayServerData(serverData);

            //Same as start host button using debugger we used in an earlier section

            NetworkManager.Singleton.StartHost();

            //Event trigger for on join (like removing a HUD)

            e\_relayDone.Invoke();

            return joinCode;

        }

        catch(RelayServiceException e){

            Debug.Log(e);

            return null;

        }

    }

#### Joining Relay:

Again, pretty much getting the necessary functions to connect.

//Relay manager

public async void JoinRelay(string joinCode){

        try{

            // Join Service

            JoinAllocation alloc = await RelayService.Instance.JoinAllocationAsync(joinCode);

            //Connect our Transport to Relay Server

            RelayServerData serverData = new RelayServerData(alloc, "dtls");

            NetworkManager.Singleton.GetComponent<UnityTransport>().SetRelayServerData(serverData);

             //Same as start client button in debugger.

            NetworkManager.Singleton.StartClient();

            e\_relayDone.Invoke();

        }catch(RelayServiceException e){

            Debug.Log(e);

        }

    }

### Testing

#### Simple GUI

For a testing GUI, I create a textbox to input a joinCode, 2 buttons (one for join, one for hosting), and finally a UIManager script on some GameObject in the Scene:

1. A black and white screen with a black background

   Description automatically generated

// UI Manager  
using TMPro; // for input field and Text

public class UIManager : MonoBehaviour

{

    public static UIManager Instance;

    public GameObject ui\_hud;

    public TMP\_InputField input\_joinCode;

    public TMP\_Text display\_joinCode;

    public void JoinButton(){ // Wrapper to join relay with input

        RelayManager.Instance.JoinRelay(input\_joinCode.text);

        display\_joinCode.text = input\_joinCode.text;

// should probalby move this out so you don't display a wrong code but i'm lazy, you can delegate it to relay\_join event and remove both in Join+Host button

    }

    public async void HostButton(){

// you don't need to do this, you can just have button reference RelayManager and  use the CreateRelay() directly.

        await RelayManager.Instance.CreateRelay();

        display\_joinCode.text = RelayManager.Instance.joinCode ;

    }

    void Awake(){ // ensures you only have 1 manager

        if(Instance != null){

            Debug.Log("You have two UI managers");

        }

        else{

            Instance = this;

        }

    }

    void Start(){

        // When we join/create a relay, we disable this UI

        RelayManager.Instance.e\_relayDone.AddListener(delegate{ ui\_hud.SetActive(false); });

    }

}

1. Once you connect the components to UIManager, you can Build and Run
2. On one side, hit host:
   1. A white and blue background

      Description automatically generated  
      Get the JoinCode from the RelayManagere, your Text Display, or maybe your Console Log
3. On other instance, put Code into TextBox and Hit Join
   1. Your second player Prefab should join in!  
      A cartoon character in a snowy area

      Description automatically generated

# Changes to Scripting

In this section, I’ll mention important features you should add to a basic single player script (you should’ve gone over this in another tutorial or in class)

## NetworkBehavior

Most scripts default to ***MonoBehavior***. To have access to unique network functionality like *object* *ownership*, *ClientID*, and *Remote Procedure Calls*, you’re going to need to change that to ***NetworkBehavior***

using UnityEngine;

using Unity.Netcode;

public class PlayerMovement : NetworkBehaviour

You’ll also gain access to some notable functions and methods to use:

### Useful Functions/Variables

1. **IsOwner()** – See if you’re owner
   1. **IsLocalPlayer()** is also a nice alternative
2. OwnerClientId
3. NetworkManager.IsServer // bool for logic only on server side

### Useful Methods

public override void OnNetworkSpawn(){

base.OnNetworkSpawn();

… }

Logic of when object is spawned on all connections. Useful for deleting/deactivating features or logic only one player should see (like disabling a HUD or life-bar)

## Camera

You can attach a camera to your player prefab object and the instantiated player will automatically select it as the main camera. Although this plays nice, you may want to consider using **OnNetworkSpawn()** to remove/deactivate the camera if you don’t own the object (**IsOwner())**.

From earlier, if you were joining through the Unity Editor, you might have seen:



## Movement itself

For the most part, player movement is the same is multiplayer with one exception: If you’re testing things out you might find that *one user is moving both players*. To prevent this, you want either:

if(!IsOwner) return;

to ignore further processing by non-owner controls (clients/players inherently own their own player prefabs)

OR

if(IsOwner){

// Your code here

}

### Client vs Server Authoritative network Transform

<https://docs-multiplayer.unity3d.com/netcode/current/components/networktransform/#owner-authoritative-mode>

Essentially, normally only server can perform the action of moving the object. Instead of NetworkTransform, please add *ClientNetworkTrasnform* instead of *NetworkTransform* after adding the github url below to your packages:

<https://github.com/Unity-Technologies/com.unity.multiplayer.samples.coop.git?path=/Packages/com.unity.multiplayer.samples.coop#main>

Alternatively, you could do movement Server-side through RPC’s covered later

# Object Creation

Now object instantiation is much more complicated. *Instantiation* and *Spawning* are two different concepts: Instantiation is on your local machine, while spawning will sync its location to everyone else and make it visible.

Not only that, but spawning can only be done by the server/host. If a client had a script that ran Test.spawn, it wouldn’t work.

## Example Bullet

May your bullet move forward however you want, but make sure to include these componnets:

1. **Network Object** on the VERY TOP part of Hierarchy of prefab
   1. Be sure to add to network Prefab List as well
2. **Network Transform**
   1. To update movement amongst all clients

In the example GitHub, we have a simple bullet that moves forward and expires.

## Spawning

To first demonstrate, You can try the scene with only the **Instantiate(GameObject)**

A computer screen shot of a black screen

Description automatically generated

Hard to see, but the bullets on right don’t show up on the left!

To actually get the object to be seen on all clients, you need the following line:

go.GetComponent<NetworkObject>().Spawn();

which spawns the gameobject on all connections

A white object with red eyes

Description automatically generated

Now you can see thing show up!

You see some of the objects on the left? Those are actually “destroyed” in script but are still there! This is because you also have to “despawn” when you destroy as well.

this.GetComponent<NetworkObject>().Despawn();

Another issue you might encounter is that on Host, you can create these objects and client can see, but the reverse is not true. This is because **Only the Server can Spawn Network Objects.**

### Spawning On Client Side

The above examples will work if you press buttons on the host, but doing on the client won’t get the object to appear on the other side. To get the server to do something for us, we need RPC’s: Remote Procedure Calls

## Remote Procedure Calls

These are separated into Client and Server RPC’s. Each are basically commands send to server/client from client/server to run a set of code only the can do.

### Proper Formatting

[ServerRpc(RequireOwnership = false)]

    public void [Name]ServerRpc(ServerRpcParams serverRpcParams = default){

…

[ClientRpc]

    public void [Name]ClientRpc(ClientRpcParams clientRpcParams = default){

You will get errors if your functions do not end with Server/ClientRpc. In addition, you need the params.

### Spawning and Despawning

In GameManger.cs which is a NetworkBehavior:

// GameManager.Cs

public NetworkPrefabsList networkPrefabsList;

……

[ServerRpc(RequireOwnership = false)]

    public void SpawnGameObjectServerRpc(int table\_index, Vector3 pos, Quaternion quat, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            GameObject go = Instantiate(networkPrefabsList.PrefabList[table\_index].Prefab, pos, quat);

            NetworkObject no = go.GetComponent<NetworkObject>();

            no.Spawn();

        }

    }

    [ServerRpc(RequireOwnership = false)]

    public void DespawnGameObjectServerRPC(NetworkObjectReference gameObject, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        gameObject.TryGet(out NetworkObject no);

        no.Despawn();

        Destroy(no.gameObject);

    }

1. cleintID is useful information if you want to know what player is asking to do an RPC
   1. We also verify if that client is actually connected
2. We have to utilize either the networkPrefabList, a ScritpableObject, a variable in the editor, or some Resource.Load() to get reference to GameObject to create
   1. We cannot send GameObject data through RPC.
   2. In this case, I am using the same NetworkPrefabList as a variable (same as NetworkManager’s) and getting prefabs that way
3. Instantiates Server Side, then spawns it on all connections.

### Calling

These functions act the same as any other. Since I have GameManager as a Static Instance, I can simply reference it to call:

// Player Shoot or Whatever control script you have

if(Input.GetKeyDown(KeyCode.R)){

            print("pew pew");

            GameManager.instance.SpawnGameObjectServerRpc(bullet, transform.position, transform.rotation);

        }

In my example, bullet is some integer value of the *NetworkPrefabList*.

## Pooling (Optional/Advanced)

This is an advanced concept that Unity has scripts and Covers in their NCO tutorial, but I’ll gloss over the topic:

* 1. Creating Objects and Sending that data over to clients its hard work, also takes lot of bandwidth
  2. Solution: Preinstantiate Objects on Client Side
  3. When “spawning” a GameObject, you get an object from here instead! Saving you some computation time at expense of RAM.

Core idea: don’t worry about it till it becomes an issue. If you need performance/bandwidth, use pooling.

# Data Management

## Network Variables

Remember when we mentioned said Instantiating objects is different than spawning, and instantiating on one end doesn’t do that on other? Same goes for variables. If I update a value on my client side, it doesn’t automatically update on other side.

Luckily its really easy to set up: All you need to do is change the type to **NetworkVariable<Type>** and access its value with **.Value**. This will automatically update on all connections, but again NetworkVariables are only editable serverside, so we need another RPC for that, but in there all we have to do and change numbers

public NetworkVariable<float> points = new NetworkVariable<float>();

public float local\_points = 500;

void Start()

    {

        points.Value = 500;

    }

    // Update is called once per frame

    void Update()

    {

        // Update Text

        display.text = String.Format("Server Points: {0} \n Local Points {1}", points.Value, local\_points);

        if(Input.GetKeyDown(KeyCode.T)){// Increase Server Side

            AddPointsServerRpc(100);

        }

        if(Input.GetKeyDown(KeyCode.Y)){ // Increase Client Side

            local\_points += 100;

        }

    }

/ Since Network variables arer server-side, we need an RPC from client.

    [ServerRpc(RequireOwnership = false)]

    public void AddPointsServerRpc(float add, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            points.Value += add;

        }

    }

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA blue background with white text

Description automatically generated

You can see our server points are synced, but local points are not.

**Use case**: Use a **ClientRPC** to change EVERYONE’S local\_points

# TLDR

To get things started:

1. Netcode For Game Objects, Unity Relay from Package Manager
2. <https://github.com/Unity-Technologies/com.unity.multiplayer.samples.coop.git?path=/Packages/com.unity.multiplayer.samples.coop#main> from package manager for CLientNetworkTransform
3. NetworkManager Component on a GameObject
   1. Add Unity Transport, Drag this into *NetworkTransport* in the NetworkManager Componnet.
4. RelayManager Script onto a GameObject (CTRL+A CTRL+C CTRL+V)

// In a Word Doc, RelayManager

using UnityEngine;

using Unity.Services.Relay; // for relayService

using UnityEngine.Events; //For Unity events

using System.Threading.Tasks; //for Service

using Unity.Services.Relay.Models; // for allocation

using Unity.Networking.Transport.Relay; //RelayServerData

using Unity.Netcode.Transports.UTP; // for the transport

using Unity.Netcode; // To get NetworkManager Instance

using Unity.Services.Core; // get Initialization for Unity Cloud Services

using Unity.Services.Authentication;

using System; // for authenticating to Unity Cloud Services

public class RelayManager : MonoBehaviour

{

    public static RelayManager Instance; // ensures you only have 1 manager

    public bool connected = false;

    public UnityEvent e\_relayDone; //if you want triggers for when you connect

    public string joinCode;

    private String playerId;

    async void Awake(){

        if(Instance != null){

            Debug.Log("You have two relay managers");

        }

        else{

            Instance = this;

        }

        e\_relayDone.AddListener(delegate{connected = !connected;});

        try{

            // Need to connect to Unity Services to use their relay.

            await UnityServices.InitializeAsync();

            // Sign into unity services to use relay

            await AuthenticationService.Instance.SignInAnonymouslyAsync();

            playerId = AuthenticationService.Instance.PlayerId;

        }

        catch(RelayServiceException e){

            Debug.Log(e);

        }

    }

    public async Task<string> CreateRelay(int playerNumber = 4){

        try{

            Allocation alloc = await RelayService.Instance.CreateAllocationAsync(playerNumber); // Create Relay Connection

            joinCode = await RelayService.Instance.GetJoinCodeAsync(alloc.AllocationId); // get joincode for allocation we made

            // joinCode is the code needed to Join a session, you can either use this instance to get it

            // Ideally, you'll display this somewhere or find away to send it to client.

            Debug.Log("Relay created " + joinCode);

            //Idk how this works I followed a tutorial

            RelayServerData serverData = new RelayServerData(alloc, "dtls");

            //Connect our Transport to Relay Server

            NetworkManager.Singleton.GetComponent<UnityTransport>().SetRelayServerData(serverData);

            //Same as start host button using debugger.

            NetworkManager.Singleton.StartHost();

            //Event trigger for on join (like removing a HUD)

            e\_relayDone.Invoke();

            return joinCode;

        }

        catch(RelayServiceException e){

            Debug.Log(e);

            return null;

        }

    }

    public async void JoinRelay(string joinCode){

        try{

            // Join Service

            JoinAllocation alloc = await RelayService.Instance.JoinAllocationAsync(joinCode);

            //Connect our Transport to Relay Server

            RelayServerData serverData = new RelayServerData(alloc, "dtls");

            NetworkManager.Singleton.GetComponent<UnityTransport>().SetRelayServerData(serverData);

             //Same as start client button in debugger.

            NetworkManager.Singleton.StartClient();

            e\_relayDone.Invoke();

        }catch(RelayServiceException e){

            Debug.Log(e);

        }

    }

}

1. GUI for Connection
   1. Two Buttons (or buttons on your keyboard), Text Input for Relay Code, Text to change to Display Relay Code
2. Any Object to Instantiate or Move by Server/User
   1. NetworkObject Component
   2. NetworkTransform Component if moving.
      1. ClientNetworkTransform if the client will be moving it.
   3. Counter Example: A moving wall on all game instances doesn’t need network transform/object, as it will move same on all side due to code being same.
   4. **Make Sure to add onto NetworkPrefabList on NetworkManager**
3. Player Movement:
   1. Remember to add NetworkObject + NetworkTransform Components
   2. Change script from inherit MonoBehavior -> NetworkBehavior
   3. In any part of script you only want one player to control, add
   4. If(!isOwner) return
4. Instantiating Objects and Network Variables
   1. RPC’s need specific format, see section from Table of Contents
   2. For a Example GameManager copy and paste all of this:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using Unity.Netcode;

using TMPro;

using System;

using Unity.VisualScripting;

public class GameManager : NetworkBehaviour

{

    // Network Variables are values that are updated on each side

    public static GameManager instance;

    public NetworkVariable<float> points = new NetworkVariable<float>();

    public float local\_points = 500;

    public TMP\_Text display;

    /\* Some list of network prefabs to reference

    / You can also explicitly define it such as

    / public GameObject gameObject

    / public NetworkObject networkObject

    / public List<GameObject> gameObject

    /

    / We use a list since we're using a manager, but if you had for example a turret you are handling on Server-side, you can just save the reference to a single bullet there.

    \*/

    public NetworkPrefabsList networkPrefabsList;

    void Awake(){

        if(instance != null){

            Debug.Log("You have two Game managers");

        }

        else{

            instance = this;

        }

    }

    // Start is called before the first frame update

    void Start()

    {

        points.Value = 500;

    }

    // Update is called once per frame

    void Update()

    {

        // Update Text

        display.text = String.Format("Server Points: {0} \n Local Points {1}", points.Value, local\_points);

        if(Input.GetKeyDown(KeyCode.T)){// Increase Server Side

            AddPointsServerRpc(100);

        }

        if(Input.GetKeyDown(KeyCode.Y)){ // Increase Client Side

            local\_points += 100;

        }

    }

    // Since Network variables arer server-side, we need an RPC from client.

    [ServerRpc(RequireOwnership = false)]

    public void AddPointsServerRpc(float add, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            points.Value += add;

        }

    }

    //Spawning and Despawning are only allowed Server-side as well

    [ServerRpc(RequireOwnership = false)]

    public void SpawnGameObjectServerRpc(int table\_index, Vector3 pos, Quaternion quat, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            GameObject go = Instantiate(networkPrefabsList.PrefabList[table\_index].Prefab, pos, quat);

            NetworkObject no = go.GetComponent<NetworkObject>();

            no.Spawn();

        }

    }

    [ServerRpc(RequireOwnership = false)]

    public void DespawnGameObjectServerRPC(NetworkObjectReference gameObject, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        gameObject.TryGet(out NetworkObject no);

        no.Despawn();

        Destroy(no.gameObject);

    }

}

* 1. Otherwise: important ones:
     1. To set up network variable and RPC to call to change:

public NetworkVariable<float> points = new NetworkVariable<float>();

…

[ServerRpc(RequireOwnership = false)]

    public void AddPointsServerRpc(float add, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            points.Value += add;

        }

    }

/\* Some list of network prefabs to reference

    / You can also explicitly define it such as

    / public GameObject gameObject

    / public NetworkObject networkObject

    / public List<GameObject> gameObject

    /

    / We use a list since we're using a manager, but if you had for example a turret you are handling on Server-side, you can just save the reference to a single bullet there.

    \*/

    public NetworkPrefabsList networkPrefabsList;

….

//Spawning and Despawning are only allowed Server-side as well

    [ServerRpc(RequireOwnership = false)]

    public void SpawnGameObjectServerRpc(int table\_index, Vector3 pos, Quaternion quat, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        if (NetworkManager.ConnectedClients.ContainsKey(clientId))

        {

            GameObject go = Instantiate(networkPrefabsList.PrefabList[table\_index].Prefab, pos, quat);

            NetworkObject no = go.GetComponent<NetworkObject>();

            no.Spawn();

        }

    }

    [ServerRpc(RequireOwnership = false)]

    public void DespawnGameObjectServerRPC(NetworkObjectReference gameObject, ServerRpcParams serverRpcParams = default){

        var clientId = serverRpcParams.Receive.SenderClientId;

        gameObject.TryGet(out NetworkObject no);

        no.Despawn();

        Destroy(no.gameObject);

    }

* + 1. For spawning GameObjects on a singleton: call [YourSingleton].Instance.SpawnGameObjectServerRPC
    2. You can change **networkPrefabList** to a specific GameObject in some cases (like SpawnZombieServerRPC if you have a spawner script your server is handling and use **if(!isServer) return** for client to not work on it
    3. You can also use something like