Netick Docs

This PDF is included solely to comply with Unity Asset Store submission requirements. For the most up-to-date and properly formatted documentation, please refer to the online version.

View Online Documentation

Table of Contents

0 - Overview

Introduction

In this guide, we will walk you through the process of integrating Netick into your Unity project.

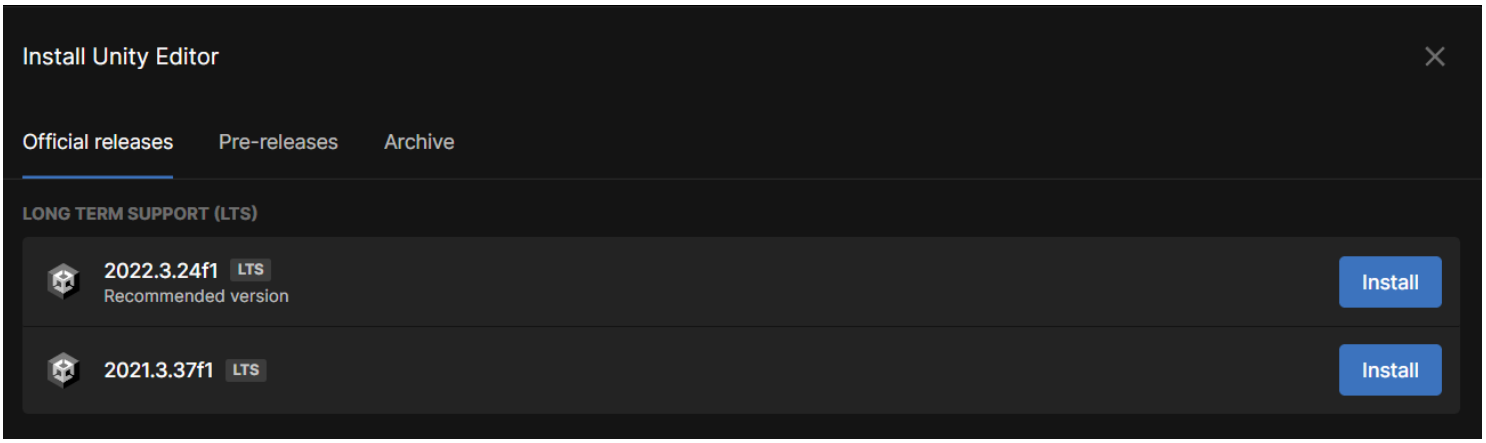
This tutorial will be very simple and only teach you the very basics. We will make a simple scene with a simple character movement controller.

> **NOTE**
>
> It's recommended that you at least have an intermediate level of C# and Unity before continuing with this tutorial.

1 - Project Setup

Step 1 - Unity Version

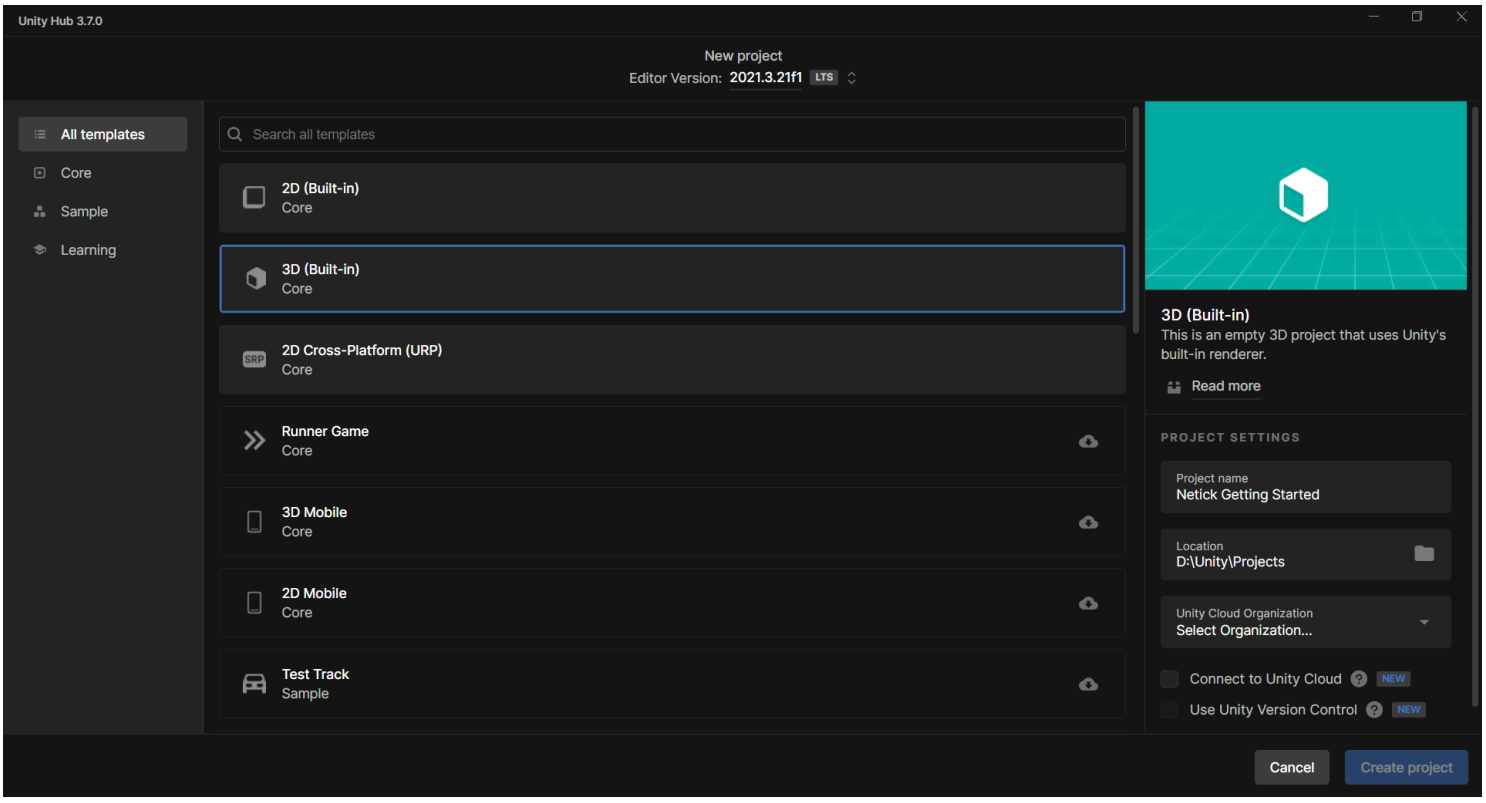
Make sure the version of Unity that you are using is **2021** or later.



Step 2 - Create a Blank Project

In this tutorial, we will choose 3D (Built-in).

Note: Render pipeline doesn't affect Netick whatsoever (unless you were using Netick samples which use built-in render pipeline).



Step 3 - Importing Netick

Go to `Window > Package Manager > "+" Icon > Add package from git URL` and fill it with https://github.com/NetickNetworking/NetickForUnity.git

Great, the project is now set, let's do some coding!

2 - Setting Up the Game

Starting the Game

There are a few methods we can use to start the game:

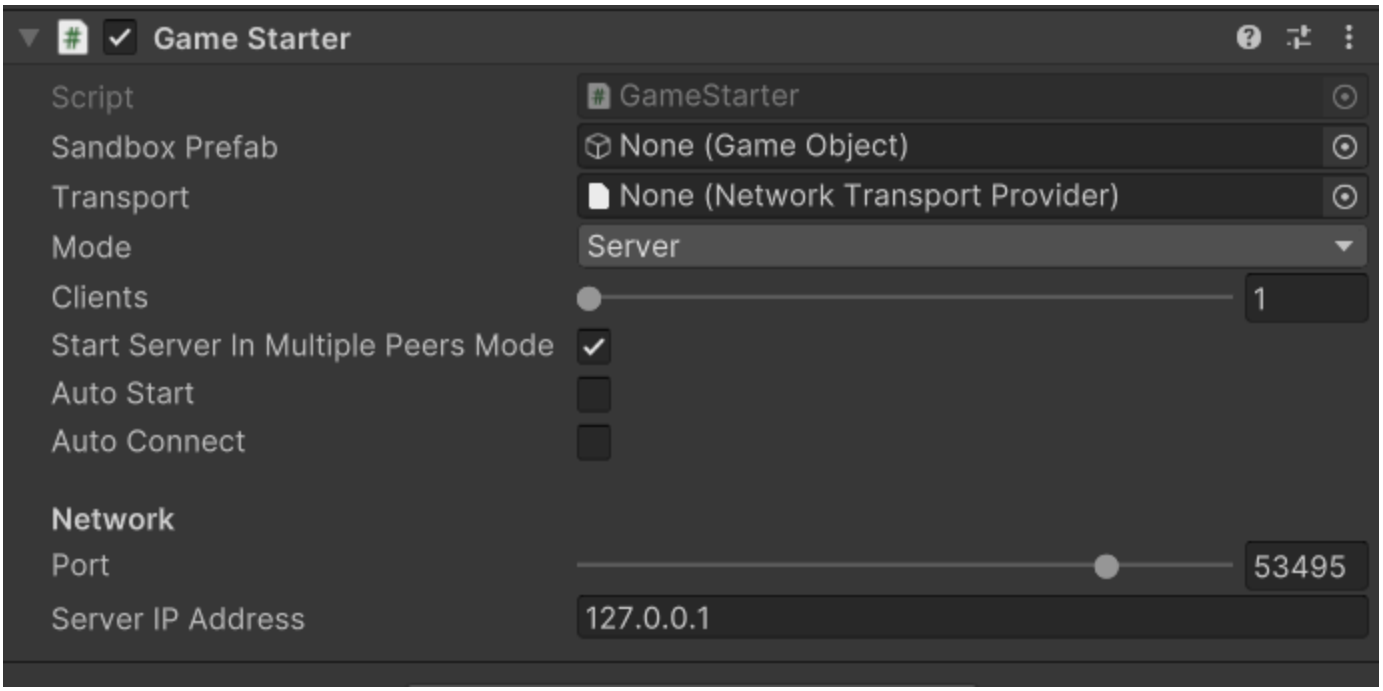
```
// manually initialize Netick (also done automatically when `Network.StartAsHost`/`Network.StartAsClient`/`Network.StartAsServer` is called)
Netick.Unity.Network.Init();

// start the game as a server with a player
Netick.Unity.Network.StartAsHost(...);

// start the game as a client
var sandbox = Netick.Unity.Network.StartAsClient(...);

// connecting the client
sandbox.Connect(...);

// or just starting a server without a player (dedicated-server way)
Netick.Unity.Network.StartAsServer(...);

// or if we wanted to shut down Netick
Netick.Unity.Network.Shutdown();
```

Learn More About Managing Netick

For quick testing, we can use the built-in `GameStarter` component, which will start the game for us.

1. Create a new empty GameObject.
2. Add the `GameStarter` Component.

After adding the component, there are several fields we need to take care of.



Sandbox Prefab

The first field asks for a `Sandbox Prefab`

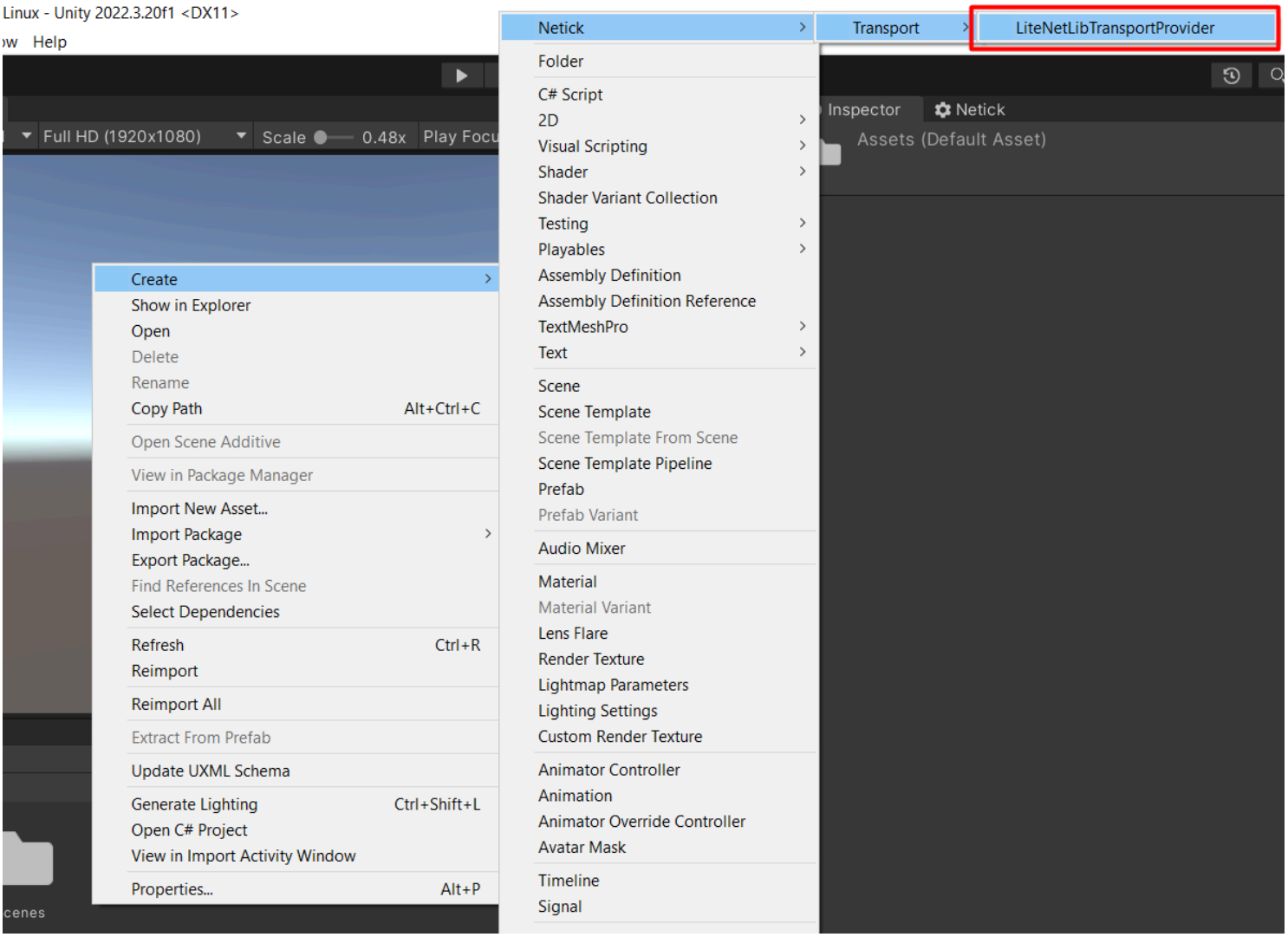
1. Create a new empty GameObject.
2. Rename it to `MySandboxPrefab` (or any name you like).
3. Save it as prefab in the Assets folder, and assign it in `Sandbox Prefab` of `GameStarter` .

> **NOTE**
>
> Netick will automatically add a few other scripts too on this prefab when it's created which will happen when you start Netick, the most important one of those scripts is `NetworkSandox`. The Sandbox Prefab is a persistent GameObject, it will only be destroyed when you shut down Netick.

Any scripts you add to the Sandbox Prefab will stay around until you shut down Netick. Methods on `NetworkSandbox` (the script Netick adds to the Sandbox Prefab) lets you do various things like connecting, spawning objects and destroying them.

Transport

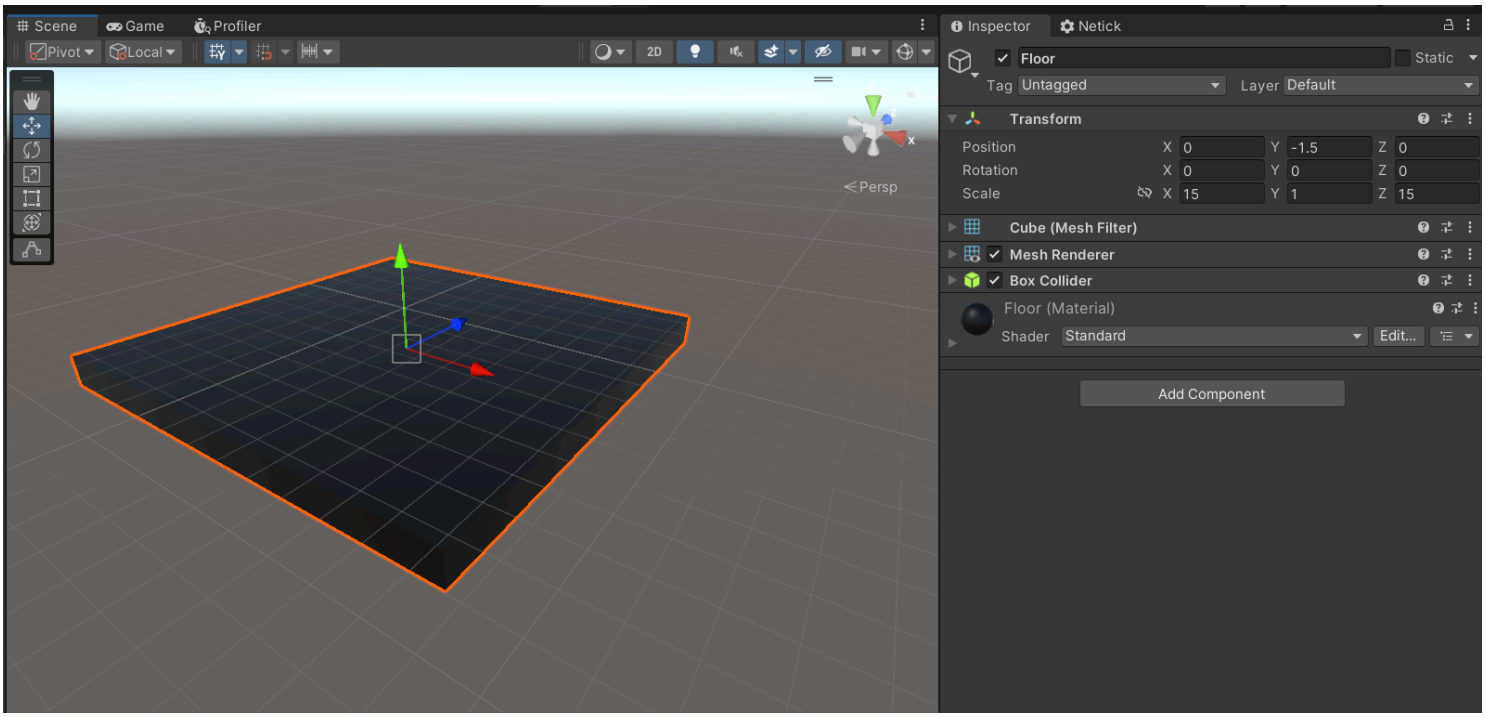
Now we need to assign a transport, Netick uses the LiteNetLib by default. To use this transport, we can right click on an empty place in the Assets folder and go to `Create > Netick > Transport > LiteNetLibTransportProvider`. Then assign the `Transport` field of Game Starter.



Setting Up the Scene

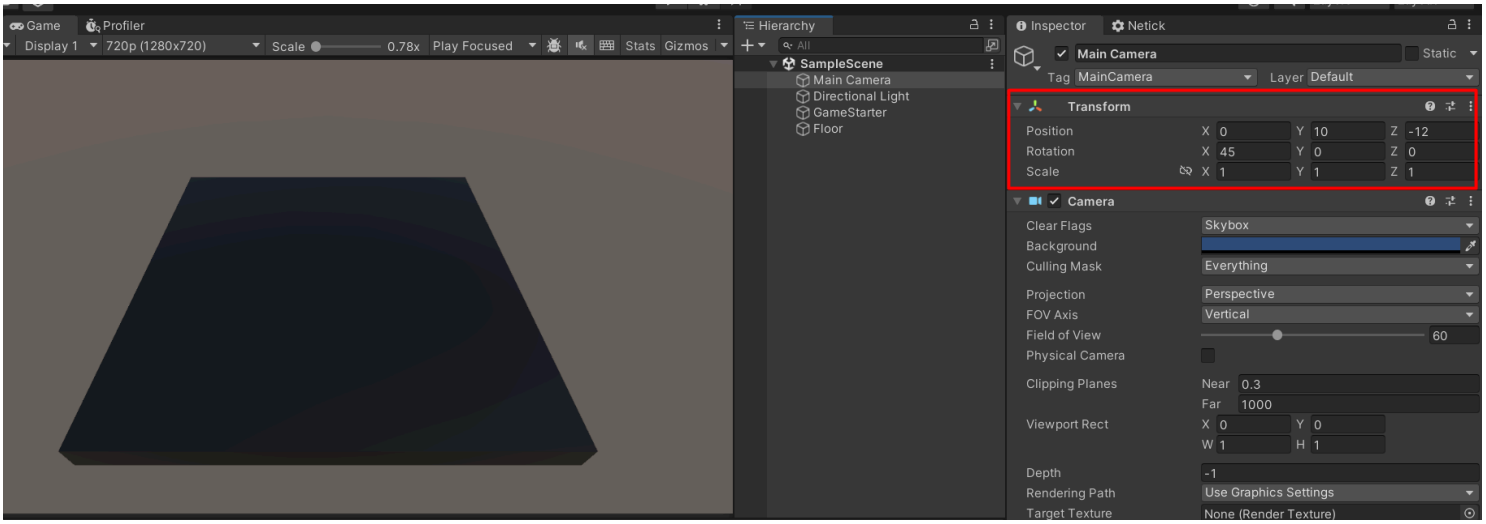
Floor

- Let's create a 3D Cube named "Floor" with a scale of `(15, 1, 15)` and a position of `(0, -1.5, 0)`.
- Create and assign a black material into it for visibility



Camera

Modify the camera's position to `(0, 10, -12)` and adjust its rotation to `(45, 0, 0)` .



Gameplay Manager

Let's create our manager script to handle gameplay aspects such as spawning the player character when a certain player joins.

Create a C# script named `GameplayManager` , then add it to the GameStarter GameObject.

This script will inherit from `NetworkEventsListener` . By doing this, `GameplayManager` now has the ability to listen to important network events such as when a player connects, disconnects, etc.

[Learn More About Listening to Network Events]

```
using Netick;
using Netick.Unity;

// Change parent class from MonoBehaviour to NetworkEventsListener
public class GameplayManager : NetworkEventsListener
{
```

```
}
```

Player Character

Let's create our player character:

1. Right click on the hierarchy and select `3D Object > Capsule` .
2. Add `NetworkObject` component.
3. Rename it to `PlayerCharacter` .
4. Remove the Capsule Collider from the Capsule we just created (we won't use physics in this tutorial)

Adding `NetworkObject` to a GameObject will give it an identity across the network, so that it's synced.

Spawning our Player

1. Add a field to hold the player character prefab in our gameplay manager script and give it type of `NetworkObject` .
2. Then, let's also spawn the character when a player connects to the server.

Input Source

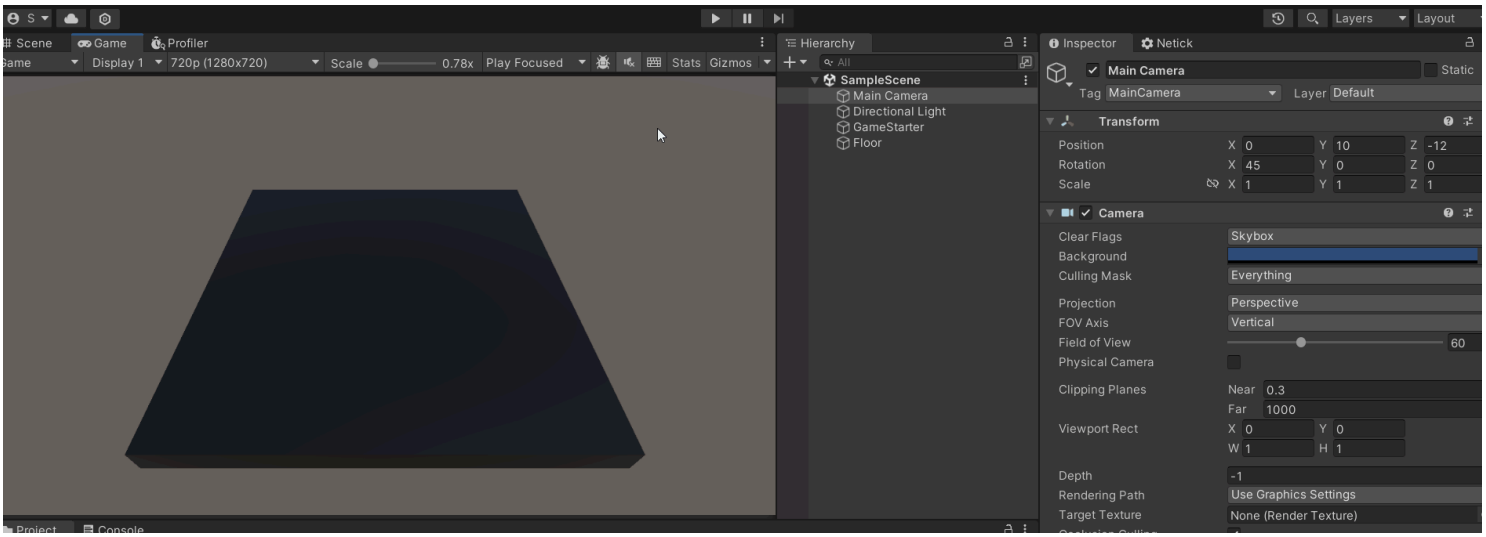
On the `NetworkInstantiate` , you can pass an Input Source. Input Source represents which peer has the authority to send inputs to this object, in this case that player is the joining player.

3. Don't forget to assign the player prefab on the `GameplayManger` script!

```
using Netick;

public class GameplayManager : NetworkEventsListener
{
    public NetworkObject PlayerPrefab;

    public override void OnPlayerJoined(NetworkSandbox sandbox, NetworkPlayerId player)
    {
        if (sandbox.IsClient)
            return;
        // Random Spawn Position (to not make them overlap)
        Vector3 spawnPosition = new Vector3();
        spawnPosition.x = Random.Range(-5f, 5f);
        spawnPosition.z = Random.Range(-5f, 5f);

        sandbox.NetworkInstantiate(PlayerPrefab.gameObject, spawnPosition, Quaternion.identity, player);
    }
}
```

Testing

Let's go ahead and enter play mode. You can see our player spawning by clicking on "Start Host".


Scene
Game
Profiler
Display 1
720p (1280x720)
Scale
0.78x
Play Focused
Stats
Gizmos
Hierarchy
SampleScene
Main Camera
Directional Light
GameStarter
Floor
Inspector
Netick
Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
Rotation
Scale
Camera
Clear Flags
Skybox
Background
Culling Mask
Everything
Projection
Perspective
FOV Axis
Vertical
Field of View
60
Physical Camera
Clipping Planes
Near
0.3
Far
1000
Viewport Rect
Depth
-1
Rendering Path
Use Graphics Settings
Target Texture
None (Render Texture)
Project
Console

3 - Player Character Movement

Since Netick is a server-authoritative networking solution, we can't directly move the player character object (in a client-auth fashion) on the client for security reasons. Instead, we use inputs, which will be used to move our player. To move the player based on our input, here's how it works: we send an input to the server, the server fetches our input, and then uses it to move our character. Netick uses something called Client-Side Prediction to make this process responsive.

Learn More About Client-Side Prediction

Input Struct

Consider the type of player inputs required for our gameplay. In this tutorial, we only use a vector for movement direction.

1. Create a C# script and call it `PlayerCharacterInput` .
2. Change the type into `struct` from `class` .
3. Make sure to implement `INetworkInput` .

This input will be sent to the server, and can be processed later on.

```
using Netick;

public struct PlayerCharacterInput : INetworkInput
{
    public Vector2 Movement;
}
```

Setting and Sending Input

There are a few places to set your input. The preferred way is on `OnInput` on `NetworkEventsListener` .

1. Modify the `GameplayManager` script.
2. Override the `OnInput` method.
3. Use `sandbox.SetInput` to set your input.

```
using Netick;
using Netick.Unity;
using UnityEngine;

public class GameplayManager : NetworkEventsListener
{
    // ...
    public override void OnInput(NetworkSandbox sandbox)
    {
        PlayerCharacterInput input = sandbox.GetInput<PlayerCharacterInput>();
        input.Movement             = new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical"));
        sandbox.SetInput(input);
    }
}
```

Learn More About Inputs

Fetch Input

We use a method called `FetchInput` to try to fetch an input for the current tick. If we are able to fetch an input, we use it to drive the gameplay logic, such as the movement of our character.

`FetchInput` must only be called inside `NetworkFixedUpdate` . To be able to use this method, let's create a new C# script named `PlayerCharacterMovement` , to handle our movement logic. Let's also change its parent class from `MonoBehaviour` to `NetworkBehaviour` .

We also need a `moveSpeed` variable, declare it using `float` type and set the default value to `5` .

Learn More About Network Behaviour

```
public class PlayerCharacterMovement : NetworkBehaviour
{
    public float moveSpeed = 5;

    public override void NetworkFixedUpdate()
    {
        if (FetchInput(out PlayerCharacterInput input))
        {
            Vector3 movement = new Vector3(input.Movement.x, 0, input.Movement.y);
            transform.position += movement * Sandbox.FixedDeltaTime * moveSpeed;
        }
    }
}
```

In a single-player game, we use `Time.deltaTime` to move our player to make it frame independent. With Netick, instead of using `Time.deltaTime`, we use `Sandbox.FixedDeltaTime`, which represents the time between two network ticks.

> **NOTE**
>
> Do not confuse `Sandbox.FixedDeltaTime` with `Sandbox.DeltaTime` (equal to Unity's `Time.deltaTime`).

Learn More About Writing Client-Side Prediction Code

Network Transform

Adding `NetworkTransform` allows us to sync the position, rotation of our character.

- Add `NetworkTransform` component to our player character prefab.

Learn More About NetworkTransform

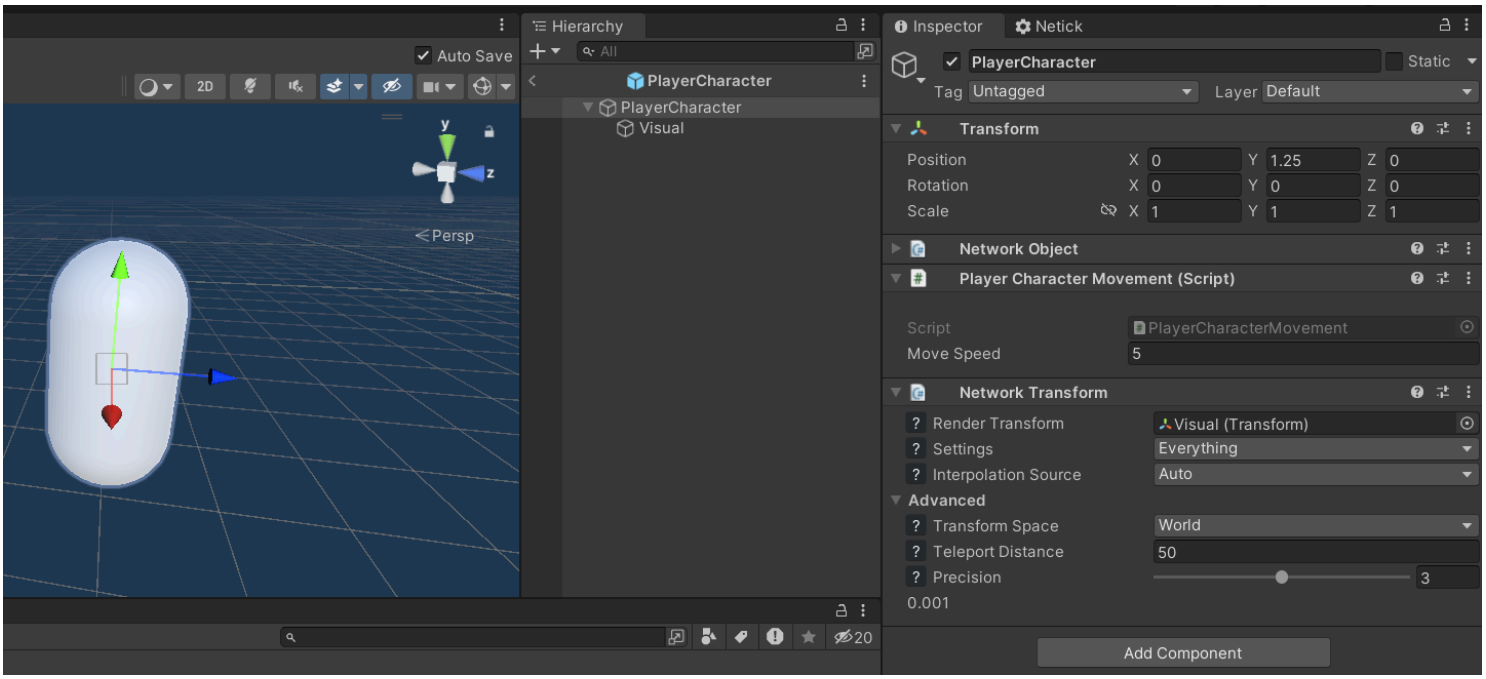
Gameplay & Visual Seperation

In the `NetworkTransform` component, there is a `Render Transform` field which asks for a `Transform`. Because Netick is a tick-based netcode, it means movement will happen at a fixed rate which is lower than your FPS. Which will cause unsmooth movement. To fix this, we use interpolation which lets us give smoothed position and rotation to our player character visual.

1. Create a child on the player and name it "Visual".
2. Delete & Move the `Capsule (Mesh Filter)` and `Mesh Renderer` component to Visual.
3. Assign Visual to `Render Transform` of `NetworkTransform`.

Learn More About Interpolation

Here's what our player character object looks like now:



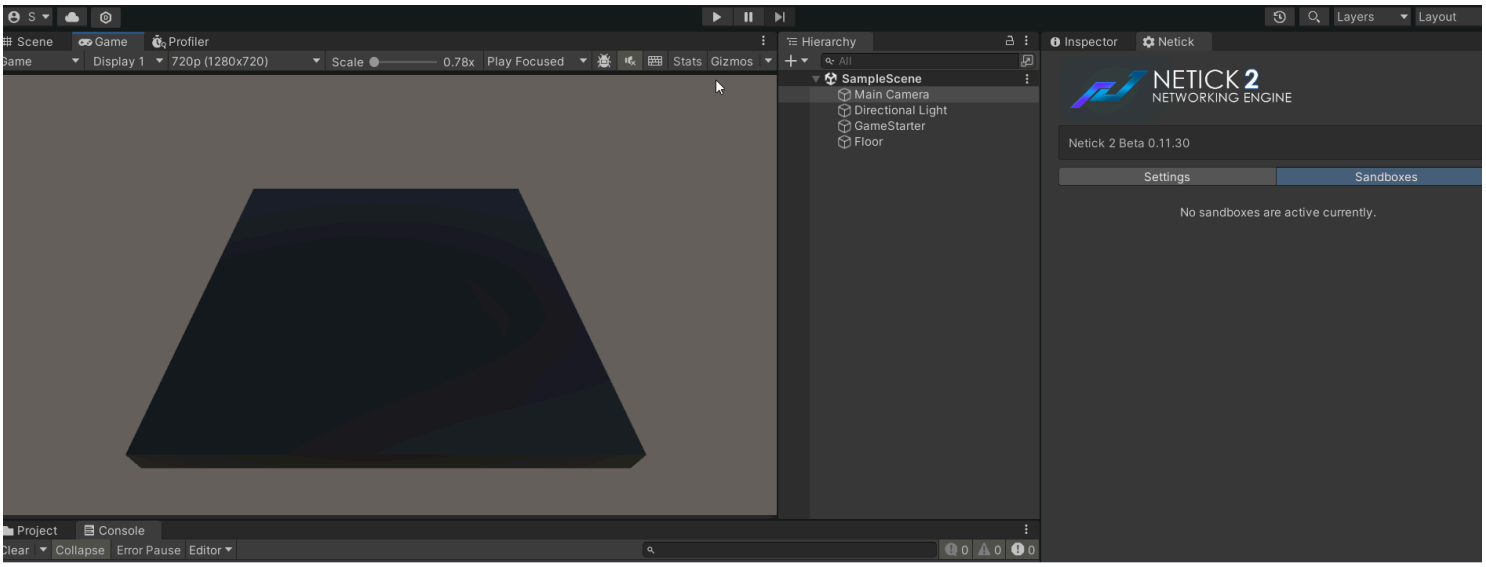
Multiplayer Testing

Let's try to run the game now. In Netick, there is something called Sandboxing (or multi-peer), that allows us to simulate multiple peers on a single Unity process, meaning we don't have to build the game (or use two editors) to test multiplayer. Instead, we run a server and a client (or multiple) together in the same project at the same time.

1. Enter play mode.
2. Click on `Run Host + Client` .
3. Click `Connect` .

You can show/hide sandboxes in `Netick > Sandboxes` .

[Learn More About Sandboxing]



4 - Network Property

A network property allows us to replicate things in the game and keep them in sync across the network. In this tutorial, we are going to replicate our mesh color between players using inputs and network property.

Learn More About Network Properties

Color Input

Let's add one more type of input which is a `bool` and give it the name of `randomizeColor`. If this bool is true, then we will randomize the color.

```
public struct PlayerCharacterInput : INetworkInput
{
    //...
    public bool RandomizeColor;
}
```

Let's modify our `GameplayManager` to also set the `randomizeColor` using the Space key.

```
public class GameplayManager : NetworkEventsListener
{
    //...
    public override void OnInput(NetworkSandbox sandbox)
    {
        //...
        input.RandomizeColor = Input.GetKey(KeyCode.Space);

        sandbox.SetInput(input);
    }
    //...
}
```

Defining a Network Property

1. Create a new C# script and name it `PlayerCharacterVisual`.
2. Replace the parent class from `MonoBehaviour` to `NetworkBehaviour`.
3. Declare a network property of a `Color` type.

```
using UnityEngine;
using Netick;
using Netick.Unity;

public class PlayerCharacterVisual : NetworkBehaviour
{
    [Networked] public Color MeshColor { get; set; }
}
```

Note that you must make your variable a property by adding `{get; set;}` to its end, this is used by Netick to make it synced automatically.

4. Let's use the `FetchInput` method to handle the color changing logic. When `RandomizeColor` field of the input is true, we generate a random color and assign it to the `MeshColor` network property.

```
using UnityEngine;
using Netick;
using Netick.Unity;

public class PlayerCharacterVisual : NetworkBehaviour
{
    [Networked] public Color MeshColor { get; set; }

    public override void NetworkFixedUpdate()
    {
        if (FetchInput(out PlayerCharacterInput input))
        {
```

```
            if (input.RandomizeColor)
                MeshColor = Random.ColorHSV(0f, 1f);
        }
    }
}
```

5. Declare a field of `MeshRenderer` .

Detecting Changes

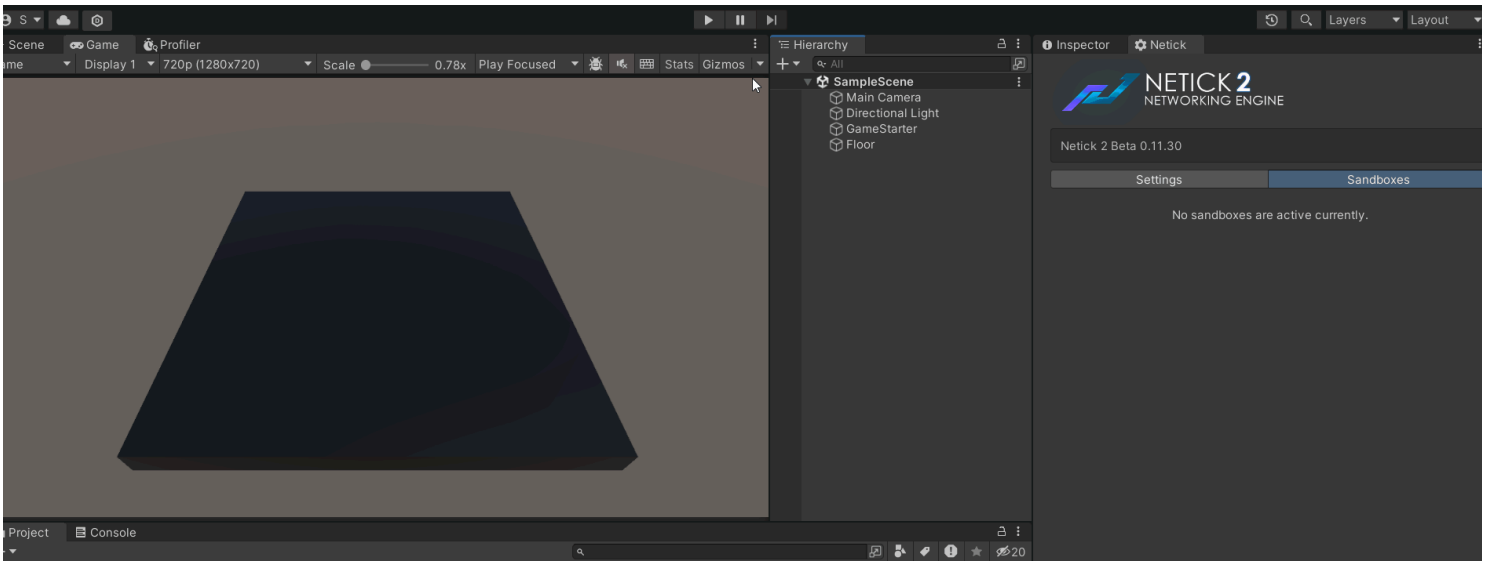
Netick lets you automatically detect whenever a certain network property changes, which is by using the `[OnChanged]` attribute on a method that will be invoked when the specified property changes.

1. Create a method and name it `OnColorChanged` with `OnChangedData` parameter.
2. Add `[OnChanged]` attribute on top of the method.
3. Supply the property name we want to detect inside the `[OnChanged]` attribute which is `MeshColor` .
4. Update the material color on `OnColorChanged` .

```
using UnityEngine;
using Netick;
using Netick.Unity;

public class PlayerCharacterVisual : NetworkBehaviour
{
    [Networked] public Color MeshColor { get; set; }

    public MeshRenderer meshRenderer;

    public override void NetworkFixedUpdate()
    {
        if (FetchInput(out PlayerCharacterInput input))
        {
            if (input.RandomizeColor)
                MeshColor = Random.ColorHSV(0f, 1f);
        }
    }

    [OnChanged(nameof(MeshColor))]
    private void OnColorChanged(OnChangedData onChangedData)
    {
        meshRenderer.material.color = MeshColor;
    }
}
```

Learn More About OnChanged

Don't forget to assign the `meshRenderer` field in our player component!



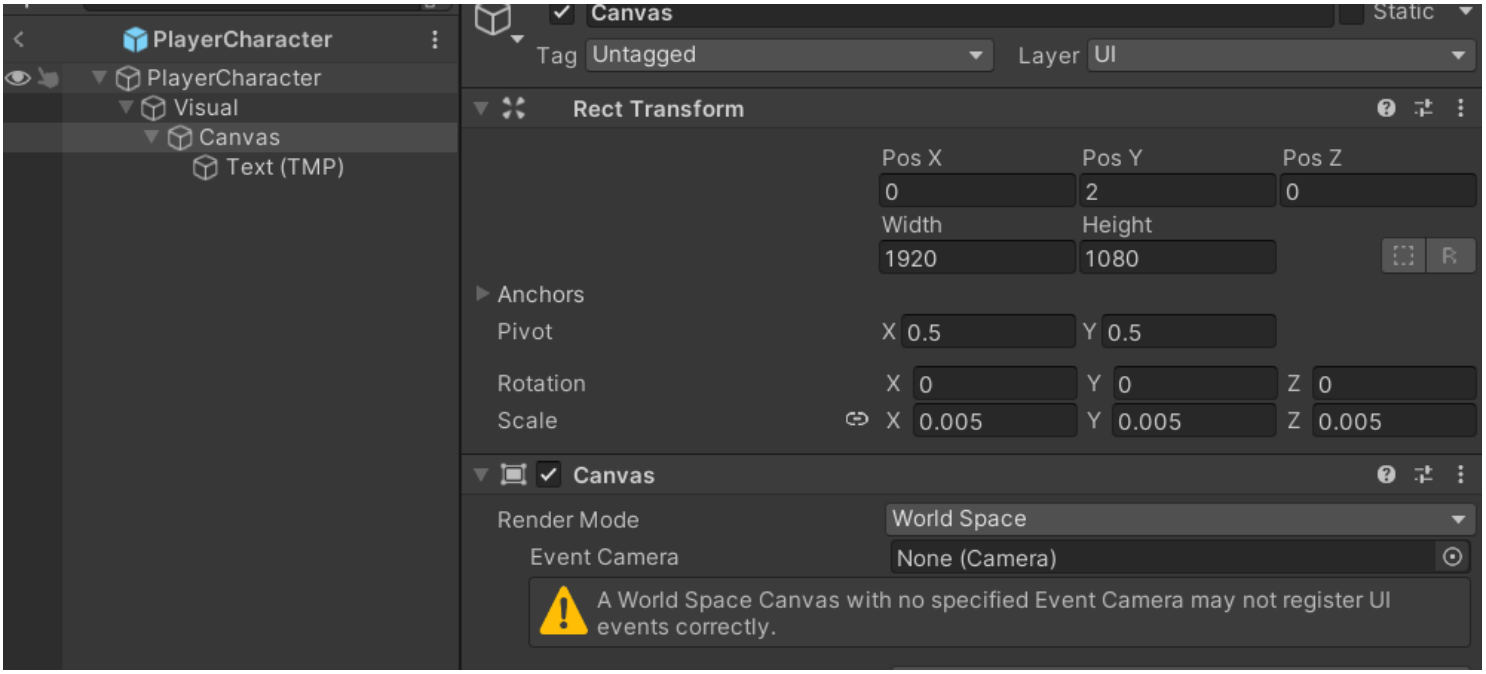
5 - Remote Procedure Call

In this tutorial, we are going to use an RPC (Remote Procedure Call) to set our nickname randomly. RPC is the most primitive way to sync things in the game. It's not recommended to use RPCs most of the time, and should only be used for infrequent actions like sending the player's name to the server.

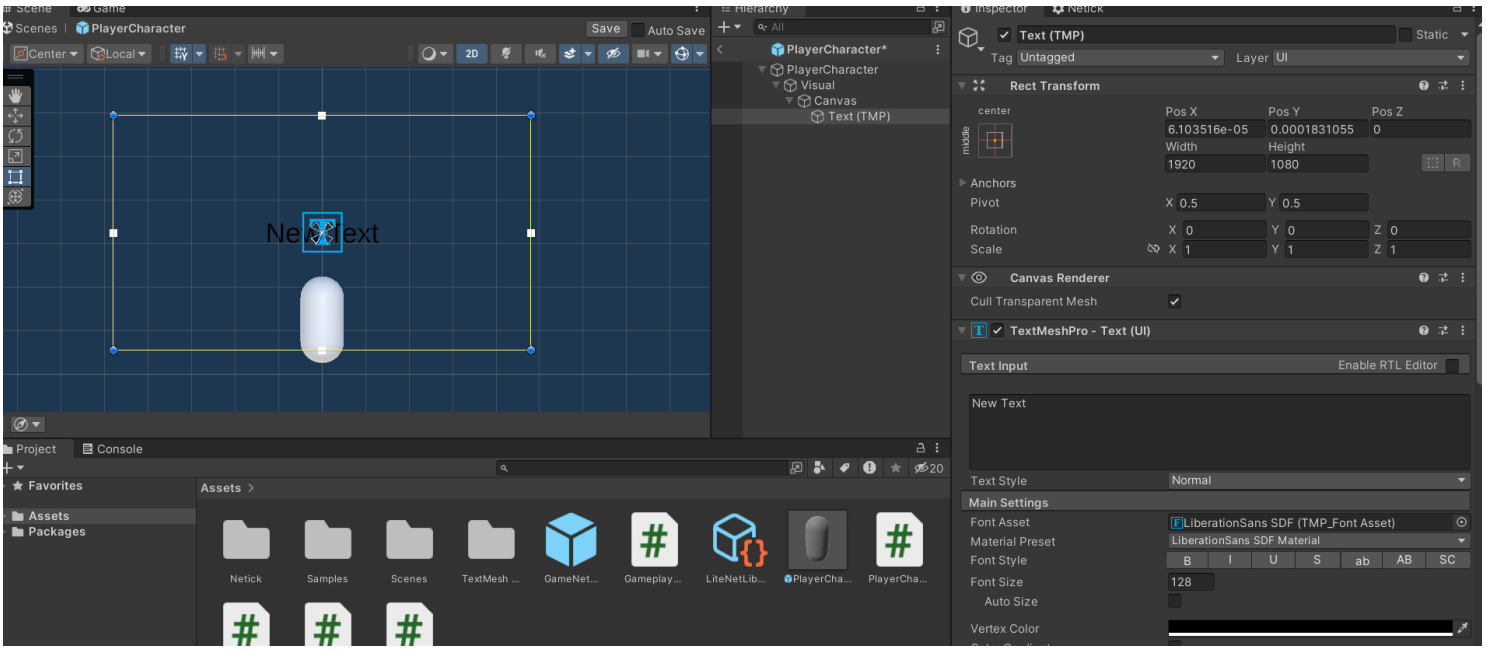
[Learn More About RPCs]

UI Setup

1. In PlayerCharacter prefab, on the `Visual` transform, add a `UI > Text - TextMeshPro` .
2. It might asks you to `Import TMP Essentials` , import it and close the window afterwards.
3. Change the `Canvas` `Render Mode` from `Screen Space - Overlay` to `World Space` .
4. Position your canvas to be Pos X: `0` Pos Y: `2`
5. Change the canvas scale to `0.005` for all axis.



6. In your text component, change font size to `128` , and middle & center alignment. Make sure your text now is in the center.



PlayerCharacterNametag

1. Create a new script and name it `PlayerCharacterNametag`.
2. Change parent class to `NetworkBehaviour`.
3. Create a network property named Nickname and give it the type of `NetworkString32` (`string` works too, however you should always use NetworkString as `string` can't work as an RPC parameter or a struct field).

RPC Implementation

We're going to set the RPC source to `InputSource` and the target to `Owner` (Server/Host). This means only the input source peer is able to call this RPC, but only the server will execute the RPC. We also want to set `isReliable` to true, this will ensure that this RPC will arrive to the server, even if packet loss occurs.

```
using UnityEngine;
using Netick;
using Netick.Unity;

public class PlayerCharacterNametag : NetworkBehaviour
{
    [Networked] public NetworkString32 Nickname { get; set; }

    [Rpc(source: RpcPeers.InputSource, target: RpcPeers.Owner, isReliable: true)]
    public void RPC_SetNicknameRandom()
    {
        Nickname = new NetworkString32($"Player_{Random.Range(1000, 9999)}");
    }
}
```

Calling the RPC

RPCs can be called from any place. We're going to call them from inside `NetworkUpdate` (not to be confused with `NetworkFixedUpdate`) which is just a regular Unity `Update`. We only want to call the RPC if we have the input authority and if we press the Enter keycode.

```
public class PlayerCharacterNametag : NetworkBehaviour
{
    //....

    public override void NetworkUpdate()
    {
        if (IsInputSource && Input.GetKeyDown(KeyCode.Return) && Sandbox.InputEnabled)
        {
            RPC_SetNicknameRandom();
        }
    }

    //...
}
```

Nickname OnChanged

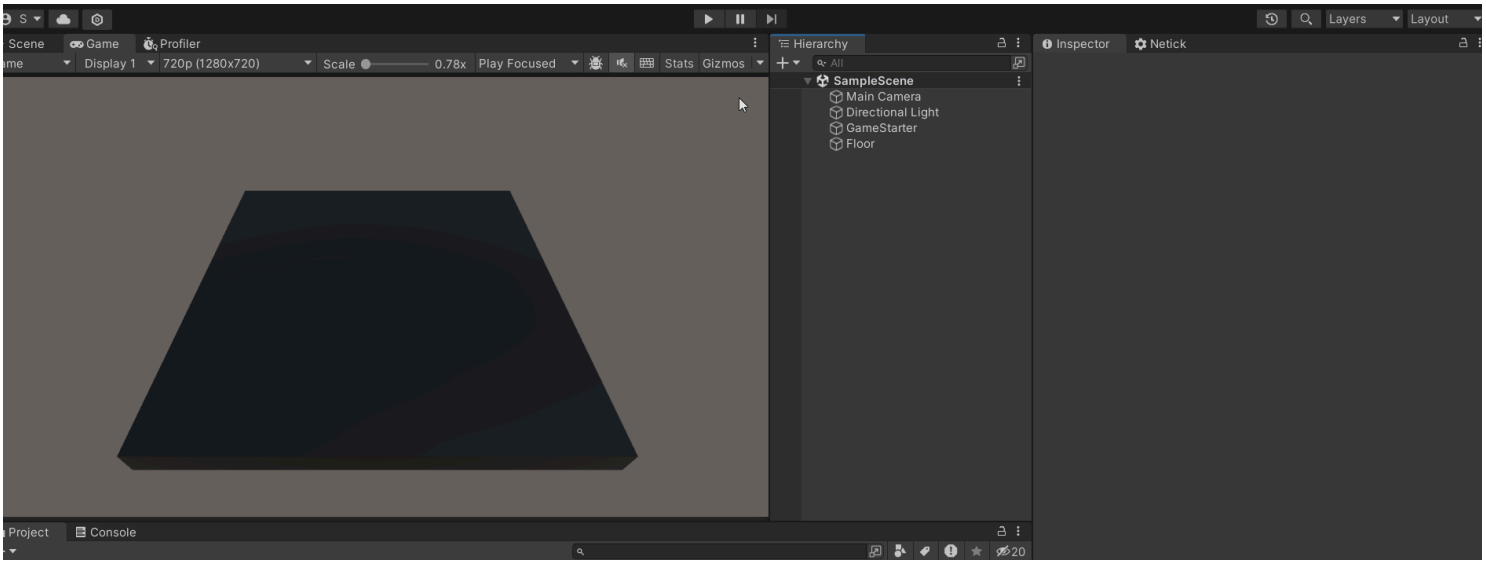
Then, we're going to use an OnChanged callback for our Nickname:

```
public class PlayerCharacterNametag : NetworkBehaviour
{
    //...
    public TMP_Text TextNametag;
    //...

    [OnChanged(nameof(Nickname))]
    private void OnNicknameChanged(OnChangedData onChangedData)
    {
        TextNametag.SetText(Nickname);
    }

    //...
}
```

Assign the TextNametag with the TextMeshPro UI we have created before.

Final Testing

Let's press the Enter key repeatedly to check if the RPC is working.



6 - Next

There are still many things which remain unexplored, but for now you should be able to experiment and learn yourself.

If you find yourself stuck, feel free to join our discord server and we will help you!

Go to Manual

More Learning Resources

For a full beginners video guide, this is a Udemy course done by one our community members on how to make a 3D top-down shooter using Netick:

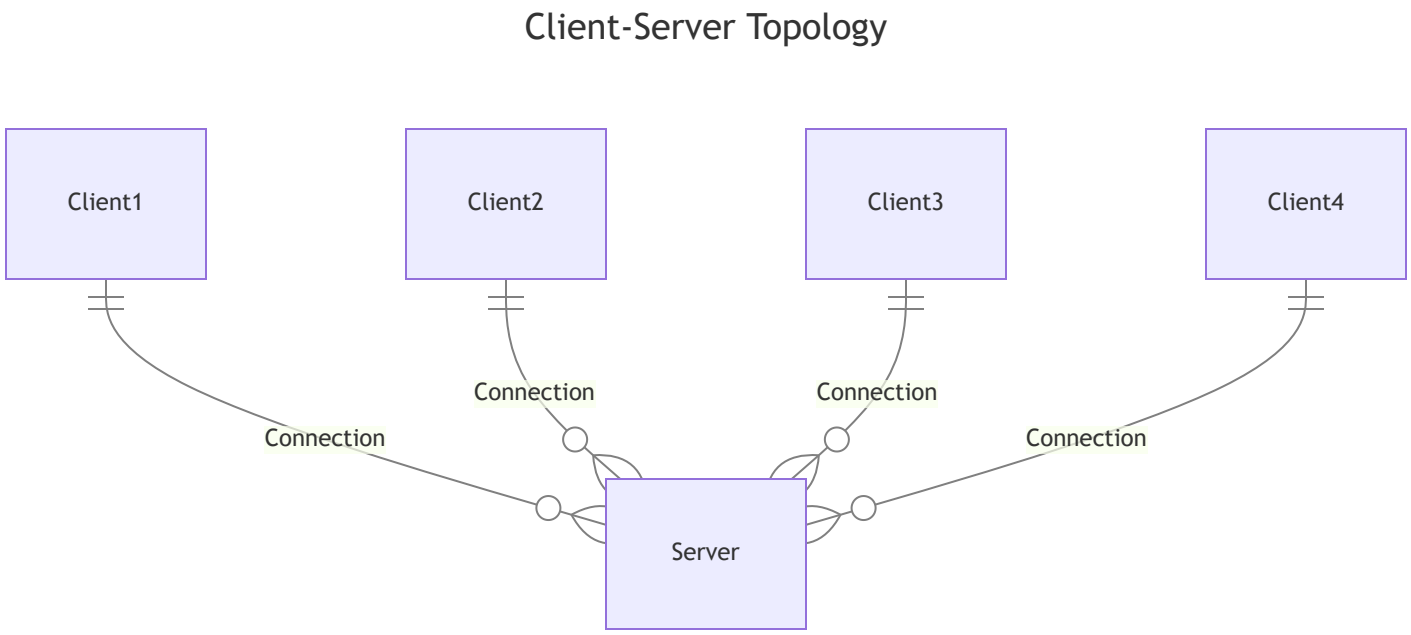
Link

Understanding Client-Server Model

In single-player games, developers typically don't need to enforce strict validation of player actions or behaviors. Since the game runs entirely on the player's local machine, preventing cheating isn't feasible—players have full access to all game data and logic. With no central authority to enforce the game's rules and logic, any form of security or validation is effectively meaningless.

This challenge becomes more apparent in peer-to-peer multiplayer setups, where all players are directly connected to each other and can independently simulate the game. In such models, each client can interpret the game state differently, making cheating trivial and trust between peers difficult to maintain.

This is where the client-server model comes in.



Instead of connecting players to each other, each client connects to a single authoritative node: the server. In an ideal client-server setup, clients do not simulate gameplay on their own. Instead, they send input commands (e.g., movement, shooting) to the server. The server processes these inputs, updates the game state, and sends the results back to the clients.

Because the server is solely responsible for executing game logic, enforcing rules, and validating inputs, clients are unable to unilaterally alter the game. This architecture significantly reduces opportunities for cheating and ensures consistent gameplay across all clients.

Netick follows this model by implementing a server-authoritative architecture combined with client-side prediction. You'll soon learn how to construct input and state systems to take full advantage of this approach and build secure, responsive multiplayer gameplay.

Core Concepts

Network Sandbox

NetworkSandbox is what controls the whole network game. It can be thought of as representing an instance of the game. You can have more than one network sandbox in a single Unity game, and that happens when you start both a client and a server on the same project. This can be extremely useful for testing/debugging, because it allows you to run a server and a client (or multiple thereof) in the same project and therefore see what happens at both at the same time, without interference.

- Therefore you can think of a sandbox as representing a server or a client.
- You can show/hide the current sandboxes from the Network Sandboxes panel.

Network Object

Any GameObject which needs to be synced/replicated must have a NetworkObject component added to it. If you want to see something on everyone's screen, it has to have a NetworkObject component added to it. It's the component that tells Netick that a GameObject is networked. The NetworkObject component by itself just informs Netick that the object is networked. To add networked gameplay-logic to it, you must do so in a component of a class derived from NetworkBehaviour. Netick comes with a few essential built-in components:

- NetworkTransform: used to sync position and rotation
- NetworkRigidbody: used to sync controllable physical objects
- NetworkAnimator: used to sync Unity's animator's state

Network Behaviour

The NetworkBehaviour class is your old friend `MonoBehaviour`, just the networked version of it. To implement your networked functionality, create a new class and derive it from NetworkBehaviour. You have several methods you can override which correspond to Unity's non-networked equivalents (they must be used instead of Unity's equivalents when doing anything related to the network simulation):

- NetworkStart
- NetworkDestroy
- NetworkFixedUpdate
- NetworkUpdate
- NetworkRender

Example:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Netick;
using Netick.Unity;

public class MyBehaviour : NetworkBehaviour
{
    [Networked]
    public int   IntPropertyExample   { get; set;}
    [Networked]
    public float FloatPropertyExample { get; set;}

    public override void NetworkStart()
    {
        // Called when this object has been added to the simulation.
    }

    public override void NetworkDestroy()
    {
        // Called when this object has been removed from the simulation.
    }

    public override void NetworkUpdate()
    {
        // Called every frame. Executed before NetworkFixedUpdate.
    }

    public override void NetworkFixedUpdate()
    {
        // Called every fixed-time network step. Any changes to the networked state should happen here.
```

```
        // Check out the chapter named "Writing Client-Side Prediction code" to learn more about this method.
    }

    public override void NetworkRender()
    {
        // Called every frame. Executed after NetworkUpdate and NetworkFixedUpdate.
        // IMPORTANT NOTE: properties (which can be interpolated) marked with [Smooth] attribute will return interpolated values when accessed in
this method.
    }
}
```

Don't forget to include `using Netick` and `using Netick.Unity` .

A class derived from NetworkBehaviour is of limited use without the utilization of Network Properties, which are the building blocks of your networked synced state. Network Properties are delta compressed, letting you create objects with complex states and not worry about it.

Networked State

Networked state is the data of the game that you want to replicate to players. In Netick, networked state is delta compressed, therefore only changes are replicated. If a field of a struct changes, only that field is replicated. If a counter increases, only the delta to the previous value is replicated. If your counter was at 32534536, and now it is at 32534537, it will be replicated as a delta of one. It applies to vectors and quaternions too. Thus, using as little bandwidth as possible.

Every networked variable can be predicted and interpolated too. Allowing you to create complex networked systems easily.

Network Properties

A network property is a C# property which is replicated across the network. For a property to be networked, the attribute [Networked] must be added to it.

Examples of networked properties:

```
[Networked]
public int              Health   {get; set;}

[Networked]
public float            Speed    {get; set;}

[Networked]
public Vector3          Velocity {get; set;}

[Networked]
public int              Ammo     {get; set;}

[Networked]
public NetworkBool      IsAlive  {get; set;}

[Networked]
public NetworkString32  Name     {get; set;}
```

> **WARNING**
>
> Reference types are not networkable.

> **WARNING**
>
> If you are intending on building your game using `IL2CPP`, you must use `NetworkBool` instead of `bool`.

> **NOTE**
>
> Don't use types with sizes lower than 4 bytes such as `byte` or `short`, instead simply use `int`. Netick already compresses everything to only the required bits of the current value of a variable. So if an `int` variable current value is `5`, it will only be serialized as a few bits, not anywhere near 4 bytes (the byte size of `int`).

Network Arrays

Network arrays are just like regular C# arrays, but their syntax is a bit different. They are defined using the NetworkArray<T> generic class.

Example of a network array:

```
[Networked(size: 5)]
public readonly NetworkArray<int> IntArrayExample = new NetworkArray<int>(5) { 55, 66, 77 };
```

> **WARNING**
>
> `size` of `[Networked(size: 32)]` must be the same as the value that is passed to the array constructor `new NetworkArray<int>(32)` .

Network Array Structs

Netick 2 introduces a new type of network array, network arrays that are completely value types - Network Array Structs. These are fixed-size struct arrays available only in 4 fixed sizes: 8, 16, 32, and 64.

Network Array Structs are pretty useful since they can used as members of another struct, or even nested inside other arrays. In addition, they can be sent as RPC parameters.

```
// Network Array Struct Examples

[Networked]
public NetworkArrayStruct8<int>                    IntFixedArray { get; set; } = new int[] {1 , 4 ,5}.ToNetworkStructArray8();

[Networked]
public NetworkArrayStruct8<NetworkArrayStruct8<int>> ArrayOfArrays { get; set; };
```

> **NOTE**
>
> Network Array Structs are treated as if they were simple struct types like `int` or `float` , so they must be defined as a property not as a field (like `NetworkArray<T>` that is non-fixed size).

Changing Elements of Network Array Struct

Because Network Array Structs are structs, the whole array will be replaced even when you change a single element. To avoid bugs, this should be how you change array elements:

```
IntFixedArray = IntFixedArray.Set(index, value);
// as you can see, we are reassigning the property with the new changed array which has the change.
```

Network Collections

In addition to `NetworkArray<T>` , Netick also has alternatives to C# collections that are fully synced, predicted, and interpolated.

- `NetworkLinkedList<T>`
- `NetworkDictionary<TKey,TValue>`
- `NetworkHashSet<T>`
- `NetworkUnorderedList<T>`
- `NetworkStack<T>`
- `NetworkQueue<T>`

In terms of bandwidth usage, the most expensive collection is `NetworkDictionary` , while the cheapest is `NetworkUnorderedList` (excluding `NetworkArray`). The order from the most expensive to the cheapest is: `NetworkDictionary` > `NetworkLinkedList` > `NetworkHashSet` > `NetworkQueue` > `NetworkStack` > `NetworkUnorderedList` . However, note that this only relates to the bandwidth usage when adding/removing elements, as all collections (or any networked variable) use no bandwidth or CPU time when they're idle and not changing.

Usage examples:

```
[Networked(size: 16)]
public readonly NetworkDictionary<int, int>  MyNetworkDictionary    = new(16);

[Networked(size: 16)]
public readonly NetworkHashSet<int>          MyNetworkHashSet       = new(16);

[Networked(size: 16)]
```

```
public readonly NetworkLinkedList<int>       MyNetworkLinkedList    = new(16);

[Networked(size: 16)]
public readonly NetworkUnorderedList<int>    MyNetworkUnorderedList = new(16);

[Networked(size: 16)]
public readonly NetworkStack<int>            MyNetworkedStack       = new(16);

[Networked(size: 16)]
public readonly NetworkQueue<int>            MyNetworkedQueue       = new(16);
```

Removing and adding elements is the same as with C# collections.

> **NOTE**
>
> The `size` that you pass to `[Networked]` and the constructor represents the fixed capacity of the collection. The collections don't support resizing as all network state sizes are set at compile time.

Network Structs

Netick can synchronize any struct that does not contain class-based arrays or references. Which includes all C# primitive types and Unity/Godot/Flax primitive types.

Example:

```
[Networked]
public struct MyNestedStruct
{
    public int                      Int;
    public NetworkBool              Bool;
    [Networked]
    public float                    Float    { get; set;}
    [Networked]
    public Vector3                  Position { get; set;}
    [Networked]
    public Quaternion               Rotation { get; set;}
    [Networked]
    public Color                    Color    { get; set;}
    public NetworkString8           Name;
}

[Networked]
public struct MyStruct
{
    public MyNestedStruct           MyNestedStruct;
    public NetworkArrayStruct8<int> StructArray;
    public int                      Int;
    public NetworkBool              Bool;
    [Networked]
    public float                    Float    { get; set;}
}

[Networked]
public MyStruct MyStructProperty {get; set;}
```

> **NOTE**
>
> `string` is not supported as a type that can be used inside a struct. Use `NetworkString` instead.

> **NOTE**
>
> `[Networked]` attribute on structs is optional. However, when adding it to a struct, it allows float-based types (such as `float` or `Vector3`, which also have `[Networked]` on them) of a struct to have extra compression on them.

Networking References to `NetworkObject` and `NetworkBehaviour`

Since you can't directly synchronize class references, we provide two helper structs that are used to synchronize a reference to `NetworkObject` and `NetworkBehaviour`:

`NetworkObjectRef`

Usage Example:

```
public class PlayerController : NetworkBehaviour
{
    [Networked]
    public NetworkObjectRef MyPlayer { get; set;}

    public override void NetworkStart()
    {
        // assigning the ref
        MyPlayer = Object.GetRef();
    }

    public void ExampleOfUsingTheRef()
    {
        // getting the object from the ref
        var netObj = MyPlayer.GetObject(Sandbox); // or TryGetObject
    }
}
```

`NetworkBehaviourRef`

Usage Example:

```
public class PlayerController: NetworkBehaviour
{
    [Networked]
    public NetworkBehaviourRef<PlayerController> MyPlayer { get; set;}

    public override void NetworkStart()
    {
        // assigning the ref
        MyPlayer = this.GetRef<PlayerController>;
    }

    public void ExampleOfUsingTheRef()
    {
        // getting the behaviour from the ref
        var playerController = MyPlayer.GetBehaviour(Sandbox); // or TryGetBehaviour
    }
}
```

Replication Relevancy

You can choose to replicate a property only to the Input Source client of the object. This is done using the optional parameter `relevancy` of [Networked].

Example:

```
[Networked(relevancy: Relevancy.InputSource)]
public int              Ammo     {get; set;}
```

State Synchronization

Updates to the network state are atomic, it's not possible for a property to update in the client without other changed properties to update alongside it. If you change two (or more) properties in the server at the same time (or in two subsequent ticks), you are ensured to have both replicate together in the client. This makes it so that you don't have to worry about packet loss and possible race conditions that might occur due to some properties updates arriving while others arriving later. This simplifies how you program your game as you never have to worry about such things happening.

This also means that when you create an object in the server, assign some initial values to some network properties, when this object is created in the client, inside `NetworkStart` of that object you will have full initial state that you assigned in the server.

Change Callback

You can have a method get called whenever a networked variable changes, which is very useful. To do that, add the attribute [OnChanged] to the method and give it the name of the variable. The method must must have a parameter of OnChangedData type which can be used to retrieve the previous variable value.

For Properties

Example:

```
[Networked]
public int Health { get; set; }

[OnChanged(nameof(Health))]
private void OnHealthChanged(OnChangedData onChangedData)
{
  var previous = onChangedData.GetPreviousValue<int>();
}
```

For Arrays

Example:

```
[Networked(size: 16)]
public readonly NetworkArray<int> ArrayExample = new(16);

[OnChanged(nameof(ArrayExample))]
private void OnArrayExampleChanged(OnChangedData onChangedData)
{
  // getting the changed element value directly

  var changedPreviousElementValue = onChangedData.GetArrayPreviousElementValue<int>();

  // or just getting the index

  var changedPreviousElementIndex = onChangedData.GetArrayChangedElementIndex();

  // or maybe getting the previous value of another index we want

  var someRandomPreviousElementValue = onChangedData.GetArrayPreviousElementValue<int>(13);
}
```

For Collections

NetworkLinkedList

Example:

```
[Networked(size: 16)]
public readonly NetworkLinkedList<int> MyNetworkLinkedList = new(16);

[OnChanged(nameof(MyNetworkLinkedList))]
private void OnMyNetworkLinkedListChanged(OnChangedData onChangedData)
{
  // getting a snapshot of the previous state of the collection
  var previous = onChangedData.GetPreviousNetworkLinkedList(MyNetworkLinkedList);
}
```

NetworkDictionary

Example:

```
[Networked(size: 16)]
public readonly NetworkDictionary<int, int>  MyNetworkDictionary = new(16);

[OnChanged(nameof(MyNetworkDictionary))]
private void OnMyNetworkDictionaryChanged(OnChangedData onChangedData)
```

```
{
  // getting a snapshot of the previous state of the collection.
  var previous = onChangedData.GetPreviousNetworkDictionary(MyNetworkDictionary);
}
```

NetworkHashSet

Example:

```
[Networked(size: 16)]
public readonly NetworkHashSet<int>  MyNetworkHashSet = new(16);

[OnChanged(nameof(MyNetworkHashSet))]
private void OnMyNetworkHashSetChanged(OnChangedData onChangedData)
{
  // getting a snapshot of the previous state of the collection.
  var previous = onChangedData.GetPreviousNetworkHashSet(MyNetworkHashSet);
}
```

NetworkUnorderedList

Example:

```
[Networked(size: 16)]
public readonly NetworkUnorderedList<int>  MyNetworkUnorderedList = new(16);

[OnChanged(nameof(MyNetworkUnorderedList))]
private void OnMyNetworkUnorderedListChanged(OnChangedData onChangedData)
{
  // getting a snapshot of the previous state of the collection.
  var previous = onChangedData.GetPreviousNetworkUnorderedList(MyNetworkUnorderedList);
}
```

NetworkQueue

Example:

```
[Networked(size: 16)]
public readonly NetworkQueue<int> MyNetworkQueue = new(16);

[OnChanged(nameof(MyNetworkQueue))]
private void OnMyNetworkQueueChanged(OnChangedData onChangedData)
{
  // getting a snapshot of the previous state of the collection.
  var previous = onChangedData.GetPreviousNetworkQueue(MyNetworkQueue);
}
```

NetworkStack

Example:

```
[Networked(size: 16)]
public readonly NetworkStack<int> MyNetworkStack = new(16);

[OnChanged(nameof(MyNetworkStack))]
private void OnMyNetworkStackChanged(OnChangedData onChangedData)
{
  // getting a snapshot of the previous state of the collection.
  var previous = onChangedData.GetPreviousNetworkStack(MyNetworkStack);
}
```

> **WARNING**
>
> Don't use the array methods of `OnChangedData` on network collections. They only work on `NetworkArray<T>` .

> **WARNING**
>
> Be careful when using these methods on `OnChangedData` , since they are unsafe and can cause a crash if you go outside array range or use an incorrect type.

Finding Removed and Added Items to Collections

Using the previous snapshot (version) of the collection, we are able to compare the current collection against the previous snapshot to find the items that were added and the items that were removed.

Example:

This example uses a `NetworkDictionary` but the same applies to other collections.

```
[Networked(size: 10)]
public NetworkDictionary<int, Vector3> NetworkDictionaryExample = new NetworkDictionary<int, Vector3>(10);

[OnChanged(nameof(NetworkDictionaryExample))]
void OnNetworkDictionaryExampleChanged(OnChangedData dat)
{
  var previous = dat.GetPreviousNetworkDictionary(NetworkDictionaryExample);

  // finding the newly added items
  foreach (var item in NetworkDictionaryExample)
    if (!previous.ContainsKey(item.Key)) // does not exist in the previous version of the collection, meaning it's a new item.
      Debug.Log($"{item} was added!");

  // finding the newly removed items
  foreach (var item in previous)
    if (!NetworkDictionaryExample.ContainsKey(item.Key)) // if the current version of the collection does not have the item, it means it was removed.
      Debug.Log($"{item} was removed!");
}
```

Invoke Behavior of `[OnChanged]` Callbacks

- When you change a variable in the server or in the client (on a predicted object), the `[OnChanged]` method will be invoked from the setter of the networked variable, therefore it's immediately invoked when changing the variable.

- In the client, when the client receives data for a networked variable that was changed in the server, the client will also invoke the callback, but only if the received value is different from the current value or when there was a misprediction. A misprediction means the value of the variable before rollback is not equal to the value after rollback and resimulation. Read the article on Client-Side Prediction to learn more.

- If the server changes a variable multiple times, but then back to the original value before all of this, the client will not invoke the callback, because to the client that networked variable never changed, but to the server it did but it eventually went back to the same value at the start of the tick. Therefore it's important to realize that in this case the callback is invoked multiple times in the server but never in the client.

Invoking `[OnChanged]` Callbacks During Rollback & Resimulation

Read the article on Client-Side Prediction before reading this section.

By default, Netick will not invoke `[OnChanged]` callbacks when the client rolls back to the latest received server state, and neither during the resimulation stage of prediction. This is usually the desired behaviour because you only want the callback to fire when the value is changed for the first time (usually in the client, on predicted objects). However, sometimes you want the `[OnChanged]` callback to always be in sync with the value of the networked variable and have it also get invoked during rollback and during resimulation. This is easily possible by simply passing true to `invokeDuringResimulation` optional parameter of `[OnChanged]` .

Understanding Client-Side Prediction (CSP)

Tick-based Networking

Before diving into client-side prediction, it's essential to first understand tick-based networking.

In a multiplayer environment, clients and the server may run at vastly different framerates. To ensure consistent and synchronized gameplay across all machines, networked game logic is executed at a fixed interval known as the tickrate. This fixed simulation rate—similar to Unity's physics fixed timestep—ensures stable and proper behavior across the network.

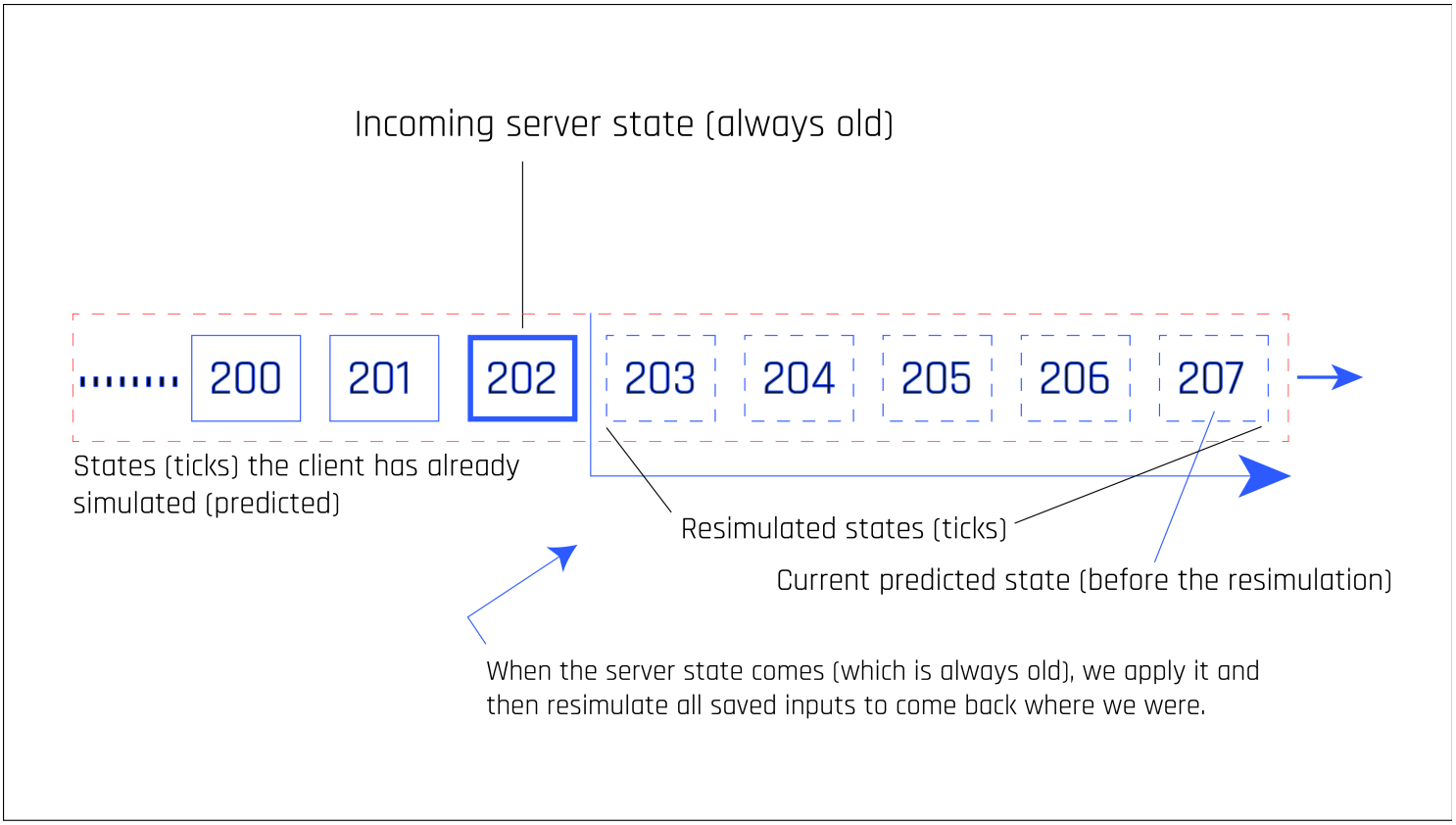
Each step of simulation at this interval is called a tick, representing a discrete moment in simulated time. By tying actions and updates to specific ticks, synchronizing a networked game becomes a lot simpler regardless of the various framerates each connected client runs at.

Client-Side Prediction (CSP)

In a client-server model, clients are not allowed to authoritatively modify the state of networked objects. Instead, the server alone owns the truth and is responsible for applying any changes. Because ultimately the client's executable can be tampered with or modified. In other words, only the server can ever change the true state of network variables. What the client does to affect changes to the networked state is send inputs which are later simulated by the server to produce the desired state which is sent back to all clients.

This model ensures security and fairness, as the server cannot be easily tampered with. However, it also introduces an issue: input delay. Due to internet latency (round-trip time), waiting for the server to simulate input leads to a sluggish and unresponsive experience.

To solve this, modern multiplayer systems use Client-Side Prediction (CSP).



With CSP, instead of waiting for the server to simulate inputs, the client predicts the outcome locally by executing its inputs immediately. When the actual state from the server arrives:

- The client rolls back to the last known server state.
- It resimulates all stored inputs from that point forward.

All of this happens within a single tick, ensuring a smooth and immediate experience for the player while still maintaining server authority.

Simulation Logic

All tick-based simulation must be implemented within `NetworkFixedUpdate` in a `NetworkBehavior`. This method is invoked every network tick to advance the simulation.

- On the server, `NetworkFixedUpdate` is called once per new tick to process fresh inputs.
- On the client, it may be called multiple times in a single tick to resimulate all inputs beyond the last confirmed server tick.

Resimulation Targets

Resimulation on the client only applies to:

- Objects for which the client is the Input Source
- Objects with `PredictionMode` set to `Everyone`, meaning all clients predict them, not just the Input Source

For all other objects, `NetworkFixedUpdate` is executed once per tick, with no resimulation.

> **NOTE**
>
> The server never resimulates past ticks—it only processes new inputs for the current tick. CSP is strictly a client-side mechanism. From the server's perspective, it's akin to running a single-player game simulation.

For basic movement, the side effects of resimulation are typically negligible. However, when handling actions like shooting, playing audio, or showing visual-only effects, it's crucial to ensure these actions only occur the first time an input is simulated. Failing to do this can result in actions being triggered multiple times due to repeated resimulations.

Note that it's usually impractical to predict everything the client does in the game, and it's simpler to:

- Avoid predicting specific features entirely (e.g., entering a vehicle or buying a weapon), and wait for the server response
- Make certain features client-authoritative, especially if mispredictions would be complex to reconcile

You don't need to make the entire game server-authoritative. Only enforce authority where it's critical to gameplay integrity—such as competitive actions, player movement, or sensitive game state.

Writing Gameplay Code

Network Input

Network Input describes what the player wants to do, which will be used to simulate the state of objects they want to control. This ensures that the client can't directly change the state – the change happens by executing the input, which, even if tampered with, won't be game-breaking.

Defining Inputs

To define a new input, create a struct that implements `INetworkInput` interface:

```
[Networked]
public struct MyInput : INetworkInput
{
   [Networked] // adding [Networked] to a struct field and making it a property allows Netick to provide extra compression to it.
   public Vector2     Movement { get; set;}
   public NetworkBool Shoot;
}
```

Network Input Constraints:

- Must not have class-based network collections as fields. Only `NetworkArrayStruct` variants are allowed as network input fields.
- Must not have reference types as fields.
- Must not have `string` as a field. Instead, you can use `NetworkString` variants.

Setting Inputs

To set the fields of an input, you first need to acquire the input struct of the current tick, using `Sandbox.GetInput`. Then, you can set it inside `NetworkUpdate` on `NetworkBehaviour`:

```
public override void NetworkUpdate()
{
    var input       = Sandbox.GetInput<MyInput>();
    input.Movement += new Vector2(Mathf.Clamp(Input.GetAxis("Horizontal"), -1f, 1f), Mathf.Clamp(Input.GetAxis("Vertical"), -1f, 1f));
    input.Shoot    |= Input.GetMouseButton(0);
    Sandbox.SetInput(input);
}
```

You could also set them on `OnInput` of `NetworkEventsListener`, which is preferred.

Simulating Inputs

To drive the gameplay based on the input struct, you must do that in `NetworkFixedUpdate`:

```
public override void NetworkFixedUpdate()
{
    if (FetchInput(out MyInput input))
    {
        // clamp movement inputs
        input.Movement     = new Vector3(Mathf.Clamp(input.Movement.x, -1f, 1f), Mathf.Clamp(input.Movement.y, -1f, 1f));

        // movement
        var movementInputs = transform.TransformVector(new Vector3(input.Movement.x, 0, input.Movement.y)) * Speed;
        _CC.Move(movementInputs * Time.fixedDeltaTime);

            // shooting
        if (input.ShootInput == true && !IsResimulating)
            Shot();
    }
}
```

To solidify your understanding, here's another more abstracted example:

```
public override void NetworkFixedUpdate()
{
    if (FetchInput(out MyInput input))
    {
        // predicted action (movement).
        Move(input);

        // non-predicted action (interacting with a world object like a vehicle or a pickup).
        if (IsServer)
            if (input.Interact)
                TryToInteract();

        // predicted action, but not resimulated (shooting).
        if (!IsResimulating)
            if (input.ShootInput)
                Shot();
    }
}
```

Here, we see three types of actions:

1. **Predicted**: used for actions like movement. It's the default case.

2. **Non-Predicted**: used for actions that are best left unpredicted. Let's use riding a vehicle as an example. Even though this action can be predicted, it's usually not, to avoid conflicts where multiple players predict that they entered the vehicle as a driver only for it to be mispredicted because another player did the action first, which can be very frustrating and look bad. This is accomplished by making the code only runs in the server using `IsServer`.

3. **Predicted but not resimulated**: used for actions that must only happen during the first time ever when a tick is executed, and not during its resimulations. Usually you use this for things like shooting, to avoid the sound effect playing more than once or the visual effects spawning multiple times.

> **NOTE**
>
> Everything that is modified around `FetchInput`, and also affects the networked state, must be networked using `[Networked]`. For example, if you have a variable that is changing over time which can affect the speed of the player, it must be networked.

> **NOTE**
>
> When `IsResimulating` equals to true, every `[Networked]` variable has an older value, since we are resimulating past ticks. The first resimulated tick will have the server state applied to every networked variable.

> **WARNING**
>
> `Sandbox.GetInput` and `Sandbox.SetInput` are used to read and set the user inputs into the input struct. While `FetchInput` is used to actually use the input struct in the simulation.

> **NOTE**
>
> Make sure to clamp inputs to prevent malicious attempts to alter inputs to have big magnitudes leading to speedhacks. Inputs are the only thing the client has authority over.

`FetchInput` tries to fetch an input for the state/tick being simulated/resimulated. It only returns true if either:

1. We are providing inputs to this object – meaning we are the Input Source of the object.

2. We are the owner (the server) of this object – receiving inputs from the client who's the Input Source. And only if we have an input for the current tick being simulated. If not, it would return false. Usually, that happens due to packet loss.

And to avoid the previous issue we talked about, we make sure that we are only shooting if we are simulating a new input, by checking `IsResimulating`.

Input Source

For a client to be able to provide inputs to be used in an Object's `NetworkFixedUpdate`, and hence take control of it, that client must be the Input Source of that object. Otherwise, `FetchInput` will return false. To check if you are the Input Source, use `IsInputSource`.

The server can also be the Input Source of objects, although it won't do any CSP, since it needs not to, after all, it's the server.

You can set the Input Source of an object when instantiating it:

```
Sandbox.NetworkInstantiate(PlayerPrefab, spawnPos, Quaternion.identity, client);
```

Changing the Input Source

To set the Input Source of the object (must only be called on the server):

```
Object.InputSource = newInputSource;
```

To remove the Input Source of the object (must only be called on the server):

```
Object.InputSource = null;
```

Callbacks

There are two methods you can override to run code when Input Source has changed or left the game:

1. `OnInputSourceChanged` : called on the Input Source and server when the Input Source changes.
2. `OnInputSourceLeft` : called on the owner (server) when the Input Source client has left the game.

RPCs vs Inputs for Client->Server Actions

Other networking solutions rely heavily on RPCs for communication between clients and the server. However, Netick offers a more robust, secure, and streamlined alternative that often makes most RPCs unnecessary.

Example: Interacting with a Pickup

Imagine a scenario where a player needs to interact with an object in the world—such as picking up an item. In traditional approaches, this might involve sending an RPC from the client to the server, which comes with a few drawbacks:

- Higher bandwidth usage, as each RPC transmits additional metadata.
- Security risks, since a malicious client could invoke RPCs to interact with objects outside their legitimate range.

With Netick, a cleaner and safer solution is to use a networked input field, such as `Interact` :

```
public override void NetworkFixedUpdate()
{
    if (FetchInput(out MyInput input))
    {
        // movement
        ExecuteMovementLogic(input);

        if (input.Interact && IsServer) // adding IsServer when you don't want the client to predict this action.
        {
          if (Raycast(..., out var objectToInteractWith)
          {
            // do things
          }
        }
```

```
    }
}
```

This way, you have full server-authority on what objects the client can interact with, and the code is very simple to read and debug. It's almost the exact same code you would use for a single-player game, excluding the input fetching logic.

Supporting Multiple Local Players

Many games—such as party games—support multiple players on a single device. In Netick, the concept of a Network Player refers to a single peer or machine, and therefore doesn't directly account for multiple local players on the same peer.

To handle this, you can define a local player index to distinguish between players sharing the same machine. Netick facilitates this pattern by including an input index parameter in all input-related methods, which can be used as the local player index.

```
Sandbox.GetInput(localPlayerIndex);
```

```
Sandbox.SetInput(localPlayerIndex);
```

```
FetchInput(localPlayerIndex, out MyInput input);
```

Framerate Lower than Tickrate

When the framerate drops below the tickrate, multiple simulation ticks must be processed within a single rendered frame. In such cases, the same input must be reused across multiple ticks.

If this scenario is not handled correctly, it can lead to inconsistent simulation results—most notably, player characters may appear to move slower at very low framerates due to insufficient input sampling.

Enabling `Input Reuse At Low FPS` in Netick Settings

Enabling this option would automatically let Netick reuse/duplicate the same input of one frame to one or more ticks. You can also know if the input fetched is a duplicated input or not as follows:

```
public override void NetworkFixedUpdate()
{
    if (FetchInput(out MyInput input, out bool isDuplicated))
    {
        if (!isDuplicated)
        {
            // do stuff when this input is not a duplicate.
        }
    }
}
```

Detailed Breakdown of What Happens in a Single Tick

> **NOTE**
>
> If you are new to Netick, you can ignore this section.

To deeply understand CSP within Netick, let's take a look at what happens during a single tick:

A. Start of Tick

The start of a new tick.

B. Rollback

The client applies the latest received authoritative server state. `Sandbox.Tick` is changed into `Sandbox.AuthoritativeTick`. Prior to this, `Sandbox.Tick` matched `Sandbox.PredictedTick`.

During this step:

- `NetcodeIntoGameEngine` of `NetworkBehaviour` is invoked on all predicted objects to integrate the networked state into Unity components (e.g., Transform).
- As a result, all predicted objects, in the interest list of the client, are reverted to a past state—this is the essence of rollback.
- Non-predicted objects, in the interest list of the client, receive new states from the server.
- Additionally, any non-predicted objects that were modified locally on the client (but haven't received any new state) will also be rolled back to match the authoritative state.

C. Resimulation (Resim)

- `NetworkFixedUpdate` is called on all predicted objects, repeated for the number of times specified by `Sandbox.Resimulations`.
- `Sandbox.IsResimulating` is equal to true.
- This advances the simulation from the rolled-back state to the predicted tick.

At the end of this step:

- `Sandbox.Tick` is again equal to `Sandbox.PredictedTick`.
- The predicted state is now reconciled with the server. If no mispredictions occurred, the predicted objects will appear unchanged compared to the original state at A. However, non-predicted objects may have changed due to updated server data.

This entire process occurs on the client only. While the following step happens for both the client and the server.

D. Forward Tick (Advancing The Simulation Forward)

This marks the actual progression of the game state:

- `NetworkFixedUpdate` is called for all objects (predicted and non-predicted).
- `Sandbox.IsResimulating` is equal to false.

At the end of this step:

- `GameEngineIntoNetcode` of `NetworkBehaviour` is invoked on all objects on the server, and only on predicted objects on the client. This ensures that the network state of components like `NetworkTransform` is updated to reflect the latest state of their corresponding Unity components (e.g., syncing `transform.position` and `transform.rotation` into the internal network variables of `NetworkTransform`).
- `Sandbox.Tick` / `Sandbox.PredictedTick` is incremented by one.

Note: A predicted object refers to any object for which the local player is the Input Source, or any object configured with the Everyone prediction mode.

Prediction In-Depth

Let's start this by doing a recap on what prediction means.

Prediction is when the client tries to guess what the game (the networked state of the game) looks like in the server, starting from the latest received server world snapshot as a starting point. The client uses its local inputs (that have yet to be acknowledged and processed by the server) to simulate forward up to the latency (ping/RTT) between it and the server. Therefore, the client runs ahead of the server time, so that the inputs of the client arrive just when they are needed in the server. Put differently, from the perspective of the client, prediction means predicting the future state of the server world.

However, for many types of games, namely First Person Shooters, it's common to only predict one (and few more) objects, notably the local player character.

When we do this, our local player character object will live in the local (predicted) timeline. While remote objects (including remote players) will live in the remote timeline. What do we mean by these terms?

- Remote Timeline: Remote Timeline refers to the old or delayed timeline we see proxy objects in the client. It's delayed because of ping/latency, incoming data from the server takes a bit of time to arrive. A proxy object is any object that the client is not providing inputs to (not an Input Source of), and the client merely observes the incoming server state snapshots for it. Because the server data is delayed/old (due to latency), what we see for these objects is delayed by `half RTT + interpolation delay` .

- Local/Predicted Timeline: Local/Predicted Timeline refers to the timeline predicted objects in the client live in. The local timeline differs to the remote timeline by being ahead of the remote timeline by `RTT + additional buffering (due to adaptation to non-ideal network conditions)` .

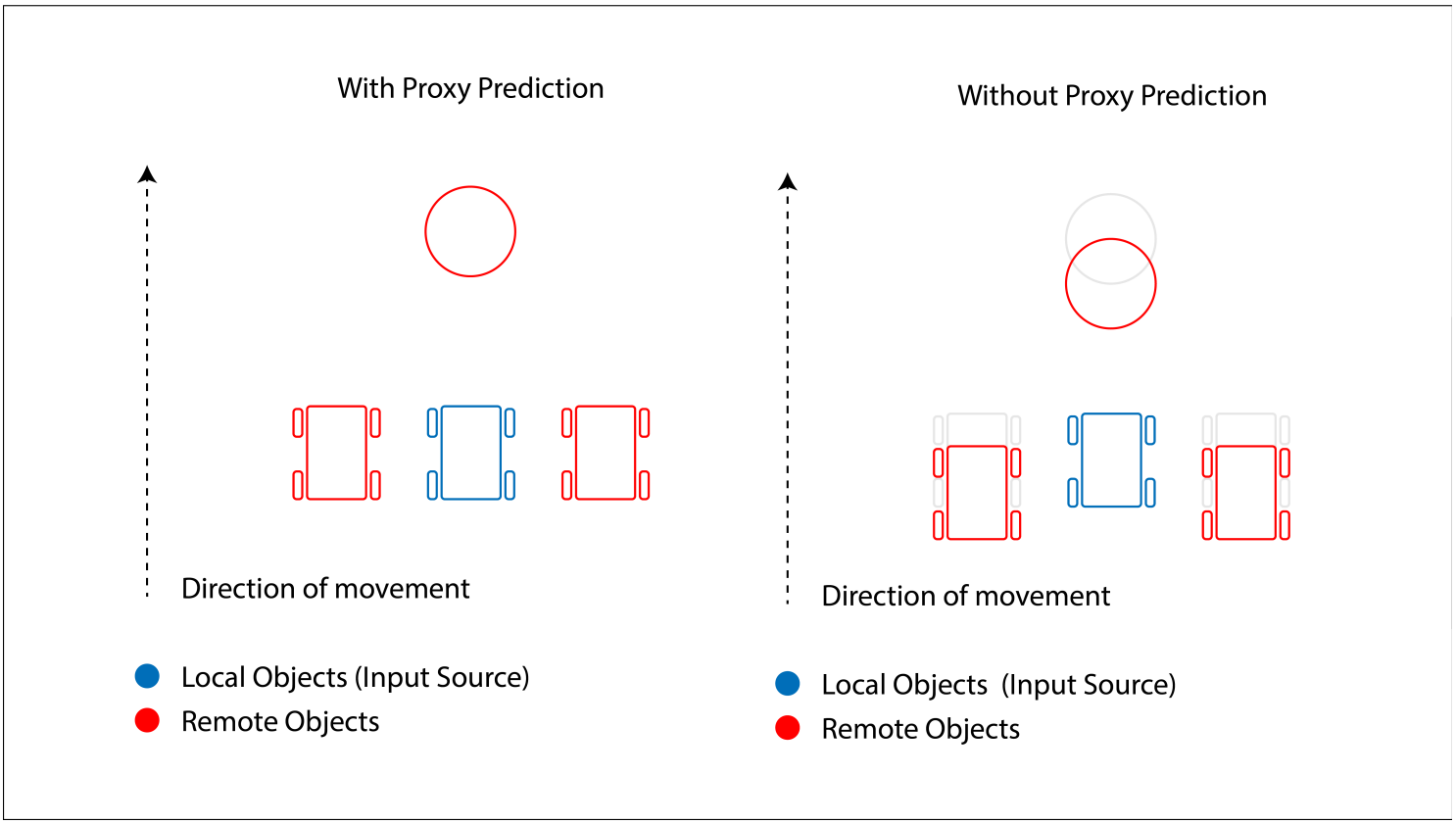
Therefore, there is a discrepancy between the two timelines. Some objects will be in the local timeline, and others will be in the remote timeline. The remote timeline is out-of-sync with the local timeline, which gets worse with ping. Even though this might seem bad, but that's how almost every First Person Shooter works. The local player in an FPS game is in the predicted timeline, while other players (remote players) are in the remote timeline. Why is that though? Why not put all objects in the predicted timeline for an FPS game?

The problem with this is that, usually, the acceleration rate of an FPS character is so fast that the prediction will always be wrong, and it results in a poor gameplay experience. You will see a player come out of a corner and suddenly disappear, due to mispredictions and their subsequent corrections. In addition, server-authoritative bullet hit-detection is tricky on predicted remote objects. Since, due to mispredictions, missed shots will be common. In contrast, lag-compensation, which is a technique that only works with the remote timeline, allows for perfect hit-detection.

An important fact to emphasize is that it's impossible for the client to accurately predict the inputs of other clients. However, players usually don't drastically change their inputs from one moment to the other, which is a good thing.

But, does this mean remote prediction is to be avoided? No. For many types of games, remote prediction provides a better gameplay experience than putting every other object (except the local player) in the remote timeline. Notably anytime you want to do comprehensive interactions (mostly physical interactions) between players, remote prediction will result in a better experience. This is because, without prediction, if you were to collide with a remote object, you will only see the effect of the collision by RTT time. Physics-based games, fighting games, and racing games are all examples of games where predicting remote objects is a better strategy.

By default, Netick only predicts objects that the client is the Input Source of. To understand what it means to predict remote/proxy objects, let's explore an example.



In Rocket Cars, we not only predict the local car, but also the other (remote) cars and the ball. Now, let's see what that means, and also let's see what happens when we don't do that.

Look at the previous image. In this scenario, we assume that each car starts moving at the exact same time. Let's also assume that the cars were moving at the exact same speed for some time, and we are looking at what they look like after that amount of time. In addition, let's say that the ball was also moving in the same direction as the cars. Therefore, all the objects in this scenario are moving, and at the same direction.

In the left-side figure, we see that all cars are aligned with each other, which is what we expect if they started moving at the exact same time and at the exact same speed. Everything looks correct. This is because the clients are not changing their inputs, so the input prediction is correct. But this is usually not the case in practice. However, this shows that prediction will converge to the correct state if the clients are not changing their inputs too much.

Now, let's see what the game looks like if we didn't predict remote/proxy objects. Let's look at the right-side figure above. What we see here is that, now, only our local car is in the predicted position. Other cars are, to us, delayed. They are out-of-sync with the local player car. The gray ghost shapes show where the cars should be, if they were to be predicted. The difference in position here is the amount of positional discrepancy between the local/predicted timeline against the remote timeline, which is proportional to RTT/latency.

So, the conclusion here is that neither approach is perfect. Not predicting remote objects will result in delayed collisions. Predicting them will result in mispredictions. This is the reality of game-networking, there is not a one-size-fit-all solution. You choose the lesser evil.

The lesser evil for this game is to predict remote objects. Therefore, it's a matter of choosing which approach works better for a particular game.

In conclusion, let's see the pros and cons of each approach:

Without Proxy/Remote Prediction

Pros

- Accurate Snapshots: the states of remote objects are correct, since they come directly from the received server snapshots.
- Lag Compensation: you can have perfect server-authoritative hit-detection using lag compensation for clients since what the clients see is what actually happened, without mispredictions.
- Low CPU Overhead: since you only simulate the local player, CPU performance will be better.

Cons

- Weak Player-to-Player Interactions: usually the best approach is to disable collisions between players.
- Multiple Timelines: the local player is out-of-sync with remote players, due to being in the Local Timeline, while they are in the Remote Timeline.

With Proxy/Remote Prediction

Pros

- Good Player-to-Player Interactions: you can have smooth and responsive interactions between players, such as collisions.
- Single Timeline: all objects live in the same timeline, which is the local/predicted timeline. No desync between objects.
- Simpler Code: by being able to simulate other objects and have them all in the same timeline, the coding experience will be closer to single-player development.

Cons

- Mispredictions: the rendered states of predicted remote objects are not necessarily states that actually happened in the server, due to mispredictions. One player can report seeing different things compared to other players, creating contradictory perspectives on what happened. Mispredictions get worse with higher pings, so clients with very high pings (+300) might have almost an unplayable experience.
- No Lag Compensation: you can't perform lag compensation on predicted objects. However, because all objects are in the same timeline, there is no need for lag compensation. But, due to mispredictions, the client hits will often miss.
- High CPU Overhead: predicting more objects will use more CPU time, and the cost of that increases with ping and tickrate.

Predicting Remote/Proxy Objects

Now, let's see how we can actually predict remote objects, in practice. Using Netick, this is quite simple. Simply change the `Prediction Mode` of an object to be equal to `Everyone` in the inspector. This will cause the `NetworkFixedUpdate` of this object to execute multiple times due to resimulation.

However, this is not all. To be able to predict the input of other players, we need to sync their inputs. The following code snippet is taken from Rocket Cars.

```
[Networked] public GameInput   LastInput  { get; set; } // We sync the last input for the player. So we can use it to predict remote players cars.
public override void NetworkFixedUpdate()
{
  if (FetchInput(out GameInput input))
    LastInput          = input;

  SimulateVehicle(LastInput);
}
```

That's all there is to it. `FetchInput` only returns true on the Input Source itself, and the server. So, by simply defining a network property to store the input in, we are able to sync the input to everyone, including observing (proxies) players.

Notice that we don't actually try to predict the input, we simply use the last input for prediction. Because predicting that the client pressed something it never did is a lot worse than simply assuming the client is still pressing the same buttons.

Rocket Cars serves as an excellent example of how Proxy/Remote Prediction works.

Prediction Error Correction Smoothing

By default, correcting mispredictions is instantaneous. This will cause the predicted remote objects to snap somewhere else when a player changes their movement direction suddenly. And as we said, the magnitude of mispredictions is proportional to latency. Therefore, for a smooth visual experience, we must smooth out the prediction correction. Netick implements a smooth correcter in `NetworkTransfrom` / `NetworkRigidbody` . By enabling it, it will smooth out the corrections over multiple frames. There are a few settings for it which will need to be fine-tuned to find what is best for your object.

Input Delay

A powerful technique to reduce mispredictions on remote objects is to to delay the inputs of everyone by a specific amount/ticks. This will make the game almost perfectly synced without mispredictions for players with pings roughly below the input delay.

Example:

```
public float                              InputDelay         = 100; // in milliseconds
public const int                          InputQueueCapacity = 6;
[Networked(size: InputQueueCapacity)]
public readonly NetworkQueue<MyInput>     InputQueue        = new(InputQueueCapacity);
[Networked]
public MyInput                            LastInput         { get; private set; }

public override void NetworkFixedUpdate()
{
  if (FetchInput(out MyInput i))
  {
```

```
      if (InputQueue.Count == InputQueueCapacity)
         InputQueue.Dequeue();
       InputQueue.Enqueue(i);
     }

    int inputDelayInTicks = (int)Mathf.Round((InputDelay / 1000f) / Sandbox.FixedDeltaTime);

    if (InputQueue.Count > 0 && (Sandbox.Tick - InputQueue.Peek().Tick >= inputDelayInTicks))
       LastInput = InputQueue.Dequeue();

    // logic
    Move(LastInput);
  }
```

Note that we've added a field called `Tick` to the input struct, which we assign it the value of `Sandbox.Tick` when setting the input struct fields.

Remote Procedure Calls (RPCs)

Remote Procedure Calls (RPCs) are methods defined on `NetworkBehavior` scripts that can be invoked remotely across the network. They are typically used to synchronize discrete events or transmit small amounts of data between clients and the server.

A common use case for RPCs is initializing gameplay logic or sending configuration data at the start of a session. Such tasks should use **reliable RPCs** to ensure delivery.

> **NOTE**
>
> While other solutions are heavily dependent on RPCs, Netick is designed to make usage of RPCs very minimal (less than 3 RPCs in the entire game). RPCs teach bad practices and produce spaghetti code. Read the article on RPCs vs Properties for more.

> **WARNING**
>
> RPCs are not suitable for sending large amounts of data (e.g., over 500 bytes) or transferring files. For those use cases, refer to this article.

Basic Example

Here's a simple example of an RPC:

```
[Rpc(source: RpcPeers.Everyone, target: RpcPeers.InputSource, isReliable: true, localInvoke: false)]
private void MyRpc(int arg1)
{
    // Code to be executed remotely
}
```

To declare a method as an RPC, decorate it with the [`Rpc`] attribute.

Static RPCs

Static RPCs are RPCs not tied to a specific instance of a `NetworkBehavior` . The first parameter of a static RPC must be of type `NetickEngine` , which allows access to the current `NetworkSandbox` .

```
[Rpc]
public static void MyStaticRpc(NetickEngine engine, int someRpcPara)
{
    var sandbox = engine.UserObject as NetworkSandbox;
}

// Invoking the RPC:
MyStaticRpc(Sandbox.Engine, 56);
```

> **WARNING**
>
> RPCs are not executed on resimulated ticks.

> **WARNING**
>
> Additionally, all RPCs are **unreliable by default** unless explicitly marked otherwise.

`[Rpc]` Method Requirements

RPC methods must adhere to the following constraints:

- Must have a return type of `void`.
- Reference types are not allowed as parameters.
- Class-based network collections are not allowed. Use `NetworkArrayStruct` for array parameters.
- `string` parameters are not allowed. Use one of the `NetworkString` variants instead.

`[Rpc]` Attribute Parameters

The `[Rpc]` attribute accepts the following options:

Parameter	Description
`source`	Specifies which peer(s) the RPC originates from
`target`	Specifies which peer(s) should execute the RPC
`isReliable`	If `true`, the RPC will be sent reliably
`localInvoke`	If `true`, the RPC will also be invoked locally

Peer Options

The `source` and `target` can be any of the following:

- `Owner` — The server.
- `Input Source` — The player providing input for the object.
- `Proxies` — All peers except the Owner and Input Source.
- `Everyone` — All connected peers, including the server.

Targeted RPCs

To send an RPC to a specific peer (e.g., a single player), include a parameter of type `NetworkPlayerId` decorated with the `[RpcTarget]` attribute. For instance:

```
[Rpc(source: RpcPeers.Everyone, target: RpcPeers.Everyone, isReliable: true, localInvoke: false)]
private void MyRpc([RpcTarget] NetworkPlayerId target, int arg1)
{
    // ...
}

// Invocation example:
MyRpc(Sandbox.Players[2], someValue);
```

In the case of static RPCs, the `NetworkPlayerId` must be the **second** parameter, following the `NetickEngine` parameter.

Identifying the RPC Source

To determine which player originally called an RPC, add a final parameter of type `RpcContext`. This allows you to retrieve the `Source` of the RPC at runtime:

```
[Rpc(source: RpcPeers.Everyone, target: RpcPeers.Everyone, isReliable: true, localInvoke: false)]
private void MyRpc(int arg1, RpcContext ctx = default)
{
    var rpcSource = ctx.Source; // Identifies the caller/sender
}
```

RPCs vs Properties

Remote Procedure Calls (RPCs) and Network Properties (Networked State) serve distinct purposes in a multiplayer game.

RPCs are an option for non-critical, often cosmetic events that occur at specific moments. In contrast, Network Properties are used to synchronize critical gameplay state, especially data that must persist and remain synced for all clients, including late joiners.

When to Use Network Properties

Network Properties are designed for values that:

- Change frequently.
- Must remain synchronized for all players.
- Need to be known by clients who join mid-game.

Example: a player's health should be a Network Property, as its current value is critical for gameplay and must remain synced across all clients at all times.

When to Use RPCs

RPCs are event-based, meaning they only execute at a specific point in time. Clients that join after an RPC is called will not receive that RPC. For this reason, they are best used for:

- Visual or cosmetic events.
- One-off setup instructions (e.g. sent from client to server).
- Chat messages or UI updates.

Example: a visual damage effect or sound effect that doesn't impact game logic can be triggered via an RPC.

However, you can, and should, avoid using RPCs even for such events, and that's by using a change callback using `[OnChanged]` attribute. Ideally, RPCs should only be used for setting up things in the server by the client (client->server RPC), or sending chat messages.

Avoiding RPCs

In general, you should minimize your usage of RPCs — especially server->client RPCs, which are bandwidth-heavy and harder to manage.

- RPCs from the client can be a security concern. Since you can't control how the client calls them. And they are not tick-aligned, which can be a problem if an RPC is intended to be used for tick-accurate gameplay logic. You can use network inputs to handle most of your client->server actions.
- RPCs from the server to every connected client are expensive. You can always find a way to mimic an RPC using a network property and an `[OnChanged]` event.

Using [OnChanged] Callbacks for Events

OnChanged callbacks are very powerful. Their use-cases are endless. For a couple examples:

Jump Sound

Use a `JumpCounter` Network Property. Increment it each time the player jumps. In the `[OnChanged]` callback, play the jump sound.

Death Effect

Monitor a `Health` Network Property. If it drops to zero (and was higher before), trigger a death animation or sound in the callback.

Circular Buffers

One of the easiest ways to avoid many types of RPCs is to use Circular Buffers. A circular buffer is a `NetworkArray` that you update in a circular/ring fashion. When you reach the end of the array, you start over at the beginning of the array - this is accomplished by using the modulo operator when indexing the array, using an incrementing value. Using this, you can synchronize many short-lived things such as projectiles, hit indicators, and more.

Example:

This example showcases how we can use circular buffers to synchronize hit indicators, usually seen in FPS games. Hit indicators are rotating arrows around the middle of the screen that point to the locations of where you were last being damaged from.

```
public struct DamageIndicatorData
{
```

```
    public NetworkObjectRef AttackerPlayer;
    public int              Incrementor;
}

private int                                     _hitIncrementor;

[Networked(size: 5)]
public readonly NetworkArray<DamageIndicatorData> DamageIndicatorsSources = new NetworkArray<DamageIndicatorData>(5);

[OnChanged(nameof(DamageIndicatorsSources))]
private void OnDamageIndicatorsSourcesChanged(OnChangedData info)
{
    // invoking an event when the array changes. We subscribe to this event in a UI script to show the damage indicators and fade them overtime.
    OnDamageIndicatorsSourcesChangedEvent?.Invoke(info.GetArrayChangedElementIndex());
}

// This is an example of modifying the circular buffer.
public void ApplyDamage(NetworkObjectRef AttackerPlayer, int damageAmount)
{
    // ..............
    // other code
    if (IsClient)
        return;

    DamageIndicatorsSources[_hitIncrementor % DamageIndicatorsSources.Length] = new DamageIndicatorData()
    {
        AttackerPlayer = AttackerPlayer,
        Incrementor    = _hitIncrementor // we included the incrementor variable as part of the struct to force the OnChanged callback to fire again
if the same attacker player was assigned.
    };

    _hitIncrementor++;
}
```

Network Object Instantiation and Destruction

Instantiate

To Instantiate a network prefab:

```
Sandbox.NetworkInstantiate(prefab, position, Quaternion.identity);
```

This must be called only in the server.

Destroy

To destroy a network object:

```
Sandbox.Destroy(obj)
```

This will destroy `obj` and all of its nested Network Objects.

This must be called only in the server.

> **WARNING**
>
> Make sure to never use Unity's instantiate/destroy methods to create/destroy a network object, only Netick's methods.

> **WARNING**
>
> Make sure that all your prefabs are registered by Netick in Netick Settings panel. And also make sure the prefab list is identical in both the client and the server (if you are running two Unity editors), otherwise, weird stuff will occur.

Network Prefab Pool

Object pooling is a very effective technique to avoid run-time allocations (and thus, improve performance), by creating a pool of objects of the same type, at the start of the game. So that when you want to instantiate a certain prefab, you will not create a new object in memory. But rather, all instances of that prefab are already created, and you simply grab one out of the pool and initialize it. And when you want to destroy an instance, instead of removing it from memory (which causes GC), you put it back on the pool – recycling it.

Pooling is extremely useful and effective if you have a prefab in your game that you instantiate and destroy repeatedly. For instance, the bomb in Bomberman.

Netick has a built-in pooling system that you can use.

By default, all prefabs are not pooled. To enable pooling for a certain prefab, you must call `InitializePool` **(should be called at the start of Netick in `NetworkEventsListener` so that all prefab instances created are pooled)** on that prefab and pass it the initial amount to create:

```
public override void OnStartup(NetworkSandbox sandbox)
{
     var bombPrefab = sandbox.GetPrefab("Bomb");
     sandbox.InitializePool(bombPrefab, 5);
}
```

> **NOTE**
>
> Check out the Bomberman sample if you are confused. It demonstrates pooling of the bomb prefab.

And if this amount happens to be exceeded, Netick will simply create more objects in the pool automatically.

And you don't need to use special instantiate or destroy methods to deal with pooled prefabs, it all works through the same `Sandbox.NetworkInstantiate` and `Sandbox.Destroy` methods.

Although you still need to reset your objects. However, Netick automatically resets all network properties to their declaration values.

Resetting Prefab Instances

You can simply use `NetworkStart` to reset your prefab instances.

Managing Netick

Starting and Shutting Down Netick

When you start Netick, you need to specify the mode you want to start it in. Like this:

Single Peer

Start as Client

```
var sandbox = Netick.Unity.Network.StartAsClient(Transport, Port);
```

Start as Host (a server with a local player)

```
var sandbox = Netick.Unity.Network.StartAsHost(Transport, Port);
```

Start as Server

```
var sandbox = Netick.Unity.Network.StartAsServer(Transport, Port);
```

Start as Single-Player (disables low level networking)

```
var sandbox = Netick.Unity.Network.StartAsSinglePlayer();
```

Multiple Peers (Sandboxing)

Learn More About Sandboxing

You can start both a client and a server together:

```
var sandboxes       = Netick.Unity.Network.Launch(StartMode.MultiplePeers, new LaunchData()
{
  Port              = Port,
  TransportProvider = Transport,
  NumberOfServers   = 1,
  NumberOfClients   = 1
});
```

Starting multiple servers:

```
int   portOffset    = 4567;
int[] ports         = new int[20];
for (int i = 0; i < 20; i++)
  ports[i]          = portOffset + i;

// starts multiple servers (20 servers)
var sandboxes       = Netick.Unity.Network.Launch(StartMode.MultiplePeers, new LaunchData()
{
  Ports             = ports,
  TransportProvider = Transport,
  NumberOfServers   = 20
});
```

To shut down Netick completely:

```
Netick.Unity.Network.Shutdown();
```

Connecting to the Server

To connect the client to the server:

```
sandbox.Connect(serverIPAddress);
```

Disconnecting From the Server

To disconnect the client:

```
sandbox.Disconnect();
```

You are advised to have a game starting scene used for server finding/matchmaking.

Scene Management

Scene Loading

To switch from the current scene to another:

```
Sandbox.SwitchScene("sceneName");
```

Additive Scenes

Loading an additive scene:

```
Sandbox.LoadSceneAsync("sceneName", LoadSceneMode.Additive);
```

Unloading an additive scene:

```
Sandbox.UnloadSceneAsync("sceneName");
```

> **WARNING**
>
> All scene load/unload methods must only be called in the server.

> **WARNING**
>
> `UnloadSceneAsync` must only be called for unloading additively loaded scenes. To unload the main scene, use `SwitchScene` or `LoadSceneAsync` with `LoadSceneMode.Single` .

> **NOTE**
>
> To find the build index of a scene, open the `Build Settings` window where you will see a list of all added scenes. If the desired scene is not present, open that scene and add it to the list.

Scene Events

When you call `Sandbox.LoadSceneAsync` in the server, for instance, `OnSceneOperationBegan` event will be invoked in both the client and the server. You can use the `NetworkSceneOperation` parameter to know information about the scene load/unload operation like the current progress.

`OnSceneOperationDone` will be invoked when that scene operation finishes. `NetworkSceneOperation` struct includes a `Scene` getter you can use to access the `UnityEngine.SceneManagement.Scene` struct.

Using `NetworkEventsListener`

On a script inheriting from `NetworkEventsListener` , you can run code for when a certain scene operation has began and when it has finished.

```
public override void OnSceneOperationBegan(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)
 {
  // invoked in both the client and the server when when you call Sandbox.LoadSceneAsync, Sandbox.UnloadSceneAsync, or Sandbox.SwitchScene.
  // sceneOperation lets you know information about the scene operation like the current progress of the scene load/unload.
 }

 public override void OnSceneOperationDone(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)
 {
  // invoked in both the client and the server when a scene operation caused by calling Sandbox.LoadSceneAsync, Sandbox.UnloadSceneAsync, or
```

```
Sandbox.SwitchScene finishes.
 }
```

Using `Sandbox.Events`

Or you can manually subscribe/unsubscribe on a `NetworkBehaviour` .

```
public override void NetworkAwake()
{
  Sandbox.Events.OnSceneOperationBegan += OnSceneOperationBegan;
  Sandbox.Events.OnSceneOperationDone  += OnSceneOperationDone;
}

public override void NetworkDestroy()
{
  Sandbox.Events.OnSceneOperationBegan -= OnSceneOperationBegan;
  Sandbox.Events.OnSceneOperationDone  -= OnSceneOperationDone;
}

private void OnSceneOperationBegan(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)
{
}
private void OnSceneOperationDone(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)
{
}
```

Listening to Network Events

Netick has several useful callbacks you can use:

Callbacks	Description	Invoke target
OnStartup(NetworkSandbox sandbox)	Invoked when Netick has been started.	Client/Server
OnShutdown(NetworkSandbox sandbox)	Invoked when Netick has been shut down.	Client/Server
OnInput(NetworkSandbox sandbox)	Invoked to read inputs.	Client/Server
OnConnectRequest(NetworkSandbox sandbox, NetworkConnectionRequest request)	Invoked when a client tries to connect. Use request to decide whether or not to allow this client to connect.	Server
OnConnectFailed(NetworkSandbox sandbox, ConnectionFailedReason reason)	Invoked when the connection to the server was refused, or simply failed.	Client
OnConnectedToServer(NetworkSandbox sandbox, NetworkConnection server)	Invoked when the connection to the server has succeeded.	Client
OnDisconnectedFromServer(NetworkSandbox sandbox, NetworkConnection server, TransportDisconnectReason transportDisconnectReason)	Invoked when connection to the server ended, or when a network error caused the disconnection.	Client
OnPlayerJoined(NetworkSandbox sandbox, NetworkPlayerId player)	Invoked when a specific player has joined the game.	Client/Server
OnPlayerLeft(NetworkSandbox sandbox, NetworkPlayerId player)	Invoked when a specific player has left the game.	Client/Server
OnSceneOperationBegan(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)	Invoked when a scene operation has began.	Client/Server
OnSceneOperationDone(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)	Invoked when a scene operation has finished.	Client/Server
OnObjectCreated(NetworkObject obj)	Invoked when a network object has been created/initialized.	Client/Server
OnObjectDestroyed(NetworkObject obj)	Invoked when a network object has been destroyed/recycled.	Client/Server

You can override these methods on a class inheriting from `NetworkEventsListener`, and add it to an object in the scene, and Netick will find it automatically. Or, you can add it to a network prefab that you instantiate, and Netick will also find it can call methods on it.

You could also add the component `NetworkEvents` to an object, which does the same, but the difference is that you can plug your events right into it.

And finally, you can use `Sandbox.Events` to directly subscribe/unsubscribe network events from any script, including network behaviours:

```
public override void NetworkStart()
{
  Sandbox.Events.OnPlayerJoined += OnPlayerJoined ;
}

public override void NetworkDestroy()
{
  Sandbox.Events.OnPlayerJoined -= OnPlayerJoined ;
}

private void OnPlayerJoined (....)
{
}
```

This is the cleanest way of using network events.

Parenting

The parent of a network object can only be changed if you are the Input Source of the object, or if you are the owner (server).

To change the parent of an object: `Object.SetParent(newParent);`

> **CAUTION**
>
> The original hierarchy of network prefab instances shouldn't be changed at run-time. In other words, you shouldn't unparent the original children of a prefab. Although, you can parent objects to them, just not unparenting them (prefab children) from their original parent.

Script Execution Order

The network methods on your Network Behavior classes are called from inside Netick, which means standard Unity `MonoBehaviour` script order control does not work here. To specify the order of execution, use the attributes:

```
[ExecuteAfter(typeof(SomeOtherScript))]  /* to specify that this script executes after SomeOtherScript) */
[ExecuteBefore(typeof(SomeOtherScript))] /* to specify that this script executes before SomeOtherScript */
[ExecutionOrder(65)] /* to specify explicitly the order number
```

Example:

```
[ExecuteAfter(typeof(SomeOtherScript))]
public class BomberController : NetworkBehaviour
{
    // ...
}
```

The methods that are affected by these attributes are

- `NetworkStart`
- `NetworkUpdate`
- `NetworkFixedUpdate`
- `NetworkRender`

The following methods are also affected, but only in respect to the object itself:

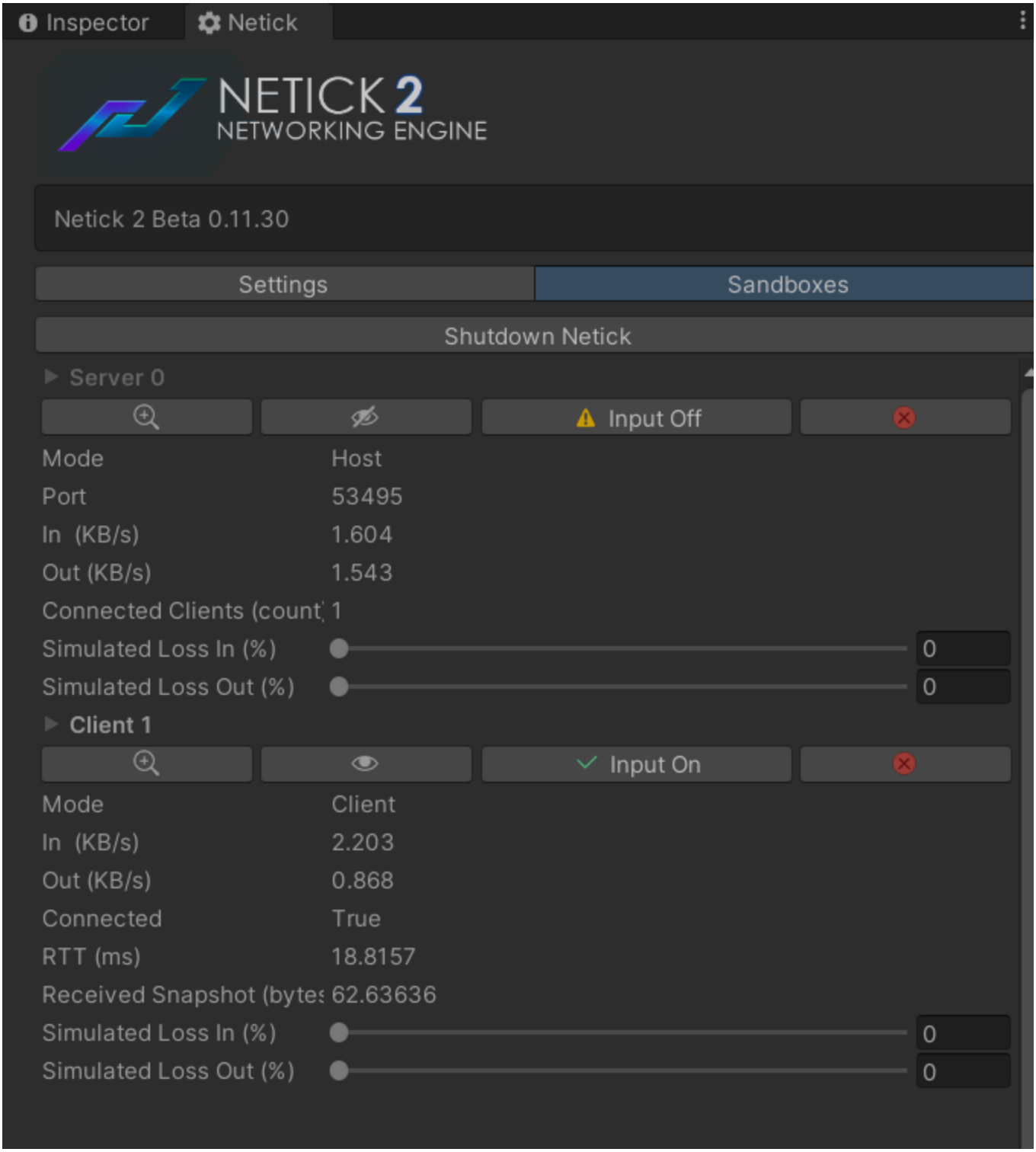
- `NetworkDestroy`
- `NetcodeIntoGameEngine`
- `GameEngineIntoNetcode`

Sandboxing

Sandboxing (also known as multi-peer) allows you to start multiple Netick instances in a single Unity process for various purposes:

- Starting multiple clients and a server.
- Starting multiple servers in a single Unity process.

Sandboxes panel can be accessed by going to `Netick > Sandboxes`



Starting Netick as Multiple Peers

Starting a Client and a Server

```
var sandboxes       = Netick.Unity.Network.Launch(StartMode.MultiplePeers, new LaunchData()
{
  Port              = Port,
  TransportProvider = Transport,
  NumberOfServers   = 1,
  NumberOfClients   = 1
});
```

Starting Multiple Servers

```
int   portOffset    = 4561;
int[] ports         = new int[20];
for (int i = 0; i < 20; i++)
  ports[i]          = portOffset + i;

var sandboxes       = Netick.Unity.Network.Launch(StartMode.MultiplePeers, new LaunchData()
{
  Ports             = ports,
  TransportProvider = Transport,
  NumberOfServers   = 20
});
```

Making Your Project Sandbox-Safe

Running any project in multiple-peers mode does not always work, because of how some projects are set up. We call scripts or projects that can work with multiple sandboxes without issues as sandbox-safe.

Notes on how to make your project sandbox-safe:

- Try to completely avoid using `static` variables. This is because you will run more than one instance of the game and the `static` variable will conflict between the different Netick instances, since each Netick instance must have its own copy of that variable. If you were using `static` for singleton types, you can do the same by using a Sandbox Prefab. Attach all your singleton-like components to your sandbox prefab and you can access them from any network behaviour using `var mySingleton = Sandbox.GetComponent<TypeOfScript>();` . This way each Netick instance will have its own singleton-like scripts.

- Use `Sandbox.Physics.Raycast` instead of `Physics.Raycast` when wanting to perform a raycast, same thing goes for other physics queries too. Using `Sandbox.Physics.Raycast` lets you query against the physics scene associated with this sandbox. Since `Physics.Raycast` simply uses the main Unity physics scene that is created when starting Unity, which is not sandbox-safe since it would raycast against objects in the first sandbox only (the sandbox that has the main Unity physics scene associated with it).

- When you want to disable a render/audio component on a GameObject, use `SetEnabled` instead of `enabled` . This method respects the running sandboxes so when a hidden sandbox enables a mesh renderer, for instance, it will not be visible because that sandbox is hidden.

- When you want to instantiate a non-networked GameObject, use `Sandbox.Instantiate` instead of `GameObject.Instantiate` . `Sandbox.Instantiate` respects the running sandboxes, so when a hidden sandbox instantiates a new GameObject, it will not be visible because that sandbox is hidden.

- Use `Sandbox.Log` , `Sandbox.LogWarning` , and `Sandbox.LogError` instead of Unity equivalents. These will include the name of the sandbox at the start of the log message.

- Use `Sandbox.FindObjectOfType` , `Sandbox.FindObjectsOfType` , `Sandbox.FindGameObjectWithTag` , and `Sandbox.FindGameObjectsWithTag` instead of Unity equivalents. The Netick methods respect the running sandboxes, and only try to find objects in the Sandbox's scene.

Some useful properties and events for working with multiple sandboxes:

```
// is the sandbox visible
Sandbox.IsVisible

// is input enabled for the sandbox
Sandbox.InputEnabled

// invoked when a sandbox visibility state (shown/hidden) changes.
Sandbox.Events.OnVisibilityChanged
```

Timers

Netick provides several ways to work with time in your networked game.

Converting Between Ticks and Time

Sometimes you need to convert between ticks and time (in seconds). Netick provides methods to do this conversion:

Sandbox.TickToTime()

The `TickToTime()` method converts ticks to time in seconds. This is useful when you need to check if a certain amount of time has passed since an event occurred. For example, if you want to explode a bomb after a delay:

```
public override void NetworkFixedUpdate()
{
    if (Sandbox.TickToTime(Sandbox.Tick - Object.SpawnTick) >= ExplosionDelay)
        Explode();
}
```

In this example, `Sandbox.Tick` is the current tick, `Object.SpawnTick` is the tick when the object was spawned, and `ExplosionDelay` is the delay in seconds before the explosion. The difference between the current tick and spawn tick determines the number of ticks that have passed since the object was spawned, which is then converted to seconds using `TickToTime()` .

Sandbox.TimeToTick()

The `TimeToTick()` method does the opposite conversion, from time in seconds to ticks:

```
// Convert time in seconds to ticks
Tick tickValue = Sandbox.TimeToTick(timeInSeconds);
```

Using NetworkTimer

Netick provides a `NetworkTimer` class that makes it easier to work with time-based events in your networked game. The `NetworkTimer` class offers methods to check if a timer is running, if it has stopped, and how much time remains.

Starting a Timer

You can start a timer using the `Sandbox.StartTimer()` method:

```
// Start a 5-seconds timer
NetworkTimer myNetworkTimer = Sandbox.StartTimer(5.0f);
```

> **NOTE**
>
> By default, the system uses predicted timing. You can use `usePredictedTiming: false` to opt out of this feature: `Sandbox.StartTimer(duration, usePredictedTiming: false)` .

Checking Timer Status

You can check the status of a timer using the following methods:

```
// Check if the timer is still running
if (myNetworkTimer.IsRunning(Sandbox))
{
    // Timer is still running
}

// Check if the timer has stopped
if (myNetworkTimer.IsStopped(Sandbox))
{
```

```
    // Timer has stopped/finished
}
```

Getting Remaining and Elapsed Time

You can get the remaining time and elapsed time of a timer:

```
// Get the remaining time in seconds
float remainingTime = myNetworkTimer.GetRemainingTime(Sandbox);

// Get the elapsed time in seconds
float elapsedTime = myNetworkTimer.GetElapsedTime(Sandbox);
```

Physics Prediction (Unity)

Predicting physics means resimulating multiple physics steps in one tick. This can be very expensive and so by default physics prediction is turned off. To enable it, go to `Netick -> Settings` and enable `Physics Prediction`.

To make a `Rigidbody` / `Rigidbody2D` predictable, add `NetworkRigidbody` / `NetworkRigidbody2D` to its GameObject.

To enable/disable Physics Prediction in the client at runtime, use `Sandbox.PhysicsPrediction`.

> **NOTE**
>
> When Netick starts and the `Physics Type` option in Netick Settings/Config is set to either `Physics2D` or `Physics3D` (as opposed to `None`), Netick will take control of the physics stepping and automatically set `UnityEngine.Physics.simulationMode` to `Script`.

Cost of Predicting PhysX (Rigidbody3D)

It's very expensive to predict 3D physics as PhysX and its integration with Unity perform very badly when calling `PhysicsScene.Simulate` multiple times in one frame, even with small numbers of rigidbodies.

The cost of predicting physics increases with two factors:

- Ping
- Tickrate

As ping increases, you would need to simulate more ticks in one frame for rollback and resimulation. As tickrate increases, the time period between each tick becomes smaller, therefore you will need to simulate more ticks for smaller values of latency.

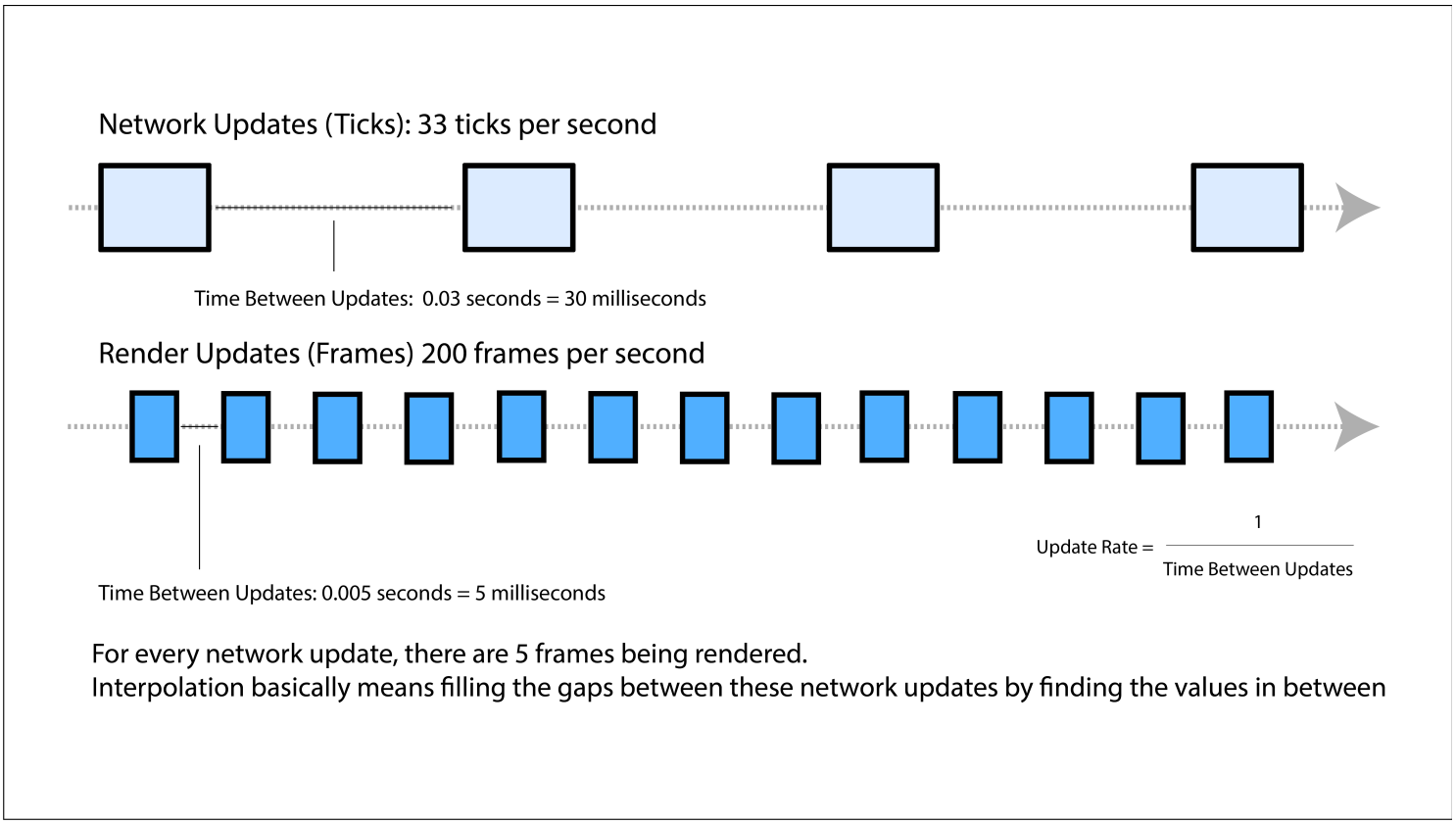
It's very much not recommended to enable 3D physics prediction, as it's almost impractical on some machines on relatively high tickrates (+33). It can take more than 10ms on some machines on 100ms ping just to resimulate a bunch of physics ticks.

If you don't want to predict physics, you have two other alternatives:

- Not predicting physics at all, and make it server-authoritative completely.
- Make your physics objects client-authoritative by sending the resultant states as RPCs or network inputs.

Interpolation

Netick runs at a fixed-time step, equal to the inverse of the tick rate used for the simulation, which you can specify in Netick Settings. Because of that, the motion of network objects will appear unsmooth and jittery. The reason for this is that, usually, your update rate (render rate) is way higher than your fixed network tick rate. The solution to this problem is called interpolation, which means filling in the gaps between these fixed-time steps/ticks:



So, for example, at tick 6, the value of a network property is 2.0. And at tick 7, it becomes 3.0.
Since there are 5 frames between two ticks, the values at each frame would be:

- Frame 1: 2.0 — Beginning of tick 6
- Frame 2: 2.25
- Frame 3: 2.5
- Frame 4: 2.75
- Frame 5: 3 — End of tick 6, beginning of tick 7

Interpolation of Network Transform

For moving objects, this is important to deal with. Every NetworkTransform has a slot for a Render transform, which is basically the smoothed/interpolated mesh of the object, while the parent would be the simulated/non-interpolated object.

So, you must break your moving objects into a parent (which has the NetworkTransform), and a child which is the interpolated object, and has the mesh/s. Then you specify that child in the NetworkTransform RenderTransform property in the inspector. Check the samples if you are confused.

Interpolation Source

The source of interpolation data can be of two options:

- **Local/Predicted Snapshot**: This is called `Local Interpolation`. It means using the local predicted snapshots for interpolation. This is what you usually use for your local player as you want to use the local predicted snapshots, and it's chosen by default for objects the client is the Input Source of when `Interpolation Source` is set to `Auto`.

- **Remote Snapshot**: This is called `Remote Interpolation`. It means using the received snapshots from the server for interpolation. This data is delayed, and that's why it's called remote. This is what you usually use for other objects (not your own player) as you want to use the smoothed and buffered server snapshots, and it's chosen by default for objects the client is not the Input Source of when `Interpolation Source` is set to `Auto`.

> **NOTE**
>
> When you want the server only to move your local player object, you must switch `Interpolation Source` to `Remote Snapshot`, to keep smooth rendering of the object as it's being controlled remotely and the prediction buffers will contain jittery data as the object is not being moved locally in the client.

Interpolation of Network Properties

To interpolate a property, add the [Smooth] attribute to its declaration:

```
[Networked][Smooth]
public Vector3 Movement {get; set;}
```

Automatic Interpolation

To access the interpolated value, by referencing the property in NetworkRender, you automatically get interpolated values:

```
public override NetworkRender()
{
    var interpolatedValue = Movement;
}
```

Automatic Interpolation is implemented by Netick on these types:

- `Float`
- `Double`
- `Vector2`
- `Vector3`
- `Quaternion`
- `Color`
- `Int`
- `NetworkBool`

> **WARNING**
>
> Currently this is only supported in Unity. Use Manual Interpolation in other engines.

Manual Interpolation

To manually interpolate a network property or network array, you can do that using the Interpolator struct. You also have to pass false to `[Smooth]` to inform Netick we want to manually interpolate the property.

Interpolating Properties

```
[Networked][Smooth(false)]
public MyType MyType {get; set;}

public override NetworkRender()
{
    var    interpolator      = FindInterpolator(nameof(MyType));
    bool   didGetData        = interpolator.GetInterpolationData<MyType>(InterpolationSource.Auto, out var from, out var to, out float alpha);

    MyType interpolatedValue = default;

    // if we were able to get interpolation data
    if (didGetData)
        interpolatedValue    = LerpMyType(from,to,alpha);
    else // if not we just use the non-interpolated value
        interpolatedValue    = MyType;
```

```
}

private MyType LerpMyType(MyType from, MyType to, float alpha)
{
    // write the interpolation code here
}
```

Interpolating Arrays

```
[Networked (size: 10)][Smooth(false)]
public readonly NetworkArray<MyType> MyTypeArray = new NetworkArray<MyType>(10);

public override NetworkRender()
{
    var          interpolator      = FindInterpolator(nameof(MyTypeArray));
    int          index             = 5;
    bool         didGetData        = interpolator.GetInterpolationData<MyType>(InterpolationSource.Auto, index, out var from, out var to, out
float alpha);

    MyType       interpolatedValue = default;

    // if we were able to get interpolation data
    if (didGetData)
        interpolatedValue = LerpMyType(from, to, alpha);
    else // if not we just use the non-interpolated value
        interpolatedValue = MyTypeArray[index];
}

private MyType LerpMyType(MyType from, MyType to, float alpha)
{
    // write the interpolation code here
}
```

> **NOTE**
>
> You should cache the result to `FindInterpolator` on NetworkStart, instead of calling it repeatedly every frame (NetworkRender is called every frame), since it might be a bit slow.

Optimizing Large Numbers of Objects

Usually networked objects in a video game are simulated in one of two ways:

Internal Simulation

Internal Simulation refers to objects which are simulated and changed from within. Objects such as these include the character of the player. They also include objects such as vehicles, all sorts of physics objects, and game management objects. Any object that is self-controlled and needs to run each tick is within this category.

External Simulation

External Simulation refers to objects that never simulate or change themselves, instead they are controlled from the outside. There are many examples of such objects: doors, pickups, trees, buildings, etc. All these objects are altered from an external object that belongs to the first category (Internal Simulation). These objects don't need to run themselves or have `NetworkFixedUpdate`, `NetworkUpdate`, or `NetworkRender` run on them each tick/frame. Therefore we should simply exclude them from the network loop, meaning all of their network loop methods (`NetworkFixedUpdate`, `NetworkUpdate`, `NetworkRender`, etc) will not be invoked. This will save a lot of performance, potentially making some types of games possible when they wouldn't be.

Netick makes this very simple. To make an object externally simulated, disable `Add to Network Loop` on its NetworkObject. It will no longer be simulated each tick, but it will still be synced, and its `NetworkStart` and `NetworkDestroy` methods invoked. The object will have no CPU cost almost at all, you can have 1000 objects and shouldn't see a difference in CPU performance, excluding the rendering cost which is irrelevant.

It's important to note that you should stop using the built-in components such as `NetworkTransform` on these objects since these components assume the object is internally simulated. Instead, you would write simple scripts that react to networked properties using `[OnChanged]` callbacks to handle them.

Example

This is an example of a script that is used to synchronize the position of an object that will only be simulated externally. By a script on the player object, for instance.

```
using UnityEngine;
using Netick;
using Netick.Unity;

public class CustomPosSyncer : NetworkBehaviour
{
  [Networked]
  public Vector3 Position { get; set; }

  [OnChanged(nameof(Position))]
  void OnPositionChanged(OnChangedData dat)
  {
    transform.position = Position;
  }
}
```

Lag Compensation [Pro]

Understanding the Need for Lag Compensation

Due to varying latencies (ping) of connected players, each player will see the world at a different point in time than the server. For instance, when the client sends an input to the server to shoot its weapon, the target that the client was aiming at would be at a different place in the client than the server. Therefore the client would miss its shoots. Because, usually, from the perspective of the client, the positions of other objects (players) are in the ***remote snapshot timeline***, which is always in the past compared to the timeline of the player-controlled character, which's the ***predicted snapshot timeline***.

On the server, **everything is in the present**. While on the client, **only the player-controlled character is in the present, while other players' (proxies) positions are in the past.** Though this is not always the case, because due to the ability of Netick to do full-world prediction, it's possible to put proxies in the predicted snapshot timeline, in which case lag compensation wouldn't be needed.

So, what's Lag Compensation?

Lag Compensation basically means going back in time to what the client was seeing at the time of the shooting, and simulating its shooting in that past view.

Question: why not just let the client tell the server the target that it hit and how much damage it dealt?
Answer: we can't trust the client. We should never trust the client, especially in game-critical aspects like applying damage. **Lag Compensation gives us authority over hit detection.**

Watch this video for a visual explanation:

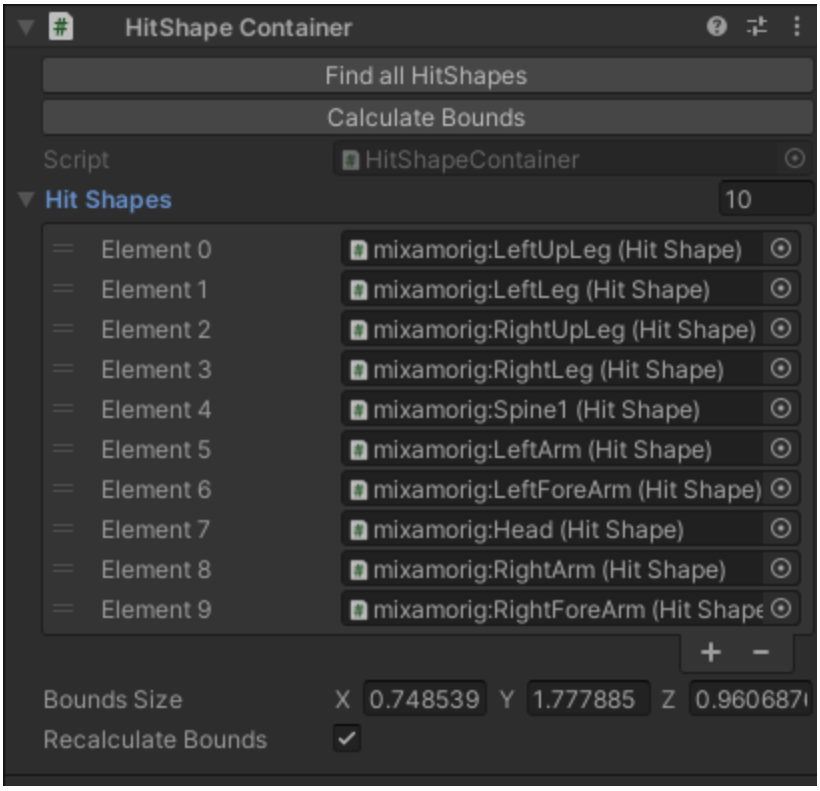
Lag Compensation in Netick

To use Lag Compensation in your project, you first need to enable it in Netick Settings. Go to `Netick Settings -> Lag Compensation` and turn on `Enable`.

Setting up your character for Lag Compensation

You have to add a HitShape component (commonly known as a hitbox) on every part of your character which can move. And in the root of your character, you have to add a `HitShape Container` component which will register all child HitShapes.

HitShape on each bone



HitShape Container on the root of the character.

The hierarchy should be as follows:

```
> Root (with NetworkObject)
    > `HitShape Container`
        > Render Transform
            > Character Rig
                > Character Bone (with HitShape)
```

> **WARNING**
>
> Make sure to enable Lag Compensation in Netick Settings.

Performing a Lag-Compensated Raycast in Unity

```
// lag-compensated Raycast
  if (Sandbox.Raycast(
      shootPos,
      shootDirection,
      out var hit,
      Object.InputSource,
      Mathf.Infinity,
      includeUnityColliders: true,
      queryTriggerInteraction: QueryTriggerInteraction.Ignore))
  {
      if (hit.HitShape != null)
      {
          // code to be executed when a HitShape was hit
      }
  }
```

Performing a Lag-Compensated OverlapSphere in Unity

```
// lag-compensated OverlapSphere
  List<LagCompHit>  overlapSphereHits = new List<LagCompHit>(32);
  Sandbox.OverlapSphere(point,
      _projectileBlastRadius,
      overlapSphereHits,
      InputSource,
      queryTriggerInteraction: QueryTriggerInteraction.Ignore);
```

For a practical example, you might want to get our comprehensive Arena Shooter sample which covers everything we talked about and more: https://netick.net/arena-shooter-sample/

Interest Management [Pro]

Understanding Interest Management

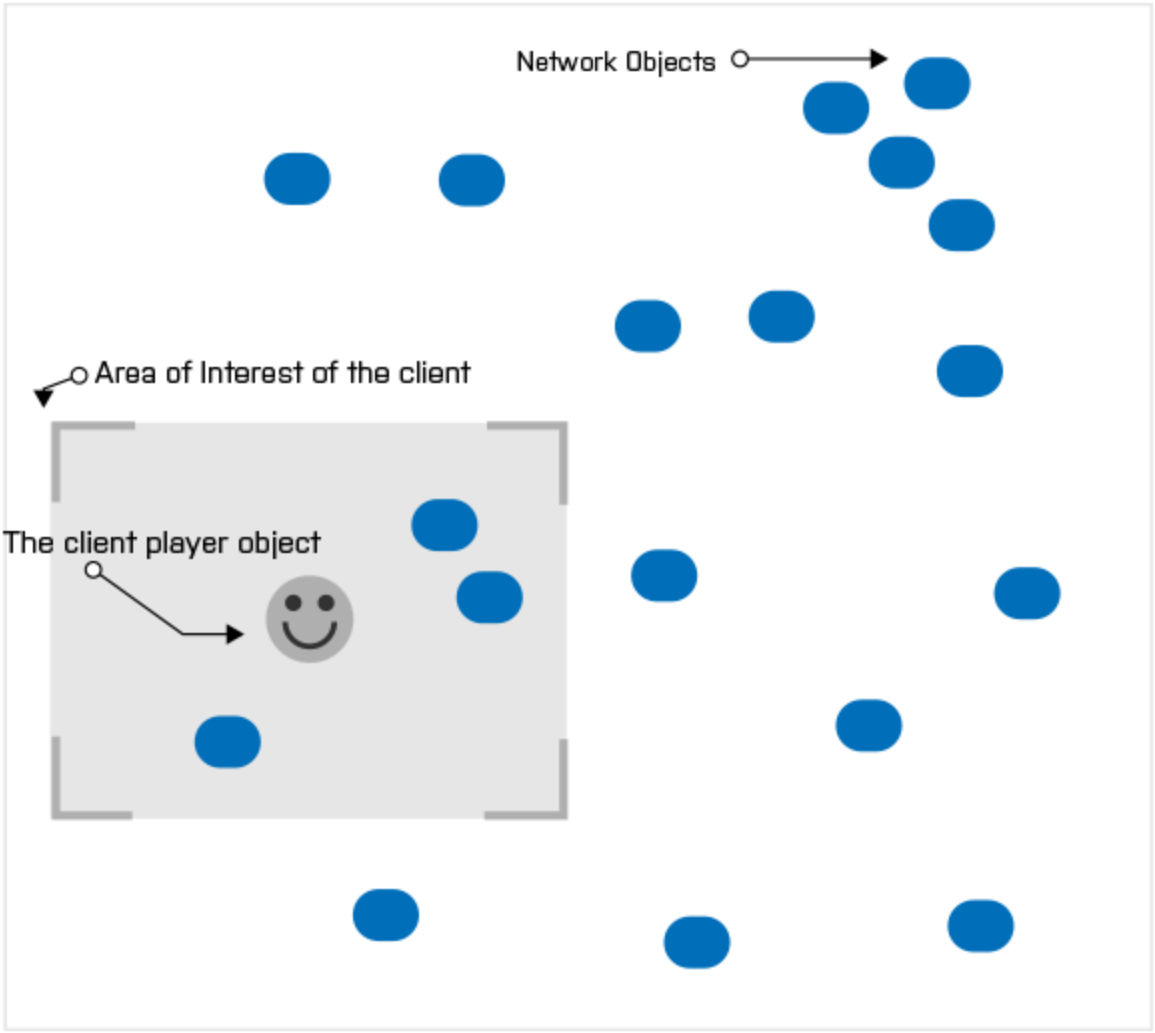
When you have a game with a big world and a high player count, it becomes more and more expensive to replicate every object in the game to every connected client. It's an $O(N^2)$ problem, meaning the bandwidth sent from the server scales quadratically with the number of players connected. Interest Management mitigates this problem by filtering objects that are of no interest to the client. Usually, this is done using Area of Interest, which is the area around the player. The client will only receive objects existing in this area. Interest management is often used in battle-royale scale games

Interest Management is also useful as an anti-cheat measure. For instance, you can replicate team-specific objects to only players of the same team, using a Custom interest group, which we will learn about in this article.

Netick handles Interest Management (filters objects replicated to a client) in two phases:

- Broad Phase Filtering
- Narrow Phase Filtering

Broad Phase Filtering



Broad Phase Filtering is done using a group. An interest management group is a list of network objects that are processed together. Every network object has an option to choose the broad phase filter source, it can be one of three options:

- `Global` : no filtering, the object is replicated to everyone.
- `Area of Interest` : only when a client's Area of Interest intersects this object's group cell it will be replicated.
- `Custom` : using an explicitly assigned custom group through code, with an index. Only clients who explicitly choose to be interested in this group will receive updates to the object.

> **NOTE**
>
> Area of Interest implementation is done using a grid, which you can specify its settings in Netick Settings window.

> **NOTE**
>
> The number of available custom interest groups can be set in Netick Settings.

Explicitly specifying the custom interest group of an object

```
Object.InterestGroup = Sandbox.InterestManagement.CustomGroups[groupIndex];
```

> **NOTE**
>
> The above is only valid when the `Broad Phase Filter` of the object is set to Custom. When it's set to `Area of Interest`, it will only be controllable by Netick - meaning the object will be assigned a group that corresponds to its current position in the world.

Narrow Phase Filtering

Narrow Phase Filtering allows for finer control. Instead of being handled per-group basis, it's done per-object per-client basis. Which is slower. Thus, it should only be used when necessary.

Filtering an object to a specific client

```
Object.SetNarrowphaseInterest(client, false);
```

Specifying the interest of a client

Adding area of interest regions

The area of interest of the client is specified using bounding boxes that you can add.

```
InputSource.AddInterestBoxArea(new Bounds(transform.position, (InterestBox)));
```

Adding custom interest groups

```
InputSource.AddInterestGroup(customInterestGroupIndex);
```

> **CAUTION**
>
> This must be done every `NetworkFixedUpdate` callback since it's cleared at the end of the tick. Usually, you should do this in the movement controller of your player character.

Callbacks

There are two callbacks on `NetworkBehaviour` for when the interest status of an object changes in the client.

```
public override void OnBecameInterested()
{
    // called when this client becomes interested in this object.
```

```
}
public override void OnBecameUninterested()
{
    // called when this client becomes uninterested in this object.
}
```

One of the things you can do using these callbacks is hide/show the object when the client becomes interested/uninterested.

Sending Large Amounts of Data (Byte Arrays)

It's true that RPCs can be used to send small amount of data, but they shouldn't be used to send anything bigger than 500 bytes. For that, the proper way to send data is through the Data Sending API of Netick.

Usage Example

```
const int MyDataId = 5;

unsafe void Sending_Data_Example()
{
  var    text  = "Trying to send some data!";
  byte[] bytes = Encoding.ASCII.GetBytes(text);

  // there are two variations of SendData, one that takes a pointer and one that takes a byte array. We are using the byte array version here.

  // sending to the server (in the client)
  Sandbox.ConnectedServer.SendData(MyDataId , bytes, bytes.Length, TransportDeliveryMethod.Reliable);

  // sending to a certain player (in the server)
  NetworkConnection playerConn = someObject.InputSource as NetworkConnection;
  playerConn.SendData(5, bytes, bytes.Length, TransportDeliveryMethod.Reliable);
}

// called by subscribing it to Sandbox.Events.OnDataReceived
unsafe void OnDataReceived(NetworkSandbox sandbox, NetworkConnection sender, byte id, byte* data, int len,
TransportDeliveryMethod transportDeliveryMethod)
{
  if (id == MyDataId) // is the packet i want
  {
    // converting the data into a managed array (example)
    byte[] buffer = new byte[len]; // don't do this, it's just an example
    for (int i = 0; i < len; i++)
      buffer[i] = data[i];
  }
}
```

> **WARNING**
>
> This functionality is dependent on the underlying transport. Make sure `SendUserData` is implemented on the transport you are using. All of the available transports already implement it.

Networked Procedural Generation

When you want to procedurally generate a level of static objects, you don't need to network any of the objects. Since the procedural generation operation can be deterministic, and therefore all you have to sync to be able to independently create the level in the client is the initial random seed.

Example:

```
public class MapCreator: NetworkBehaviour
{
    [Networked]
    public int RandomSeed { get; set;}
    public override void NetworkStart()
    {
        if (IsServer)
          RandomSeed = Random.Range(0,1000);

        // setting the seed
        Random.InitState(RandomSeed);
        CreateMap();
    }

    public void CreateMap()
    {
        // add here the logic for procedurally generating the map/level.
    }
}
```

Port Forwarding

Port forwarding is a technique that allows a local server (player-hosted server) to be accessible over the internet (WAN). However, many Internet Service Providers (ISPs) block this functionality by default to reduce potential security risks. As a result, self-hosted game servers may be inaccessible to other players.

Some games do not include built-in relay or NAT traversal solutions to work around port forwarding restrictions, such as Minecraft, Terraria, and Valheim.

Below are common solutions to bypass NAT (Network Address Translation) restrictions and enable connectivity.

Cloud Hosting Providers

Hosting your game server on a cloud platform that provides a public IP address is one of the most reliable methods.

Popular cloud service providers include:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure

Relay SDKs

Relay services act as intermediaries (bridges) between clients and servers, facilitating communication without direct port forwarding. While effective, they often introduce additional latency.

Popular relay solutions:

- Steamworks
- Epic Online Services (EOS)
- Unity Relay

Reverse Proxy Services

A reverse proxy runs alongside the game server on the host machine and exposes it to the internet. This approach is conceptually similar to using a relay.

Common reverse proxy tools:

- Ngrok (TCP only)
- Playit.gg
- Hamachi

Contacting Your ISP

If your ISP restricts NAT or blocks incoming connections, you may request a public IP address. Be aware that this might incur additional charges or require specific configuration.

Solution Comparison

Solution	Pros	Cons
Cloud Hosting	Lowest latency, reliable	Higher cost, less control over the environment
Relay SDK	Seamless integration, user-friendly	Additional latency, requires compatible transport layer
Reverse Proxy	Cost-effective, flexible	Added latency, manual setup often required for players
Public IP (ISP)	Works universally with games	May involve extra cost and technical setup

Coming from Netick 1 (Unity)

This is a guide to help you migrate from Netick 1 to Netick 2, for Unity users. It shows you what has changed in Netick 2 and it also shows you many of the new features that Netick 2 brings to your toolset.

First of all, please make a back-up copy of your project. Then carefully read each section of this article. If you need help, please feel free to join our discord.

Importing Netick 2

Assuming you have already downloaded Netick 2 package, delete the root folder of Netick 1 from your project, which is located at `Assets/Netick`. After that, simply unpack/copy Netick 2 into your project. It is recommended to do this in your operating system's File Explorer instead of Unity Project Panel.

Project Settings

Go to Project Settings -> Player -> Other Settings and change these settings to be as follows:

- Allow 'unsafe' code: `true`
- Api compatibility level: `.NET Standard 2.1`

API Naming Changes:

Netick 1	**Netick 2**
NetworkSandbox.GetRpcCaller	NetworkSandbox.CurrentRpcCaller
NetworkSandbox.RpcSource	NetworkSandbox.CurrentRpcSource
NetworkEventsListner	NetworkEventsListener
NetworkBehaviour.ApplyToBehaviour	NetworkBehaviour.GameEngineIntoNetcode
NetworkBehaviour.ApplyToComponent	NetworkBehaviour.NetcodeIntoGameEngine
NetHit	LagCompHit

Game Starter

Now the transport is specified when starting Netick and not using NetickConfig. A field has been added to GameStarter for that.

Network Events Listener

A parameter for disconnection reason (TransportDisconnectReason) has been added to OnClientDisconnected:

Netick 1 | Netick 2

```
public override void OnClientDisconnected(NetworkSandbox sandbox, NetworkConnection client)
{
}
```

Network Behaviour

Add `using Netick.Unity` to every script that you have which inherits from NetworkBehaviour.

```
using Netick;
using Netick.Unity;

public class MyScript : NetworkBehaviour
{
```

```
    ...
}
```

Network Input

Network inputs are now structs instead of classes, which makes it easy to sync them as network properties if needed.

Netick 1 | Netick 2

```
public class MyInput : NetworkInput
{
    public bool       ShootInput;
    public float      MoveDirX, MoveDirY;
}
```

Because they are now value types instead of reference types, this means the previous method of populating them won't work anymore. Instead, you have to use another call to update the input.

Netick 1 | Netick 2

```
public override void OnInput(NetworkSandbox sandbox)
{
    var input        = sandbox.GetInput<BombermanInput>();
    input.Movement   = GetMovementDir();
    input.PlantBomb |= Input.GetKeyDown(KeyCode.Space);
}
```

OnChanged

Now OnChanged methods must have a parameter of OnChangedData type which can be used to retrieve the previous property value:

Netick 1 | Netick 2

```
[Networked]
public int Health { get; set; }

[OnChanged(nameof(Health ))]
private void OnHealthChanged(int previous)
{
      // Something that happens when the Health property changes
}
```

It also now supports retrieving previous array values:

```
[Networked(size: 32)]
public NetworkArray<int> ArrayExample = new NetworkArray<int>(32);

[OnChanged(nameof(IntArray))]
private void OnArrayExampleChanged(OnChangedData onChangedData)
{
  // getting the changed element value directly

  var changedPreviousElementValue = onChangedData.GetArrayPreviousElementValue<int>();
```

```
    // or just getting the index

    var changedPreviousElementIndex = onChangedData.GetArrayChangedElementIndex();

    // or maybe getting the previous value of another index we want

    var someRandomPreviousElementValue = onChangedData.GetArrayPreviousElementValue<int>(13);
}
```

Behavioral Change

[OnChanged] methods now will be called for all non-default initialization values - property definition assignments and inspector values. And this happens for the first time when the object is first created, before NetworkStart is called. So if you try to access a class instance variable inside the [OnChanged] method which is initialized inside NetworkStart, it can cause a null reference exception - because NetworkStart is invoked after [OnChanged] method, not before. To fix this, transfer all class instance variables initialization into NetworkAwake (which is called before the first [OnChanged] ever).

Network Arrays

Network arrays syntax has changed a little bit. They are now field members instead of property members.

Netick 1 | Netick 2

```
[Networked (size: 3)]
public NetworkArray<int> IntArrayExample { get; set; }
```

WARNING

Regarding network arrays for Netick 2: `size` of [Networked(size: 32)] must be the same as the value that is passed to the array constructor `new NetworkArray<int>(32)`

As you can see, it's now possible to have initialization values for network arrays.

Network Array Struct

Netick 2 introduces a new type of network array, network arrays that are completely value types - Network Array Structs. These are fixed-size struct arrays available only in 4 fixed sizes: 8, 16, 32, and 64.

Network Array Structs are pretty useful since they can used as members of another struct, or even nested inside other arrays.

```
// Network Struct Array Examples

[Networked]
public NetworkArrayStruct8<int>                    IntFixedArray { get; set; } = new int[] {1 , 4 ,5}.ToNetworkStructArray8();

[Networked]
public NetworkArrayStruct8<NetworkArrayStruct8<int>> ArrayOfArrays { get; set; };
```

NOTE

Network Array Structs are treated as if they were simple struct types like `int` or `float` , so they must be defined as a property not as a field (like normal NetworkArray that is non-fixed size).

Changing elements of Network Array Struct

Because Network Array Structs are structs, the whole array will be replaced even when you change a single element. To avoid bugs, this should be how you change array elements:

```
IntFixedArray = IntFixedArray.Set(index, value);
// as you can see, we are reassigning the property with the new changed array which has the change.
```

Network Structs

Now all structs are networked by default, so you don't need to add [Networked] to them or even implement custom equality. You no longer have a limit size for a single struct too. You can also now have nested structs. So this works as expected:

```
public struct MyNestedStruct
{
    public int                      Int1;
    public bool                     Bool1;
    public float                    Float1;
    public double                   Double1;
  }

public struct MyStruct
{
    public MyNestedStruct           MyNestedStruct;
    public NetworkArrayStruct8<int> StructArray;
    public int                      Int1;
    public bool                     Bool1;
    public float                    Float1;
    public double                   Double1;
}
[Networked]
public MyStruct MyStructProperty {get; set;}
```

Input Source

Now, to change the input source of an object you do that directly using the InputSource property setter:

Netick 1 | Netick 2

```
// assigning an input source to network object:

Object.PermitInput(myNewInputSource);

// removing the input source from the object:

Object.RevokeInput();
```

Callbacks of NetworkBehaviour, OnInputPermitted and OnInputRevoked, have been removed and replaced by one single callback:

Netick 1 | Netick 2

```
public override void OnInputPermitted()
{
  // called on the InputSource machine when InputSource is now equal to this machine.
}

public override void OnInputRevoked()
{
  // called on the InputSource machine when this machine is no longer the InputSource.
}
```

Rpcs

At this moment in time, `string` is not supported as a parameter to Rpcs. Instead, fixed-size structs can be used:

Netick 1 | Netick 2

```
[Rpc]
public void MyRpc(string myString)
{

}
```

Now, you can have static Rpcs on NetworkBehaviour classes which can be pretty useful.

```
[Rpc]
public static void MyStaticRpc(NetickEngine engine, int someRpcPara)
{
  var sandbox = engine.UserObject as NetworkSandbox;
}


// this is how you would call the rpc:

MyStaticRpc(Sandbox.Engine, 56);
```

Note that they must have a NetickEngine as the first parameter.

Lag Compensation

LagCompensation component class has been removed.

Netick 1 | Netick 2

```
Sandbox.GetComponent<LagCompensation>().Raycast(...);
```

Interpolation

Interpolator is now an non-generic struct.

To find an Interpolator, now you simply use the name of the property instead of using an Id.

Netick 1 | Netick 2

```
[Networked][Smooth(6)]
public MyType SomeProperty {get; set;}
public override void NetworkStart()
{
    var interpolator = FindInterpolator<MyType>(6);
}
```

Also, now [Smooth] takes a parameter to specify if it should give auto-interpolated values inside NetworkRender or not, by specifying a true or false value for `auto` parameter of [Smooth].

Accessing Interpolation Data

To get interpolation data, now instead of using To, From, and Alpha fields of Interpolator, you use GetInterpolationData method of Interpolator struct:

```
bool didGetData = interpolator.GetInterpolationData<int>(InterpolationMode.Auto, out var from, out var to, out float alpha);
```

It also now supports getting interpolation data for network arrays:

```
int myIndex = 4;
bool didGetData = interpolator.GetInterpolationData<int>(InterpolationMode.Auto, myIndex, out var from, out var to, out float alpha);
```

Replication

Netick 2 introduces a new replication method called Pessimistic Replication (in contrast to Optimistic Replication, which was the only replication method in Netick 1), which ensures that the client always receives the full state together, not partial, but always the full state. In addition, this new replication method uses delta encoding to highly reduce the bandwidth required. This replication method eliminates the burden of having to account for the potential bugs caused by not always having the entire changed state together using Optimistic Replication.

As of now, this is the default and only replication method. But the old Optimistic Replication will come back later in the future. Pessimistic Replication as of now works with AoI by disabling delta encoding, but this will change in the future. When that happens, Pessimistic Replication will be better than Optimistic Replication for almost every single situation. This is why it has not been a priority to make Optimistic Replication present in Netick 2 from the start.

Network Transport

> **NOTE**
>
> If you are not a transport or a transport wrapper developer, you can ignore this section.

In Netick 1, your network transport main script was inheriting from NetworkTransport, which by itself was inheriting from ScriptableObject. But now that's not possible anymore, since ScriptableObject is a Unity class.

Now, NetworkTransport does not inherit from ScriptableObject, which means you no longer can have assets on your project representing a transport like in Netick 1.

To solve this, a wrapper class has been added NetworkTransportProvider, which inherits from ScriptableObject and wraps the network transport:

```
[CreateAssetMenu(fileName = "LiteNetLibTransportProvider", menuName = "Netick/Transport/LiteNetLibTransportProvider", order = 1)]
public class LiteNetLibTransportProvider : NetworkTransportProvider
{
  public override NetworkTransport MakeTransportInstance() => new LiteNetLibTransport();
}
```

`MakeTransportInstance` is called by Netick to create an instance of the transport.

Netick now only receives data in the form of BitBuffer. BitBuffer.SetFrom is used to set a pointer to the data which BitBuffer will use. Take a look at the new LiteNetLib transport to understand how it all works.

```
public unsafe void INetEventListener.OnNetworkReceive(NetPeer peer, NetPacketReader reader, DeliveryMethod deliveryMethod)
{
  if (_clients.TryGetValue(peer, out var c))
  {
    var len = reader.AvailableBytes;
    reader.GetBytes(_bytes, 0, reader.AvailableBytes);

    fixed(byte* ptr = _bytes)
    {
      _buffer.SetFrom(ptr, len, _bytes.Length);
      NetworkPeer.Receive(c, _buffer);
    }
```

```
  }
}
```

Network Transform

NetworkTransform is a built-in component to network the `Transform` component. It syncs the position and rotation of the `Transform` .

Settings:

- Render Transform: assign here the transform that you want to use to display smoothed movement. Should be a child of this GameObject.
- Settings: the replications settings of the NetworkTransform. Choose what you want to sync, and whether you want to enable compression for it or not.
- Interpolation Source: read here for a detailed explanation.
- Transform Space: the space used when replicating the data.
- Precision: the precision of the data compression.

Teleportation

Since `Render Transform` is always interpolated between two ticks, when you instantly move your object into another position, interpolation would still be active on that object, which is undesirable. To fix this, when you want to instantly move the object and disable interpolation for that duration, you must use the `Teleport` method.

Example:

```
MyNetworkTransform.Teleport(newPosition);
```

Network Rigidbody

NetworkRigidbody is a built-in component to network the `Rigidbody` component. It syncs the position and rotation of the `Rigidbody`, in addition the physical state.

Settings:

- Render Transform: assign here the transform that you want to use to display smoothed movement. Should be a child of this GameObject.
- Settings: the replications settings of the NetworkTransform. Choose what you want to sync, and whether you want to enable compression for it or not.
- Interpolation Source: read here for a detailed explanation.
- Transform Space: the space used when replicating the data.
- Precision: the precision of the data compression.

Teleportation

Since `Render Transform` is always interpolated between two ticks, when you instantly move your object into another position, interpolation would still be active on that object, which is undesirable. To fix this, when you want to instantly move the object and disable interpolation for that duration, you must use the `Teleport` method:

Example:

```
MyNetworkTransform.Teleport(newPosition);
```

Network Rigidbody2D

NetworkRigidbody2D is a built-in component to network the `Rigidbody2D` component. It syncs the position and rotation of the `Rigidbody2D`, in addition the physical state.

Settings:

- Render Transform: assign here the transform that you want to use to display smoothed movement. Should be a child of this GameObject.
- Settings: the replications settings of the NetworkTransform. Choose what you want to sync, and whether you want to enable compression for it or not.
- Interpolation Source: read here for a detailed explanation.
- Transform Space: the space used when replicating the data.
- Precision: the precision of the data compression.

Teleportation

Since `Render Transform` is always interpolated between two ticks, when you instantly move your object into another position, interpolation would still be active on that object, which is undesirable. To fix this, when you want to instantly move the object and disable interpolation for that duration, you must use the `Teleport` method:

Example:

```
MyNetworkTransform.Teleport(newPosition);
```

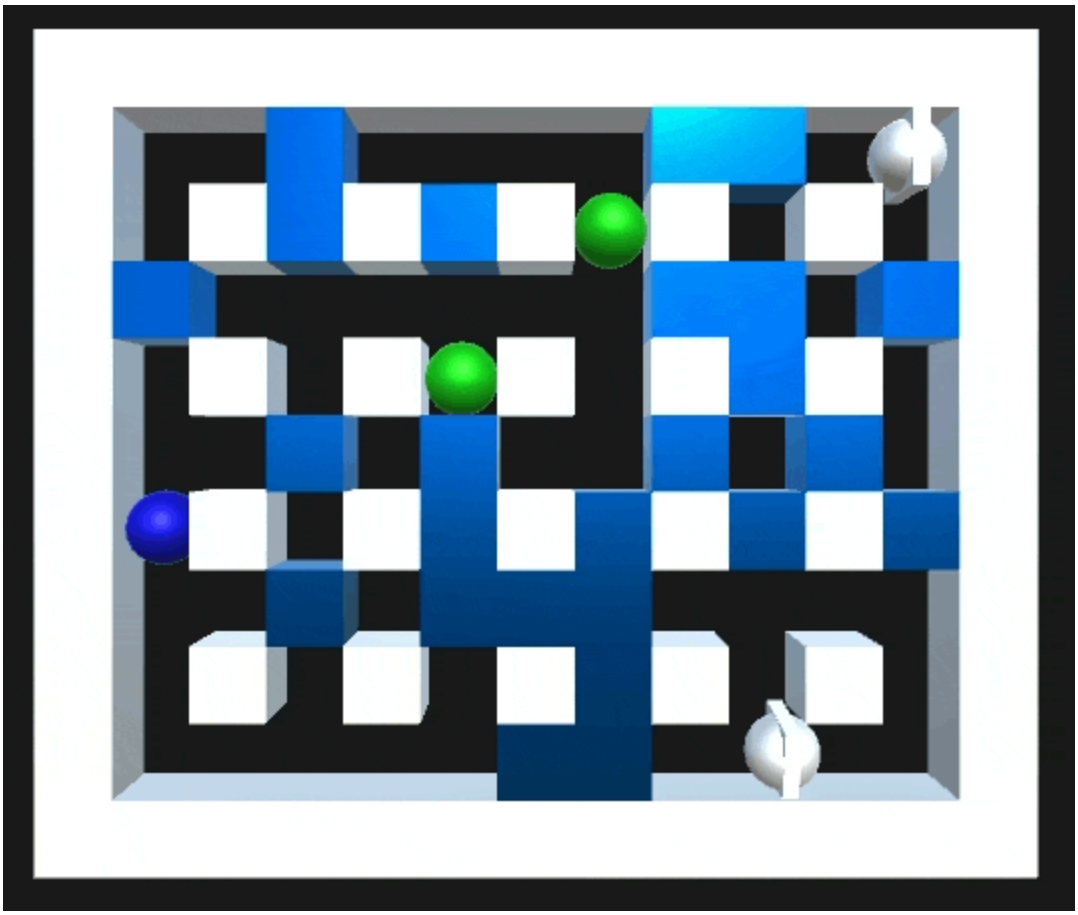
Network Animator

NetworkAnimator is a built-in component to network the `Animator` component. It syncs Animator parameters, and optionally the layer state and weights.

Settings:

- Settings: replications settings of the NetworkAnimator . Choose what you want to sync.
- Interpolation Source: read here for a detailed explanation.

Bombmerman (Unity)



This is a very simple clone of the popular Bomberman game. It's a 4-player game where players try to kill each other using bombs.

Features

- Simple character controller
- Procedural level generation
- Pickups

Download

This is included in the main Netick package.

https://github.com/NetickNetworking/NetickForUnity/tree/master/Samples~/Bomberman

Simple First Person (Unity)

This is a very simple FPS game.

Features

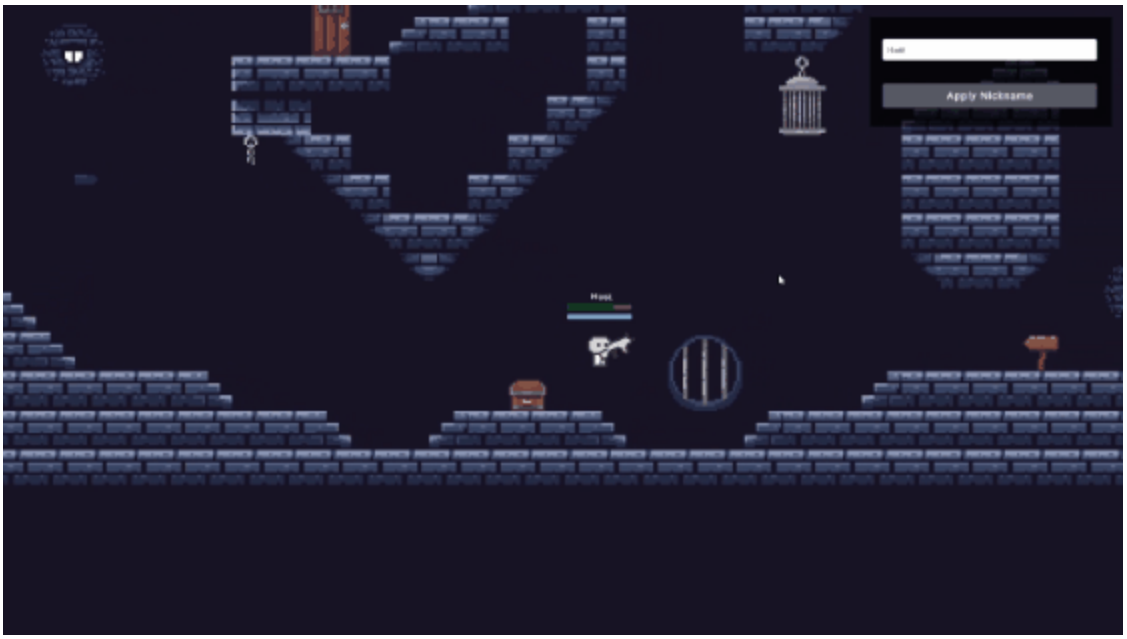
- Simple FPS character controller

Download

This is included in the main Netick package.

https://github.com/NetickNetworking/NetickForUnity/tree/master/Samples~/First%20Person%20Shooter

Dungeon Frenzy (Unity)



A simple 2D fast-paced platformer shooter. Made by one of our community members.

Features

- Efficient projectile spawn system
- Spawn system
- Custom interpolation on weapon rotation
- Optional lag compensation scripts
- Server-authoritative hit-detection
- Custom execution order

Download

https://github.com/StinkySteak/dungeon-frenzy

Transports

LiteNetLib:

- Protocol: UDP
- Platforms: All, except WebGL
- Download: included with Netick
- Maintainer: Netick Staff

UnityTransport

- Protocol: UDP & WebSocket
- Platforms: All
- Download: https://github.com/karrarrahim/UnityTransport-Netick
- Maintainer: Netick Staff

Facepunch (Steam):

- Protocol: UDP
- Platforms: Windows, Mac OS, and Linux
- Download: https://github.com/Milk-Drinker01/Netick2-Facepunch-Transport
- Maintainer: community

Steamworks.NET (Steam):

- Protocol: UDP
- Platforms: Windows, Mac OS, and Linux
- Download: https://github.com/Milk-Drinker01/Netick2-Steamworks-Transport
- Maintainer: community

LiteNetLib with NAT Punch support:

- Protocol: UDP
- Platforms: All, except WebGL
- Download: https://github.com/StinkySteak/NetickLiteNetLibNatPunchTransport
- Maintainer: community

WebRTC:

- Protocol: UDP
- Platforms: WebGL
- Download: https://github.com/StinkySteak/NetickWebRTCTransport
- Maintainer: community

Multiplex Transport:

Allows you to use more than one transport together.

- Download: https://github.com/StinkySteak/NetickMultiplexTransport
- Maintainer: community

How To Write a Transport Wrapper

Introduction

A transport is the low-level component that does the actual data sending, receiving and handling connections.

This guide will show how to implement a wrapper for the Unity Transport

Defining the Connection Wrapper

First you need to define a connection class which you will pass to Netick. This represents a transport connection. It must implement several properties and a send method:

```
public unsafe class NetickUnityTransportConnection : TransportConnection
{
  public NetickUnityTransport                        Transport;
  public Unity.Networking.Transport.NetworkConnection Connection;

  public override IEndPoint                          EndPoint => Transport._driver.GetRemoteEndpoint(Connection).ToNetickEndPoint();
  public override int                                Mtu      => MaxPayloadSize;

  public int                                         MaxPayloadSize;

  public NetickUnityTransportConnection(NetickUnityTransport transport)
  {
    Transport = transport;
  }

  public unsafe override void Send(IntPtr ptr, int length)
  {
    if (!Connection.IsCreated)
      return;
    Transport._driver.BeginSend(NetworkPipeline.Null, Connection, out var networkWriter);
    networkWriter.    WriteBytesUnsafe((byte*)ptr.ToPointer(), length);
    Transport._driver.EndSend(networkWriter);
  }
}
```

The `Send` method is called by Netick when it wants to send a packet to this connection. `Transport` represents the UnityTransport transport class which we will talk about in a bit. `Connection` represents the UnityTransport connection that corresponds to this `NetickUnityTransportConnection` type that we will pass into Netick.

Defining the End Point Wrapper

Let's also define an end point wrapper over UnityTransport `NetworkEndPoint`, and a extension class to do the conversion:

```
public static class NetickUnityTransportExt
{
    public static NetickUnityTransportEndPoint ToNetickEndPoint(this NetworkEndpoint networkEndpoint) =>
new NetickUnityTransportEndPoint(networkEndpoint);
}

public unsafe class NetickUnityTransport : NetworkTransport
{
  public struct NetickUnityTransportEndPoint : IEndPoint
  {
    public NetworkEndpoint EndPoint;
    string       IEndPoint.IPAddress => EndPoint.Address.ToString();
    int          IEndPoint.Port      => EndPoint.Port;
    public NetickUnityTransportEndPoint(NetworkEndpoint networkEndpoint)
    {
      EndPoint = networkEndpoint;
    }
    public override string ToString()
    {
      return $"{EndPoint.Address}";
    }
  }
```

Defining the Transport Wrapper

```
public unsafe class NetickUnityTransportConnection : TransportConnection
{

}
```

Let's add a few fields which will be important in the functionality of the transport.

```
    private NetworkDriver                                                                   _driver;

    private Dictionary<Unity.Networking.Transport.NetworkConnection, NetickUnityTransportConnection> _connectedPeers        = new();
    private Queue<NetickUnityTransportConnection>                                           _freeConnections       = new();
    private Unity.Networking.Transport.NetworkConnection                                    _serverConnection;

    private NativeList<Unity.Networking.Transport.NetworkConnection>                        _connections;
```

`_driver` represents an instance of a UnityTransport manager. `_connectedPeers` contains the a dictionary that maps between the UnityTransport connection type, and the transport wrapper connection type. `_freeConnections` is a pool for free connections that we will use. `_serverConnection` is only relevant when the transport is started as a client, it represents the UnityTransport connection to the server. And `_connections` is the buffer that is used by UnityTransport for the connections.

```
    private BitBuffer                                                                       _bitBuffer;
    private byte*                                                                           _bytesBuffer;
    private int                                                                             _bytesBufferSize         = 2048;
    private byte[]                                                                          _connectionRequestBytes  = new byte[200];
    private NativeArray<byte>                                                               _connectionRequestNative = new NativeArray<byte>
(200, Allocator.Persistent);
```

`_bitBuffer` is the buffer that is passed to Netick when receiving a packet. Netick only receives the packets in the form of a `BitBuffer`.

`_bytesBuffer` is an unsafe buffer that is used with `_bitBuffer`. `_connectionRequestBytes` is a managed buffer for the connection request.

In the constructor, we allocate `_bytesBuffer`. And we make sure to deallocate, in addition to disposing of `_connectionRequestNative`.

```
    public NetickUnityTransport()
    {
      _bytesBuffer = (byte*)UnsafeUtility.Malloc(_bytesBufferSize, 4, Unity.Collections.Allocator.Persistent);
    }

    ~NetickUnityTransport()
    {
      UnsafeUtility.Free(_bytesBuffer, Unity.Collections.Allocator.Persistent);
      _connectionRequestNative.Dispose();
    }
```

Let's override the `Init` method. This method is called by Netick once to initialize the transport. We initialize the `_bitBuffer` and the UnityTransport network driver `_driver`, and also let's initialize `_connections` buffer.

```
    public override void Init()
    {
      _bitBuffer      = new BitBuffer(createChunks: false);
      _driver      = NetworkDriver.Create(new WebSocketNetworkInterface());
      _connections = new NativeList<Unity.Networking.Transport.NetworkConnection>(Engine.IsServer ? Engine.Config.MaxPlayers :
0, Unity.Collections.Allocator.Persistent);
    }
```

The `Run` method is called by Netick when starting Netick. `Shutdown` is called when shuting down Netick.

```
    public override void Run(RunMode mode, int port)
    {
```

```
    if (Engine.IsServer)
    {
      var endpoint = NetworkEndpoint.AnyIpv4.WithPort((ushort)port);

      if (_driver.Bind(endpoint) != 0)
      {
        Debug.LogError($"Failed to bind to port {port}");
        return;
      }
      _driver.Listen();
    }

    for (int i = 0; i < Engine.Config.MaxPlayers; i++)
      _freeConnections.Enqueue(new NetickUnityTransportConnection(this));
  }

  public override void Shutdown()
  {
    if (_driver.IsCreated)
      _driver.   Dispose();
    _connections.Dispose();
  }
```

`Connect` method is called by Netick in the client when wanting to connect to the server.

`Disconnect` method is called when you are kicking or disconnecting a connection.

```
  public override void Connect(string address, int port, byte[] connectionData, int connectionDataLength)
  {
    var endpoint        = NetworkEndpoint.LoopbackIpv4.WithPort((ushort)port);
    if (connectionData != null)
    {
      _connectionRequestNative.CopyFrom(connectionData);
      _serverConnection = _driver.Connect(endpoint, _connectionRequestNative);
    }
    else
      _serverConnection = _driver.Connect(endpoint);
  }

  public override void Disconnect(TransportConnection connection)
  {
    var conn = (NetickUnityTransport.NetickUnityTransportConnection)connection;
    if (conn.Connection.IsCreated)
      _driver.Disconnect(conn.Connection);
  }
```

Now let's override the last method which is `PollEvents`. This is called by Netick each frame, to poll network events on the transport.

Here we are handling everything from making new connections, handling disconnections, and receiving packets, etc.

```
  public override void PollEvents()
  {
    _driver.ScheduleUpdate().Complete();

    if (Engine.IsClient && !_serverConnection.IsCreated)
      return;

    // reading events
    if (Engine.IsServer)
    {
      // clean up connections.
      for (int i = 0; i < _connections.Length; i++)
      {
        if (!_connections[i].IsCreated)
        {
          _connections.RemoveAtSwapBack(i);
          i--;
        }
      }
```

```
    // accept new connections in the server.
    Unity.Networking.Transport.NetworkConnection c;
    while ((c = _driver.Accept(out var payload )) != default)
    {
      if (_connectedPeers.Count >= Engine.Config.MaxPlayers)
      {
        _driver.Disconnect(c);
        continue;
      }

      if (payload.IsCreated)
        payload.CopyTo(_connectionRequestBytes);
      bool accepted = NetworkPeer.OnConnectRequest(_connectionRequestBytes, payload.Length, _driver.GetRemoteEndpoint(c).ToNetickEndPoint());

      if (!accepted)
      {
        _driver.Disconnect(c);
        continue;
      }

      var connection        = _freeConnections.Dequeue();
      connection.Connection = c;
      _connectedPeers.Add(c, connection);
      _connections.   Add(c);

      connection.MaxPayloadSize = NetworkParameterConstants.MTU - _driver.MaxHeaderSize(NetworkPipeline.Null);
      NetworkPeer.    OnConnected(connection);
    }

    for (int i = 0; i < _connections.Length; i++)
      HandleConnectionEvents(_connections[i], i);
  }
  else
    HandleConnectionEvents(_serverConnection, 0);
}


private void HandleConnectionEvents(Unity.Networking.Transport.NetworkConnection conn, int index)
{
  DataStreamReader  stream;
  NetworkEvent.Type cmd;

  while ((cmd = _driver.PopEventForConnection(conn, out stream)) != NetworkEvent.Type.Empty)
  {
    // game data
    if (cmd == NetworkEvent.Type.Data)
    {
      if (_connectedPeers.TryGetValue(conn, out var netickConn))
      {
        stream.     ReadBytesUnsafe(_bytesBuffer, stream.Length);
        _bitBuffer.    SetFrom(_bytesBuffer, stream.Length, _bytesBufferSize);
        NetworkPeer.Receive(netickConn, _bitBuffer);
      }
    }

    // connected to server
    if (cmd == NetworkEvent.Type.Connect && Engine.IsClient)
    {
      var connection = _freeConnections.Dequeue();
      connection.Connection = conn;

      _connectedPeers.Add(conn, connection);
      _connections.   Add(conn);

      connection.MaxPayloadSize = NetworkParameterConstants.MTU - _driver.MaxHeaderSize(NetworkPipeline.Null);
      NetworkPeer.    OnConnected(connection);
    }

    // disconnect
    if (cmd == NetworkEvent.Type.Disconnect)
    {
      if (_connectedPeers.TryGetValue(conn, out var netickConn))
      {
        TransportDisconnectReason reason = TransportDisconnectReason.Shutdown;
```

```
          NetworkPeer.     OnDisconnected(netickConn, reason);
          _freeConnections.Enqueue(netickConn);
          _connectedPeers. Remove(conn);
        }

        if (Engine.IsClient)
          _serverConnection   = default;
        if (Engine.IsServer)
          _connections[index] = default;
      }
    }
  }
```

Defining the Transport Provider

Finally, we have to define a transport provider (a ScriptableObject), which will be used by Netick to create a new instance of the transport wrapper.

```
[CreateAssetMenu(fileName = "UnityTransportProvider", menuName = "Netick/Transport/UnityTransportProvider", order = 1)]
public class        UnityTransportProvider : NetworkTransportProvider
{
    public override NetworkTransport  MakeTransportInstance() => new NetickUnityTransport();
}
```

We can then go to the Assets folder in Unity, and double click and go to Create->Netick->Transport->UnityTransportProvider. Assign the created instance to your GameStarter transport field, and you are done!