BUFF BUFF MULTIPLAYER SYSTEM

Version 1.0

We chose to implement our own multiplayer system that fits better our project, without the giant overhead that NGO (Netcode for GameObjects) and third-party solutions (like mirror) commonly have, while still providing all the tools needed to create a very-performatic and reliable network platform.

This document was created with the purpose of being an implementation guide as also a “reliability assurance procedure”. Everyone who's trying to implement any new system into a project using this library shall follow the instructions found along the guide (ignoring some special exceptions)

The system provides many features (which some are very worthy to note, so they’re listed below):

- **Reload-proof** (you can make non-aggressive code changes runtime and the server state will be kept. Clients will need rejoin with no issues)

- **Reliable and Fast** (you can choose when the reliability is really needed)

- **Small Overhead** (what you want/do is what you get)

- **Packet Based (Status & Actions)** (no Rpc commands or calls, you can use the packets directly, offering more control and performance/configuration)

- **Reconnect Friendly** (can sync retroactive states easily with no problem)

- **Data Lightweight** (tries to save data for mobile devices)

Table Of Contents

Main Components

Network Manager

Main network component, manages the client and/or server connection, packets sending/handling and network object states across the network. Can be extended to implement custom specific behaviors. You can access the current Network Manager instance using

```
var manager = NetworkManager.Instance;

manager.IsServerRunning //Returns if there's a server running locally
manager.IsClientRunning //Returns if there's a client running locally
manager.ClientId //local client id (If the client is running)
```

Network Transport

Internal interface used in background to handle connections between server and clients (players). Normally you shouldn't make changes to the transport. The default transport method is **UDP** (User Datagram Protocol), a simple, fast but unreliable protocol. A reliability layer is used to guarantee reliability on the communication.

Currently the following transport methods are supported:

- **UDP (Reliable & Unreliable)**
- **Local Split Screen (2 players, useful for testing)**

Network Identifier

Used to sync an object reference across all network-ends. **All objects that need to sync data / state need a network identifier.** This is mainly formed by three fields:

- **Id (32 byte hex number)**: The object unique identifier

- **OwnerId (int):** References the owner of the object. Default is **-1 (Server)**

- **Prefab Id (32 byte hex number):** The object source prefab id (For starting objects, the prefab value is **0000000000000000**)

```
1 public NetworkId Id => id;
2
3 public int OwnerId => ownerId;
4
5 public NetworkId PrefabId => prefabId;
```

These three fields combined provides all the needed information for basic-object handling

Network Prefab Registry

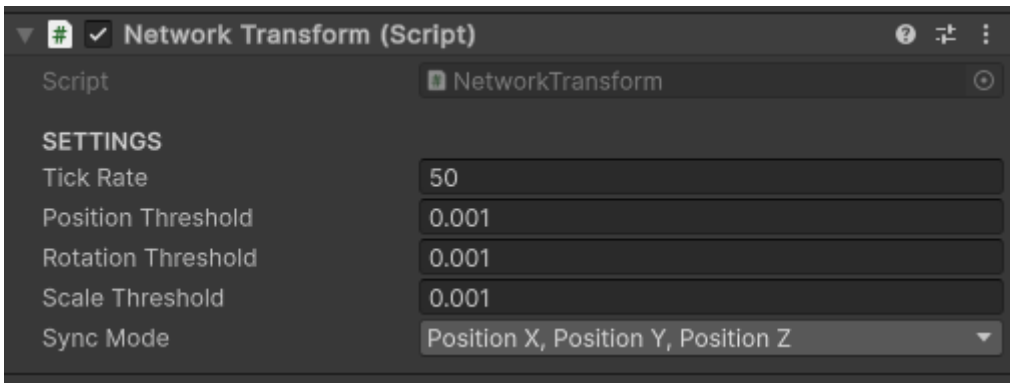
Used to assure that network objects can be instantiated on client-side correctly. All network objects instantiated runtime **shall** be originated by a registered prefab.

Network Behaviour

Base class to create any network related interaction. Any behavior / state - sync normally relies on a network behaviour instance. All network behaviors are mono behaviors so you just need to add them to a GameObject with NetworkIdentity and they will start to behave when connected on a network.

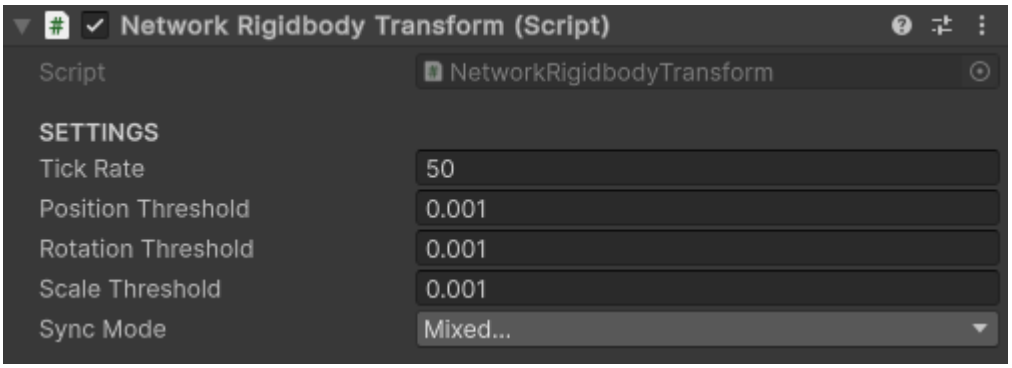
Network Transform

Basic common NetworkBehaviour. Used to sync an object transformation across the network. You can set the update threshold and even which components of Position, Rotation and Scale should be synced **(DO NOT CHANGE SYNC MODE DURING PLAYTIME).** If the tick rate is -1, the defaultTickRate from NetworkManager will be used



Network Rigidbody Transform

Extends the NetworkTransform syncing with a better support for physics based objects, syncing the velocity and angularVelocity components as needed. If the sync mode is Position X | Rotation Y, only velocity X component and angular velocity Y component will be synced, saving data. The inspector view is the same as NetworkTransport.

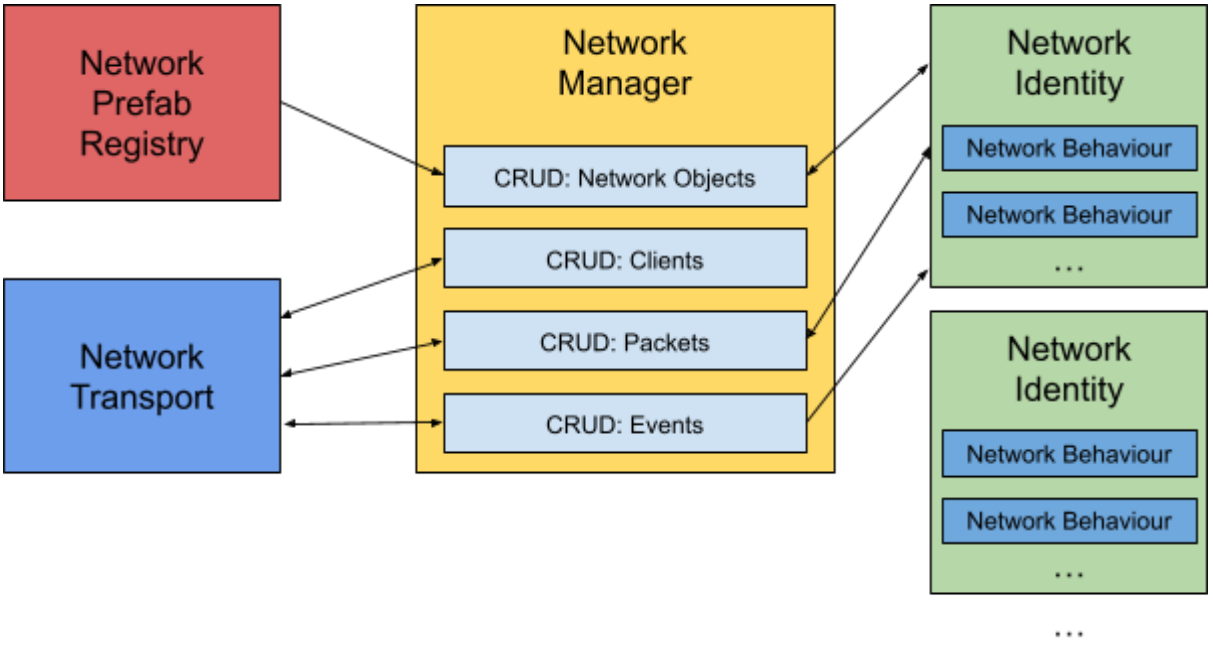


Network Animator

Syncs animation layer states, parameters, timing and transitions of an Animator across the network. Triggers are not automatically synced if you set them directly on the Animator. Use the NetworkAnimator.SetTrigger (with Authority) method to set them correctly.

```
1 public NetworkAnimator animator;
2 [...]
3 animator.SetTrigger("Punch");
```

Basic Structure Diagram



(CRUD stands for: Create, Read, Update and Delete)

In 99.99% usage cases you will just need to interact with NetworkManager and the NetworkBehaviour classes. The behaviour base class offers many **events, methods and properties** that may cover almost all use cases.

Sending/Receiving custom packets can (and shall) also be handled by the behaviors, where **each NetworkIdentity receives the callbacks for** only the **packets owned by him**, but don't worry, **you can add custom packet listeners when needed** (just remember to remove the listener as well on the OnDisable method).

All state/action synchronization may also be done using the events or your own methods inside the behaviors. **Any manager synced across the network (as a LevelManager for example) should be derived from a NetworkBehaviour.**

Network Behaviour

This section will show information of the base class NetworkBehaviour alongside with basic implementation details. As said before, the behaviour class is composed of three main component types: **Event Callbacks, Properties** and **Methods.** You can find a list of all them bellow:

Properties

Properties		
Side	**Name**	**Description**
Both	**Identity (NetworkIdentity)**	Returns the identity component attached to the network object
Both	**HasAuthority (bool)**	Returns if the local environment has authority over the object. When the OwnerId is -1 (Default), HasAuthority will return true only on the server. When the object is owned by a client, it will only be true in the client local environment. **Use this to control input / state handling.**
Both	**IsOwnedByAClient (bool)**	Returns if the OwnerId is not -1
Both	**Id (NetworkId)**	Returns the Id of the network object
Both	**OwnerId (int)**	Returns the owner id. If you want to change the ownership of the object you can use the method **SetOwner** (only works on the server or with authority)
Both	**PrefabId (NetworkId)**	Returns the prefabId used to create the object. If the object is a scene object (aka exists before the network starts), the prefab id will be empty (0000000000000000)

Event Callbacks

Event Callbacks		
Side	**Name**	**Description**

Server	**OnServerReceivePacket(IOwnedPacket packet, int clientId)**	Called when the object receives an OwnedPacket on the server side. **The packets received are not default broadcast to the clients. You should do it on this callback.** (See the section about packets for more info)
Client	**OnClientReceivePacket(IOwnedPacket packet)**	Called when the object receives an OwnedPacket on the client side. (See the section about packets for more info). **THIS IS ALSO CALLED ON SERVER (NOT HOST) ENDS**
Both	**OnSpawned(bool isRetroactive)**	Called when the object is spawned on the network end (Even the pre existing ones will call this method). **Very useful to load the initial state of an object.**
Server	**OnClientConnected(int clientId)**	Called when a client joins the server when the object already exists. **Very useful for retroactive state loading, when the server needs to send data to the client of the actual object state.**
Server	**OnClientDisconnected(int clientId)**	Called when a client leaves the server
Server	**OnDespawned**	Called when an object is despawned (normally along with OnDestroy)
Both	**OnActiveChanged(bool active)**	Called when the object. Works the same as OnEnable / OnDisable but on the network environment.
Both	**OnOwnerChanged(int newOwner)**	Called when the ownership of a network object changes

Methods

Methods		
Side	**Name**	**Description**
Server	**ServerBroadcastPacket(IPacket packet, bool reliable = false)**	Broadcast a packet to all the clients. You can choose if the packet should be reliable. If it's a constant state update, where a packet loss is not an issue, you may set it

		as false for performance and low-data usage
Server	**ServerBroadcastPacketExceptFor(IPacket packet, int except, bool reliable = false)**	Broadcast a packet to all the clients except one. Useful when spreading a packet received from a client.
Server	**ServerSendPacket(IPacket packet, int clientId, bool reliable = false)**	Sends a packet a client
Client	**ClientSendPacket(IPacket packet, bool reliable = false)**	Sends a pocket to the server
Auth	**SendPacket(IPacket packet, bool reliable = false)**	Sends a packet to the other network end. Automatically changes based if the owner is a client or the server.
Both	**GetPacketListener<T>()**	Get a packet listener to register your own listeners. Useful to handle non-owned packets
Auth	**Despawn()**	Despawns (a.k.a. destroys) a network object across the network.
Auth	**SetActive(bool active)**	Sets if the object is active across the network
Auth	**SetOwner(int clientId)**	Sets the owner of the object across the network
Both	**GetNetworkObject(NetworkId id)**	Returns a network object by its id
Both	**GetNetworkObjects()**	Returns all the network objects
Both	**GetNetworkObjectsOwnedBy(int clientId)**	Returns all the network objects owned by a client **(use -1 for server owned objects)**
Both	**GetNetworkObjectCount()**	Return the count of network objects
Client	**GetLocalClientIndex(int clientId)**	Returns the local client index for a given client id (normally 0). **Useful for split screen modes**

Auth side means that it can be used on both sides but need to have the ownership over the object

Packets and Owned Packets

The main (and only) data transfer unit across the network is packets, a data structure serialized/deserialized to/from byte arrays. Packets are very compact, blazing fast and reliable. You can define your own packets easily and the NetworkManager will register them automatically, ASSURING that a packet type will have the same id all across the network.

There are two main types of packets:

- **IPacket:** Default packet type, used by background systems and on actions not actually linked to a NetworkIdentity.

- **IOwnedPacket:** Based on IPacket, works the same, but actually carries an NetworkId used by the NetworkManager to link it to a certain NetworkIdentity, automatically calling its receiving callbacks.

Creating a Packet

You can easily create a packet type just by creating a class. For convention you shall use properties instead of fields, with the name in PascalCase:

```
public class DoorStatePacket : IOwnedPacket
{
    public NetworkId Id { get; set; }
    public bool IsOpen { get; set; }

    public void Serialize(BinaryWriter writer)
    {
        Id.Serialize(writer);
        writer.Write(IsOpen);
    }

    public void Deserialize(BinaryReader reader)
    {
        Id = NetworkId.Read(reader);
        IsOpen = reader.ReadBoolean();
    }
}
```

The Serialize method will tell the NetworkTransport how to translate into bytes and the Deserialize one will do the reverse way

YOU MAY DEFINE THE PACKETS EACH ONE ON ITS OWN SEPARATED CLASS FOR ORGANIZATION PURPOSES

Sending a Packet

To send a packet you can **use** any **one of the listed methods found in the NetworkBehaviour class that fits your use case**. Normally **you may use SendPacket** alongside **HasAuthority** checks. See the PlayerController example below:

```
1  private void Update()
2  {
3      if (!HasAuthority || !IsOwnedByClient)
4          return;
5
6      [...]
7
8      if (Input.GetMouseButtonDown(0) && punchCooldown <= 0 && IsGrounded)
9      {
10         [...]
11
12         //Just to add some dalay to fit the animation
13         Task.Run(async () =>
14         {
15             await Task.Delay(500);
16
17             SendPacket(new PlayerPunchActionPacket
18             {
19                 Id = Id
20             });
21         });
22     }
23
24     [...]
25 }
```

If the object is **guaranteed** to be always owned by the server you don't need to worry about broadcasting the packet to other players. But **normally** (as in the example above) **you shall handle the broadcasting process** in the **OnServerReceivePacket** method:

```
public override void OnServerReceivePacket(IOwnedPacket packet, int clientId)
{
    switch (packet)
    {
        [...]
        case PacketPlayerPunch punch:
            if(clientId == OwnerId)
                ServerBroadcastPacketExceptFor(punch, clientId, true);
            break;
        [...]
    }
}
```

Handling Received Packets

if the packet is an **IOwnedPacket**, the handling process is **automated**: the callback of the behaviors of the network object that owns the packet will be called. **For other packets** you may **use the NetworkBehaviour.GetPacketListener** to add/remove your own listeners **(you can also use this method to handle IOwnedPackets of other objects)**:

```
private void OnEnable()
{
    //Needs to listen to a specific packet type
    GetPacketListener<PlayerPunchActionPacket>().OnServerReceive += OnPlayerPunch;
}

private void OnDisable()
{
    GetPacketListener<PlayerPunchActionPacket>().OnServerReceive -= OnPlayerPunch;
}

private void OnPlayerPunch(PlayerPunchActionPacket obj, int client)
{
    [...]
}
```

Packet Reliability: States and Actions

There are basically two main types of packets: state packets and action packets. Don't get confused, both of them normally deals with an object state (as pretty much any method of any class), but there's an obvious difference between them:

State Packets	Action Packets
Constant Updated (Normally at a constant rate)	Updated only when needed (Actions/Commands/Etc...)
Not crucial, if a packet is lost the next one will just update the state with no problems	Crucial. If a packet is lost the game will differ between server and clients
Sent with reliable = false (Or omit the field, as false is it default value)	Sent with reliable = true, so the server will guarantee the order and the delivery of the packet
Limited to 1000 bytes	**No Bytes Limitation**

```
1 //NetworkTransport State (Non Realiable as it state updates constanntly)
2 SendPacket(CreateTransformPacket
3
4 //PlayerController Punch (Reliable as it's an action)
5 SendPacket(new PlayerPunchActionPacket
6     {
7         Id = Id
8     }, true);
```

Packing Packets

For optimization purposes packets are packed (...) together with others if they're sent at the same short-time period. Also if a reliable packet is too big to be sent on the same time, it will be automatically split.

Retroactive State Synchronization

When a client joins the server, the server synchronizes what network objects should be spawned (and destroyed), and then for each object it syncs the owner, the prefabId and default transformation (position, rotation and scale). All other custom data should be sent manually using the callback NetworkingBehaviour.OnClientConnect:

```
public override void OnClientConnected(int clientId)
{
    //Sends the player nickname to the client
    ServerSendPacket(new PacketPlayerData
    {
        Id = Id,
        Name = headplate.text
    }, clientId, true);
}
```

When the client joins, the server sends to the client all the information he needs about that PlayerController. The packet handling is the same as the default, so the only thing you should care about is: you need to send the current state for new players. After that the object will behave normally, the same as it works for all the clients previously connected.

Network Values

To make it easier to sync states we can use the NetworkValue class type, while offering change callbacks. You shall not use it for variables that update constantly, as value updating is a way more expensive than any standard packet transfer. See the table below for more info:

	Network Values	Packets
Synchronization	Automatic	Manual
Retroactive Sync	Automatic	Manual
Reliability	Reliable	Reliable / Non Reliable (User can choose)
Data Efficiency	Efficient	Very efficient
Speed	Fast	Very fast
Use-Cases	Values that don't update many times per second (Timers, points, simple states, etc...)	Values that update quite often (Position, complex states, etc...)

Implementing Network Values

To implement a network value is very simple. You create the field (serialized by unity) and just add it to the behaviour:

```
public class PlayerController : NetworkBehaviour
{
    public StringNetworkValue nickname = new StringNetworkValue("");
    public ColorNetworkValue bodyColor = new ColorNetworkValue(Color.white);

    public void OnEnable()
    {
        [...]
        WithValues(nickname, bodyColor);
        [...]
    }
}
```

DON'T CREATE THE FIELD USING NetworkValue our NetworkValue<T> as unity won't be able to serialize it correctly. Use a literal type as StringNetworkValue instead

```
public NetworkValue nickname = new StringNetworkValue("George"); //NO
public NetworkValue<string> nickname = new StringNetworkValue("George"); //NO
public StringNetworkValue nickname = new StringNetworkValue("George"); //TES
```

The field will show up on the inspector, where you can edit its default value (as any other field). For development purposes, if you change the field in the inspector, the value will be synced across the network as well (if you have authority on that object). All NetworkValues names will be yellow for a better distinction.

Listening For Network Value Changes

You can easily listen for changes on any network value, just adding a callback to it. The callbacks will be called on all the network ends:

```
public class Door : LogicOutput
{
    public BoolNetworkValue isOpen = new(false);

    private void OnEnable()
    {
        WithValues(isOpen);
        isOpen.OnValueChanged += OnIsOpenChanged;
    }

    private void OnIsOpenChanged(bool old, bool now)
    {
        open.SetActive(now);
        closed.SetActive(!now);
    }
}
```

Creating Your Own Network Value Types

By default, the system comes with the following types implemented:

- **ByteNetworkValue**

- **IntNetworkValue**
- **FloatNetworkValue**
- **DoubleNetworkValue**
- **BoolNetworkValue**
- **StringNetworkValue**
- **Vector2NetworkValue**
- **Vector3NetworkValue**
- **Vector4NetworkValue**
- **QuaterionNetworkValue**
- **ColorNetworkValue**
- **NetworkIdNetworkValue**

While all these types will probably support 99.999% of the use cases, some cases custom types will be required. But don’t worry, it’s very easy to create your own implementation. You just need to implement the serialization / deserialization of your object. Take for example the Vector2NetworkValue:

```
[Serializable]
public class Vector2NetworkValue : NetworkValue<Vector2>
{
    public Vector2NetworkValue(Vector2 defaultValue, ModifierType type =
ModifierType.OwnerOnly) : base(defaultValue, type) {}

    public override void Serialize(BinaryWriter writer)
    {
        writer.Write(_value.x);
        writer.Write(_value.y);
    }

    public override void Deserialize(BinaryReader reader)
    {
        var x = reader.ReadSingle();
        var y = reader.ReadSingle();
        SetValueCalling(new Vector2(x, y));
    }
}
```

Realtime Object Spawning

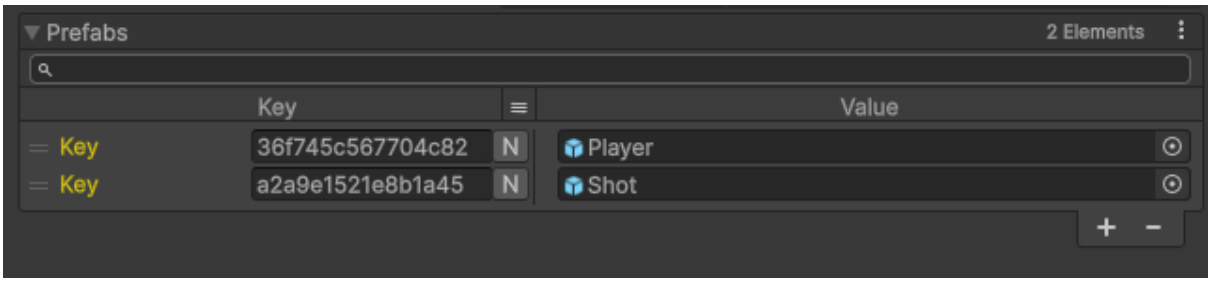
The spawning system is very simple. NetworkBehaviour class has the **Spawn static method** that you can use to spawn any prefab across the network. It will return the created object NetworkId and you can use it all around if needed

In the example below when the player presses T a bullet is spawned:

```
1 if (Input.GetKeyDown(KeyCode.T))
2     Spawn(shotPrefab, transform.position + body.forward * 1 + body.up * 1.5f,
  body.rotation, Vector3.one, true);
```

You can choose the initial object position, rotation, scale, if the object is active or not and the object's current owner. If the spawned object has no identity, the returned id and the object owner shall be ignored. By the default the owner is -1 (server)/.

REMEMBER TO REGISTER ALL THE PREFABS ON THE NETWORK MANAGER PREFAB SCRIPTABLE OBJECT, OR A ERROR WILL BE THROWN AS OTHER NETWORK ENDS WON'T BE ABLE TO KNOW WHAT PREFAB THEY SHOULD USE



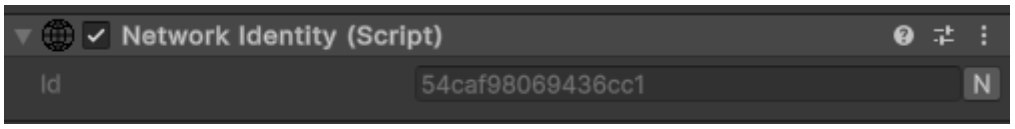
Miscellaneous

Split Screen Support

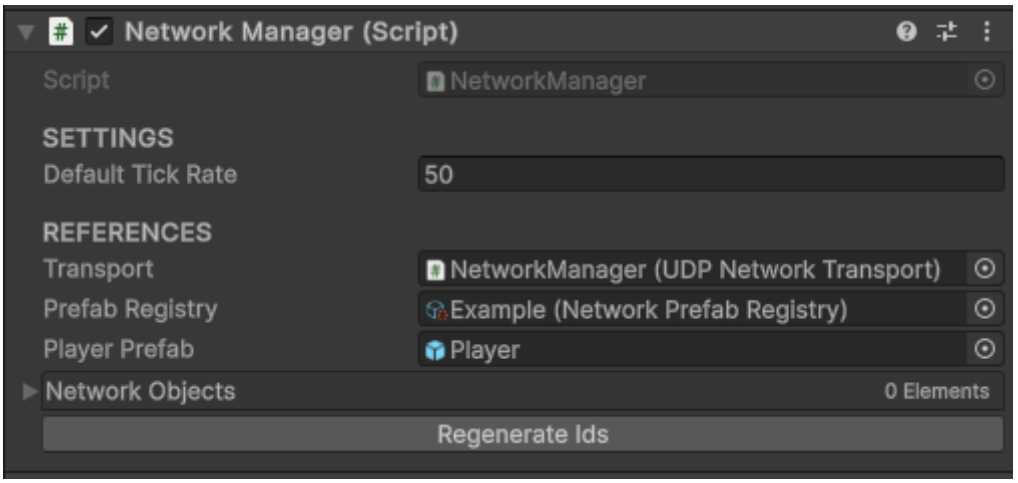
Any kind of multiplayer support can be added. Split screen/local multiplayer can be done creating a local-kind network transport or the **UDP** default transporter itself. Bluetooth and other connection types can also be considered

Regenerating Ids

Sometimes you may need to regenerate the id of an object. You can do this clicking on the **N** button on the id field **DO NOT DO THIS IN RUNTIME**



You can also regenerate all ids at once using the Regenerate Ids button on NetworkManager:



A BUILD GENERATED WITH DIFFERENT IDS WILL NOT WORK WITH A BUILD/EDITOR WITH NEW ONES