Survial Engine Online

Networking with NetcodePlus

Networking Framework

Survival Engine Online is built with NetcodePlus, a modified version of Unity's Netcode for GameObjects.

For documentation on the original Netcode, you can check here:
https://docs-multiplayer.unity3d.com/netcode/current/about/index.html

Instead of using **NetworkObject** and **NetworkBehaviour** components, this assets uses **SNetworkObject** and **SNetworkBehaviour**. It is still possible to use the standard NetworkObject component on a separate prefab object, but do not mix SNetwork… and Network…on the same object, it should either be one or the other. Most Survival Engine scripts (Selectable, Destructible, etc.) uses SNetworkObject.

What has changed in NetcodePlus?
-Components (SNetworkObject, SNetworkBehaviour)
-Spawn System (Allowing more networked objects in the scene, and automatic spawn/despawn)
-Network Action System (replacing NetworkVariable and RPC calls)

What has NOT changed?
-Transport Layer (Unity Transport)
-Scene Management
-Messaging & Serialization

Why?

While Network for Gameobject is a powerful framework. It becomes quickly limited when huge scenes contain hundreds of networked objects. The default Netcode spawning system automatically sync ALL the NetworkObjects in the scene whenever a client connects, and netcode does not have an easy way to manually manage what is synced. While it would have been possible to just Instantiate and Destroy objects. This would not allow an object to run on the server only (for example an animal walking far away from the players, or a plant growing).

For this reason I had to create a new spawn system. By adding the SNetworkOptimizer component to a prefab, the game will automatically sync objects that are near the players, and stop syncing the ones that are far (you can set the distance). Which greatly improve performances.

The new Network Action system is written in pure C# (no [tags] and no code generated at compilation, like it is the case with RPC calls). This allows to have a bit more flexibility and control on how and when actions are triggered, and also let you know exactly how the code works since there is no hidden code generated. It streamlines function calls and syncing variables, since both are done the same way. Another advantage is that you can call a function on both the client and server in one call without extra code. I may add support for NetworkVariables and RPC calls in the future, but at the moment you should use the Network Action system since the same thing can be achieved, just in a different way.

Network Scripts

Here are the main networking scripts used by NetcodePlus and SurvivalEngine

Components

SNetworkObject.cs

Add as a component to any object that needs to be synced on clients. Any object with a SNetworkBehaviour script should have this component. Call the function Spawn() to start syncing the object on clients, and call the function Despawn() to stop it.

Each SNetworkObject (prefab or scene object) needs to have a unique id that will be generated automatically (you can also run Netcode->GenerateNetworkID to do it manually).

Each NetworkObject is assigned an owner, by default with Spawn() it's the host (server), but the owner can be chosen if instead Spawn(owner_id) is called with a parameter. For example each player character is owned by that player.

SNetworkBehaviour.cs

Do not add directly as a component, instead use as base class of your own component scripts (instead of MonoBehaviour).
This will allow to use network features (such as actions) in the script.

SNetworkOptimizer.cs

Add as a component, it will automatically call Spawn() and Despawn() based on the distance to players, instead of having to call it manually. You can also choose to disable all scripts when despawned.

Managers

TheNetwork.cs

Main networking script, it handles connecting clients to host, registering prefabs, and has basics functions to know if you're on a server or client, and the ID of each client. This object is set to DontDestroyOnLoad so it persists between scenes. Should be in both your menu and game scenes.

TheNetworkSurvival.cs

The only manager that is specific to Survival Engine (and not NetcodePlus). Will handle transferring the save file and selects the spawning position of each player when connecting. Should be on the same persistent manager object as TheNetwork.

NetworkGame.cs

Unlike TheNetwork, this don't persist between scenes. Should only be added in game scenes (not menu scenes). It registers network messages specific to the gameplay. Its part of the Manager prefabs.

Lobby and Dedicated Server

ClientLobby.cs ServerLobby.cs
Managers for the client or server side of the lobby. Lobby uses Web Requests instead of connecting with Netcode. (WebTool.cs manages sending web requests).

ServerGame.cs
Main script for the dedicated server scene. It can receive command line parameters when started, such as the scene to load, game id and port. And it will start the game server based on parameters received.

Network Messages

Allows you to send simple messages over the network. For more complex actions (like calling a functions that should be executed on all instances, or messages that need to be forwarded to other clients by the server), you can use Network Actions instead (see the next section).

NetworkMessages are one-way messages, but don't require to use SNetworkBehaviour or SNetworkObject so they are great for sending messages inside game managers, or things that are not necessarily linked to a spawned SNetworkObject, like sharing a save file.

All functions related to network messages are in NetworkMessaging.cs, and can be accessed with TheNetwork.Get().Messaging

Listen for a message

```
ListenMsg(type, callback);
```

type: action identifier (string). Each message should have a unique type across the project.

callback: the function that will be called when this type of message is received. First parameters is the client_id of the sender (ulong), and second parameter is the data, either a FastBufferReader or SerializedData.

Sending a message

```
SendEmpty(type, target, delivery);
SendBytes(type, target, data, delivery);
SendString(type, target, data, delivery);
SendObject(type, target, data, delivery);
```

And more…

Type is the same than ListenMsg, target is the client_id to send to (you can also call SendStringAll, SendBytesAll.. from the server, to send to all clients). Delivery is NetworkDelivery.Reliable or NetworkDelivery.Unreliable (but faster).

Network Actions

Network Actions allow you to sync variables or to call functions over the network. To use network actions, you must have a script that inherits from SNetworkBehaviour. Then, you need to declare:

```
private NetworkActionHandler actions;
```

Inside the OnSpawn() function, you can register your actions, make sure to call Clear() on the handler inside OnDespawn().

Register an action

```
Register(type, callback, delivery, target)
```

type: action identifier, can be either a ushort, string or enum, when sent over the network it will automatically be converted to a ushort. Same type can be used in different SNetworkBehaviour, but within the same behaviour, each action should have a different type.

callback: the function that will be called on the target.

delivery: is either Reliable (slow but guaranteed) or Unreliable (fast).

target: The target is who should run the callback function, either the Server, the Clients, or both.

Trigger an action

```
Trigger(type, ...params...);
```

or

```
Refresh(type, ...params...);
```

If a client (or server) triggers an action that targets itself, it will run it locally, if it doesn't target itself, it will send the action through the network. If it targets All it will do both in one call.

Refresh does the same than Trigger, but Refresh is set to Clients target by default instead of All.

If a client with authority triggers an action that target Clients, the server will receive and then forward the action to all clients. The server will ignore triggers that are sent from clients without authority (if they aren't the owner of the network object). But actions can still be triggered locally when not the owner. The server has authority on everything.

Action Example

```
protected override void OnSpawn()
{
        //Define all the actions here
        actions = new NetworkActionHandler(this);
        actions.RegisterVector(ActionType.OrderMove, DoMoveTo,
                NetworkDelivery.Reliable, NetworkActionTarget.All);
}

protected override void OnDespawn()
{
        //Clear all the actions
        actions.Clear();
}

public void MoveTo(Vector3 pos)
{
        //This function can be called from either the client or the server
        actions.Trigger(ActionType.OrderMove, pos);
}

private void DoMoveTo(Vector3 pos)
{
      //Code here will be executed on both the client and server, since the target is All
}
```

Since the target is set to ALL and the server has authority on this NetworkObject (this is decided when Spawn() is called, not from this script), the following will happen:

If a client calls MoveTo(), the function DoMoveTo() will only be executed locally (since it target clients and server will ignore triggers from non-owner).

If a server calls MoveTo(), the function DoMoveTo() will be executed locally on the server, AND then sent to all clients to be executed there too. Because the server has authority to send the action, and because the target is set to all.

Variable Refresh Example

```
private CharacterState sync_state = new CharacterState();

protected override void OnSpawn()
{
        //Define all the actions here
        actions = new NetworkActionHandler(this);
        actions.RegisterRefresh(ActionType.SyncObject, ReceiveSync,
                NetworkDelivery.Unreliable);  //Refresh target is always Client only
}

protected override void OnDespawn()
{
        //Clear the actions
        actions.Clear();
}

public void RefreshState()
{
        //Refresh must be called from the server (will be ignored otherwise)
        actions.Refresh(ActionType.SyncObject, sync_state);
}

private void ReceiveSync(SerializedData sdata)
{
        //Read and store the variable on the clients
         CharacterState state = sdata.Get<CharacterState>();
         sync_state = state;
}
```

A Refresh works in the exact same way than an action, except that it will be ignored if triggered from a client, and the target is always the clients (so its always Server sending to Clients). The same can be achieved with a regular action, this is just a shortcut.

So this:

```
actions.RegisterRefresh(ActionType.SyncObject, ReceiveSync, NetworkDelivery.Unreliable);
actions.Refresh(ActionType.SyncObject, sync_state);
```

Is the same than this:

```
actions.RegisterSerializable(ActionType.SyncObject, ReceiveSync, NetworkDelivery.Unreliable,
                             NetworkActionTarget.Client);
if (IsServer)
        actions.Trigger(ActionType.SyncObject, sync_state);
```

Need more Doc?

Looking at the existing code should be the easiest way to learn how to use the action system, as long as you understand Register() and Trigger(), that is most of what you need to know.

If you need some good examples that uses actions, take a look at Character.cs or any networked player script: (PlayerCharacter.cs, PlayerCharacterInventory.cs, PlayerCharacterCraft.cs).

You can also ask questions on Discord: https://discord.gg/JpVwUgG