Tutorial – Unity Networking

This standalone tutorial will take you through the basics of networking in Unity.

Get the NetworkShooter project from Resources.

It's a very simple two-player "couch co-op" game, with two players in the scene who can move around and shoot at each other to subtract hit points from each other using keyboard controls.

During this tutorial we'll look at what's involved in making this into a multiplayer networked game.

Overview of the game

The Player prefabs have three scripts on them.

CharacterMovement

Moves the capsule around. Note the index variable , set to 1 and 2 for each player. This determines which axis they use to move around. Extra axes have been set up in the Input settings called "Horizontal1", "Horizontal2", "Vertical1" and Vertical2" (Edit->Project Settings->Input to view).

These use WASD for index of 1, and arrow keys for index of 2.

LaserBeam

The player has a LineRenderer and ParticleSystem attached to its nose.

It also has an index, which it gets from the CharacterMovement. When the correct Fire button is pressed ("Fire1" or "Fire2", mapped to left and right CTRL keys) the character emits particles and turns on their LineRenderer for half a second. They do a Raycast to see if they hit any Health scripts (ie the other player), and subtract health from them if they do. The Laser has a short cooldown so it can only be fired once per second.

Health

Stores the character's hit points. It also communicates with the HealthBarManager attached to a world space canvas to pop a little health bar above each character's head by instantiating a prefab.

The SinglePlayer scene is set up with two SinglePlayer prefabs with indexes 1 and 2. They're lined up to shoot each other, so press either CTRL key and see them subtract from each other's health with a visible laser beam. Move them around with WASD or arrow keys.

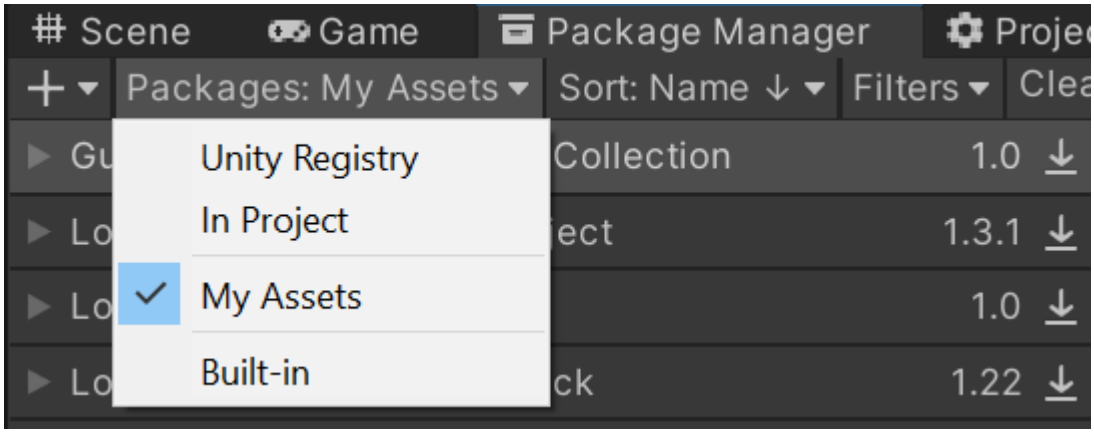
The HealthBar scripts are in a separate folder, and can be treated as a bit of a black box for this exercise.

Setting up a Network Game using Mirror

Mirror is a community fork of the old UNet library for networking that was built into Unity. It's been supported by the devs behind uMMORPG, and is robust and suitable for our purposes.

First, get Mirror for free from the Assets Store.

(Open the Unity Asset Store via a web browser, find Mirror, sign in to your account and add to your Assets, and then sign into unity with the same account and find it in the Unity Package Manager window under Packages-> 'My Assets')



We'll then tackle upgrading the NetworkShooter game in steps.

First of all, we'll set the game up to use the Mirror Networking library, and get the players moving around on the network.

- Make an empty GameObject called *_Networking* and add a NetworkManager and NetworkManagerHUD component to it. (The NetworkManager will also create a KcpTransport object that's required for it to work.)

In the CharacterMovement script, we'll derive it from NetworkBehavior instead of MonoBehaviour. To do this we'll have to add a using statement.

```
using UnityEngine;
using System.Collections;
using Mirror;

// a basic character movement script, for moving a character around in multiplayer
with a Network transform
public class CharacterMovement : NetworkBehaviour {
```

In a two player game we're going to have two player characters, but only want one of ours to read the keyboard. In our Update add the following lines so the non-local player doesn't read our keyboard.

```
// Update is called once per frame
       void Update () {
        // other player's characters still get an update - don't try to move them,
        // just you!
        if (!isLocalPlayer)
            return;
```

Go to the Player prefab and add a **NetworkIdentity** script, as this is required by NetworkBehaviour now.

Add a **NetworkTransform** to the Player prefab, which is responsible for keeping the Transform synched across the network. Tick the Client Authority box, so that each client is able to move their player object around.

Delete the Player objects from the scene.

We'll let the NetworkManager create a Player for each player when they log in. To do this, go back to the Networking Object in the scene and find the **Player Object** settings in the **NetworkManager**. Drag in the Player prefab to the **Player Prefab** setting.

We can now run the game. First off, run it in the editor, and press the first option "Host (Server + Client)" button in the UI to make sure your player still responds to WASD and left CTRL.

To test the multiplayer aspect of it, make a build using File -> Build Settings. Add the Open Scene and Build and Run. You may want to change the Resolution to Windowed in **Player Settings** first, as this makes the build a little easier to work with.

Run in the editor too, setting one as “Host (Server + Client)” and the other as “Client” with “localhost” in the text field, and you should get two player appearing in both games. Switch between them and you can control the local player on both screens and see them update on the other one.

At this point the two players should move around correctly like you’d expect from a multiplayer game. When you press fire though, both characters fire in unison. We need to fix that.

Review of the Network system and authority

We’re leaning on the **NetworkTransform** class to move the players around. When one of them moves, it sends a message across the network with the new position and rotation, and the Network Identity’s unique internal ID, just like you did manually in the C++ exercises.

However, not just anything can be moved, the host or client has to have authority to move the object. By default, the host is authoritative. When we tick **Client Authority** in the NetworkTransform, we’re essentially giving the client special dispensation to move their character and have the host allow this and respond to client messages that their player has moved.

When we fire the laser, we can’t use a NetworkTransform, because we’re turning objects on and off, firing particles and doing a raycast to modify health. We’ll do this by allowing the client to call functions on the host and the host to call functions on the client.

Before we do that, we’ll look at adding a simple bit of functionality to extend our networking.

Moving markers around on client and host

Let’s add some simple functionality where each player has a marker in the scene, and when they click on the ground it moves their marker.

We’ll add a couple of small spheres to the scene to act as markers for the two players. (You might use these markers to say swarm AI troops to in a networked strategy game.)

Add a sphere, remove the SphereCollider component from it, and drop the red material on it. Rename it ClientMarker. Add a NetworkTransform (and get a NetworkIdentity, which once again needs Local Player Authority ticked, allowing the Host to move it for all clients.)

Copy this and call it ServerMarker and colour it blue. We'll now add a script to associate each player with the correct mark depending on whether they're the client or host. Add a script called MarkerMoverNetwork to the _Networking prefab and fill it in like this.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;


// this illustrates the spawned player prefab taking control of an object outside
of it via Commands
// the return message (host to client) is handled via a NetworkTransform
automatically
public class MarkerMoverNetwork : NetworkBehaviour {

    public Transform marker;

    // Use this for initialization
    void Start () {
        // the scene has a couple of marker points - red for the client, blue for
the server.
        // mouse clicks place the marker point for your marker. We need each
character to point to
        // the correct marker on each machine, and dye each player to match their
marker

        // isServer - true if you'r the host instance of the game
        // isLocalPlayer - true if you're in control of this CharacterMovement

        if (isServer != isLocalPlayer)
        {
            // you're the client and this is you, or you're the host and this is
not you,
            // therefore its the client character!
            // TODO use better logic to accomodate 3+ players, like
            //    id = GetComponent<NetworkIdentity>().playerControllerId;
            marker = GameObject.Find("ClientMarker").transform;
            SetColor(Color.red);
            gameObject.name = "Red Player";
        }
        else
        {
            marker = GameObject.Find("ServerMarker").transform;
            SetColor(Color.blue);
            gameObject.name = "Blue Player";
        }
    }

    void SetColor(Color col)
    {
        GetComponent<Renderer>().material.color = col;
    }

…
```

We're using some other members of NetworkBehavior to work out which player represents the host and which represents the client. We're colouring the player's bodies to match the colours of the markers because it helps to see these things clearly when testing a network game.

Remember that both the host and client programme have two of these components in their scene.

isServer is consistent between both objects in the same instance of the game, that's true if the player is the host. isLocalPlayer is true for just one of the components, as the comments explain.

We now need to put in code so that when the local player clicks the mouse, it positions their marker at that point.

```
…

    // Update is called once per frame
    void Update () {
        // other player's characters still get an update - don't try to move them,
just you!
        if (!isLocalPlayer)
            return;

        // click to move a marker
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out hit))
            {
                SetMarker(hit.point);
            }
        }
    }

    void SetMarker(Vector3 pos)
    {
        marker.position = pos;
    }
}
```

Note that once again we're screening out any input for the other player's character using isLocalPlayer.

Try this code and see what happens.

When the host clicks their object, it gets moved on the host, and its updated position is sent to the client via the NetworkTransform. All good.

When the client clicks though, they move the object locally and it doesn't transmit because the client doesn't have the authority to move their marker. (It's in the scene, therefore a host-owned object)

We need to make SetMarker into a Command. A Command is a function that clients can call, and have them execute on the server. The server has authority to move the marker for them.

```
// Update is called once per frame
    void Update () {
        // other player's characters still get an update - don't try to move them,
just you!
        if (!isLocalPlayer)
            return;

        // click to move a marker
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out hit))
            {
                CmdSetMarker(hit.point);
            }
        }
    }

    [Command]
    void CmdSetMarker(Vector3 pos)
    {
        marker.position = pos;
    }
```

We've added the Command attribute to the SetMarker function, and according to Unity's naming convention, we have to append Cmd to the start of the function name.

Now when the client clicks on the ground their marker will move, because the following happens.

- The client calls the CmdSetMarker function. It has authority to do this because the component sits on the client's player prefab.
- The host executes CmdSetMarker in response to the command, and moves the marker
- The NetworkTransform updates the position on all clients.

Getting the LaserBeam to work across the network

The LaserBeam script does two things which we need to consider differently in terms of networking. It fires off some visual effects, the particles and the LineRenderer. We should let each client take care of that by itself as these are cosmetic effects with no consequence.

It also does a raycast that subtracts health off other players. We definitely don't want this happening on both the host and client, so for now we'll do this only on the host and inform each player of their updated health value.

First off we want to set up the class to work across the network, so we derive that class from NetworkBehavior like we did before.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Mirror;

public class LaserBeam : NetworkBehaviour {

    public LineRenderer lineRenderer;
    public float coolDown;
```

In the update, as before, we'll screen out any input for non-local players. We'll also change Fire() into a command so that clients can trigger it on the host.

```
void Update()
    {
        // count down, and hide the laser after half a second
        if (coolDown > 0)
        {
            coolDown -= Time.deltaTime;
            if (coolDown < 0.5f)
                ShowLaser(false);
        }

        // only check controls if we're the local player
        if (!isLocalPlayer)
            return;

        if (Input.GetButtonDown("Fire" + index) && coolDown <= 0)
            CmdFire();
    }

    [Command]
    void CmdFire()
    {
```

We now need to think about what we want to happen on each machine when we press fire. It should go like this.

- Client or Host press the Button, triggering CmdFire on the host
- CmdFire on the host does a raycast and subtracts health from any player who gets hits
- The Host tells each client to display the particles and turn on their LineRenderer.
- The particles then do their own thing, and the Linerenderer counts down and turns off on each client independently.

The third step here introduces the idea of a RPCClient call (RPC = Remote Procedure Call). This is a function that the host can call to be executed on every connected client.

We'll split the Fire() function up to have a DoLaser() function where we do the visual effects, and if we're the server, we do the raycast as well and take away health.

```
void DoLaser()
    {
        // trigger the visuals - this should happen on all machines individually
        ShowLaser(true);
        coolDown = 1.0f;
        // more visual fx, a burst around the firing nozzle
        if (fireFX)
            fireFX.Play();

        // do a raycast to subtract health. We only want to do this on the server
rather than each client doing their own raycast
        if (!isServer)
            return;

        RaycastHit hit;
        if (Physics.Raycast(new Ray(transform.position, transform.forward), out
hit, 10.0f))
        {
            Health health = hit.transform.GetComponent<Health>();
            if (health)
            {
                health.health -= 10;
            }
        }
    }
```

CmdFire will now call this locally and call a new RPCClient call that triggers the particles on each client.

```
    [Command]
    void CmdFire()
    {
        // tell all clients to do it too
        RpcFire();
    }

    [ClientRpc]
    void RpcFire()
    {
        DoLaser();
    }
```

Now run the game again.

You'll see that the Laser effects turn on and off at the right times. The health is only being subtracted on the host for now though.

Synching Variables across the network with SyncVars

It would be great if we could get the health values that are modified on the server to automatically synch across the network in the way that NetworkTransforms do. It turns out that it's pretty easy to do that.

Change Health to a NetworkBehavior, and put a SyncVar attribute on the health variable.

```
using Mirror;

// Basic Hitpoint class demonstrating a SyncVar with a hook
public class Health : NetworkBehaviour {

    [SyncVar]
    public float health = 100;
    public float maxHealth = 100;
```

This means that any changes on the Host (or on a client with authority) will be propagated to all other connected machines, in the same way that transforms get synched in a NetworkTransform.

Adding a hook function to a SyncVar

You can set up a function to be called locally whenever a SyncVar value changes.

This acts as a convenient replacement to a Command and RpcClient pair, as the function will get called locally on all parties when either the server or a client with authority updates the variable.

Add a declaration of this "hook" function like so:

```
    [SyncVar(hook = "onHealthChanged")]
```

And then declare a function with the appropriate signature to get called by C# reflection when the health value is changed. You have the option to reject the newValue that's being passed in hereby setting it back to the oldValue if you like. The SyncVar itself is automatically updated before entering this function.

I'll leave it to you to add the blood splat particle effect back in here.

```
public void onHealthChanged(System.Single oldValue, System.Single newValue)
    {

    }
```