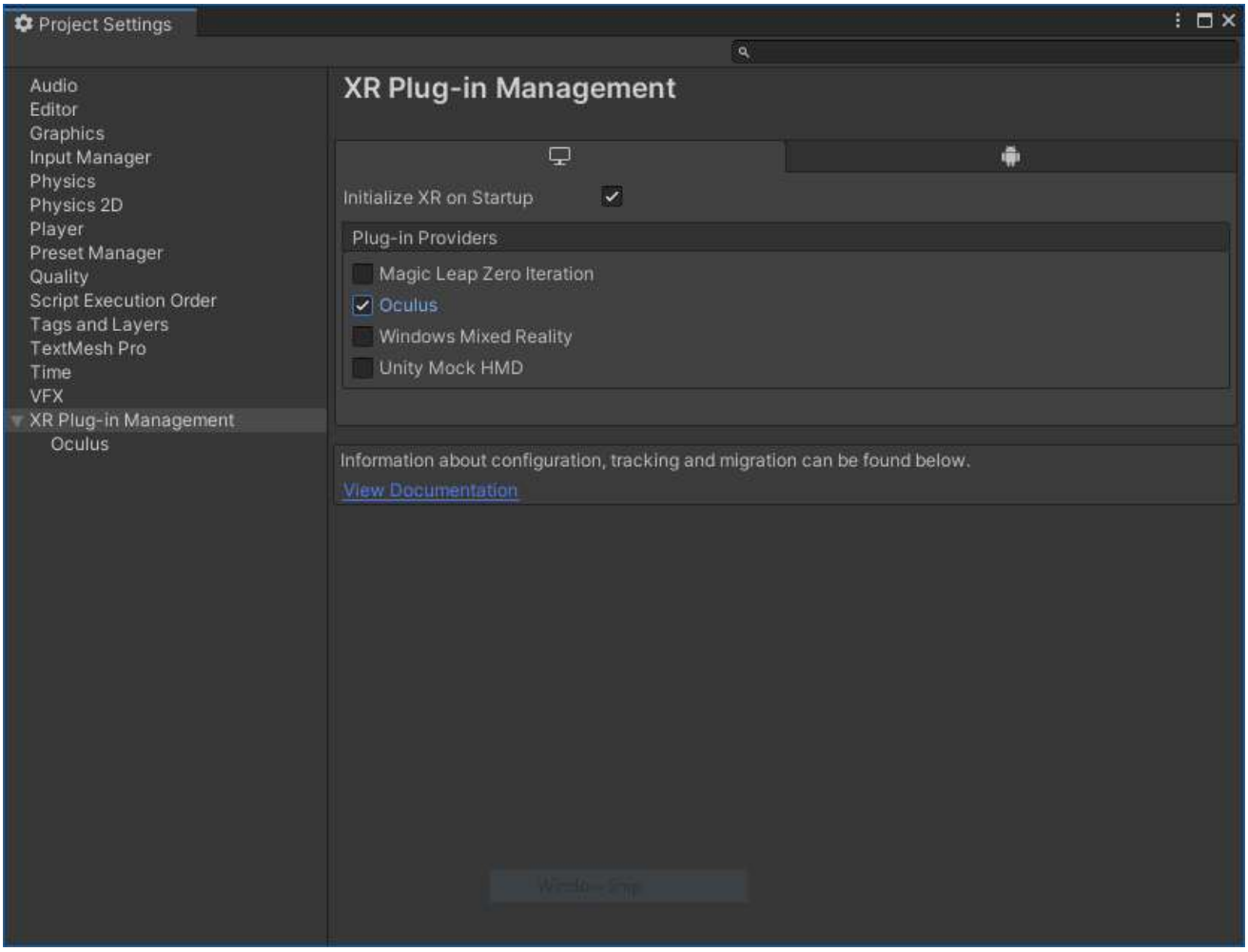
Setting Up Ubik for VR

Ubiq uses Unity's XR Plug-in Management system for all samples. This allows you to enable a VR plugin matching your device and to interact with it through a unified interface. There is a quick guide below for each supported platform.

Oculus or Windows Mixed Reality (desktop)

1. In Unity, open the project settings window (Edit/Project Settings/...) and go to the XR Plug-in Management menu.
2. Enable the plug-in here - just tick either the Oculus or Windows Mixed Reality box. Also tick the box for "Initialize XR on Startup".



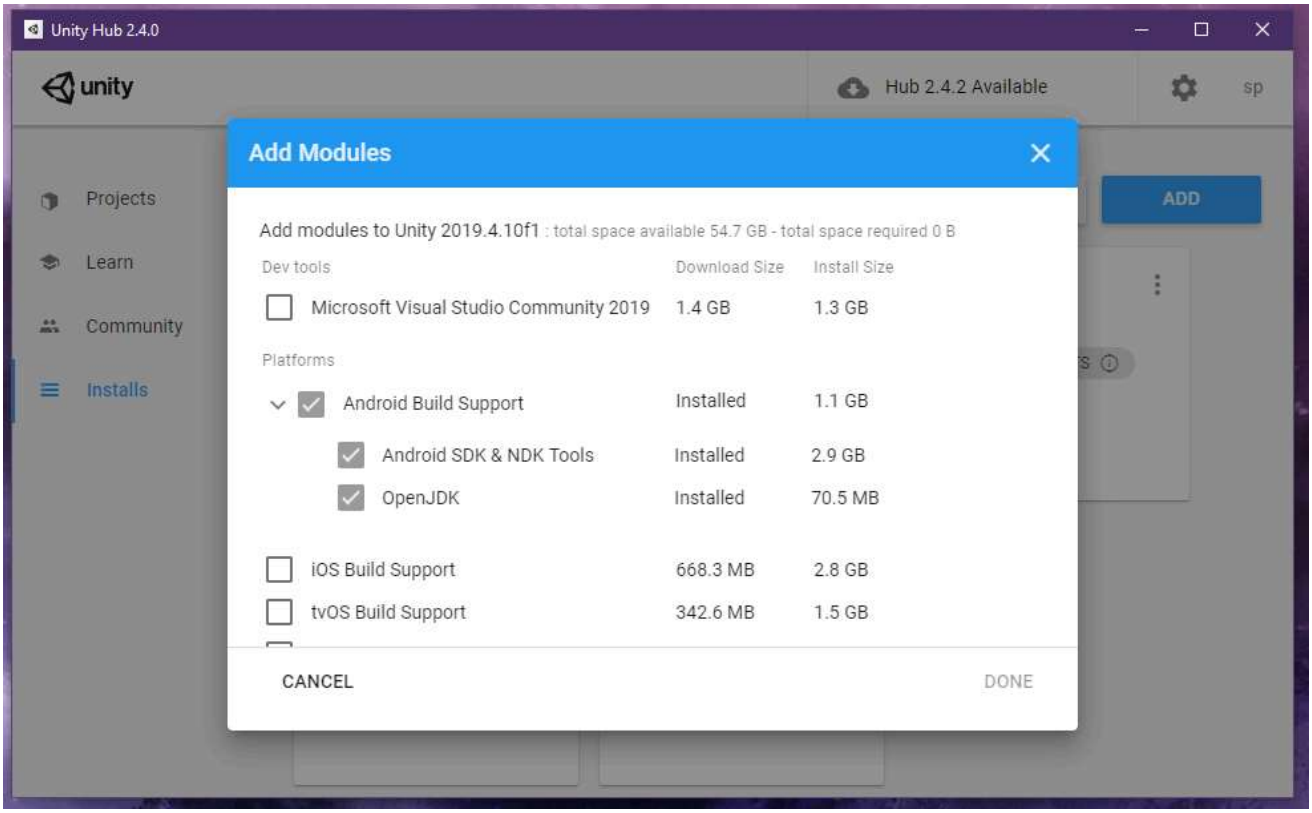
OpenVR (desktop)

1. Note that while rendering and tracking works well, this subsystem is currently missing input from the hand controllers. Unfortunately, this is a limitation with the plugin and a fix does not seem to be on the horizon.

2. Follow the instructions to download the OpenVR Unity XR plug-in here: https://github.com/ValveSoftware/unity-xr-plugin/releases/tag/installer

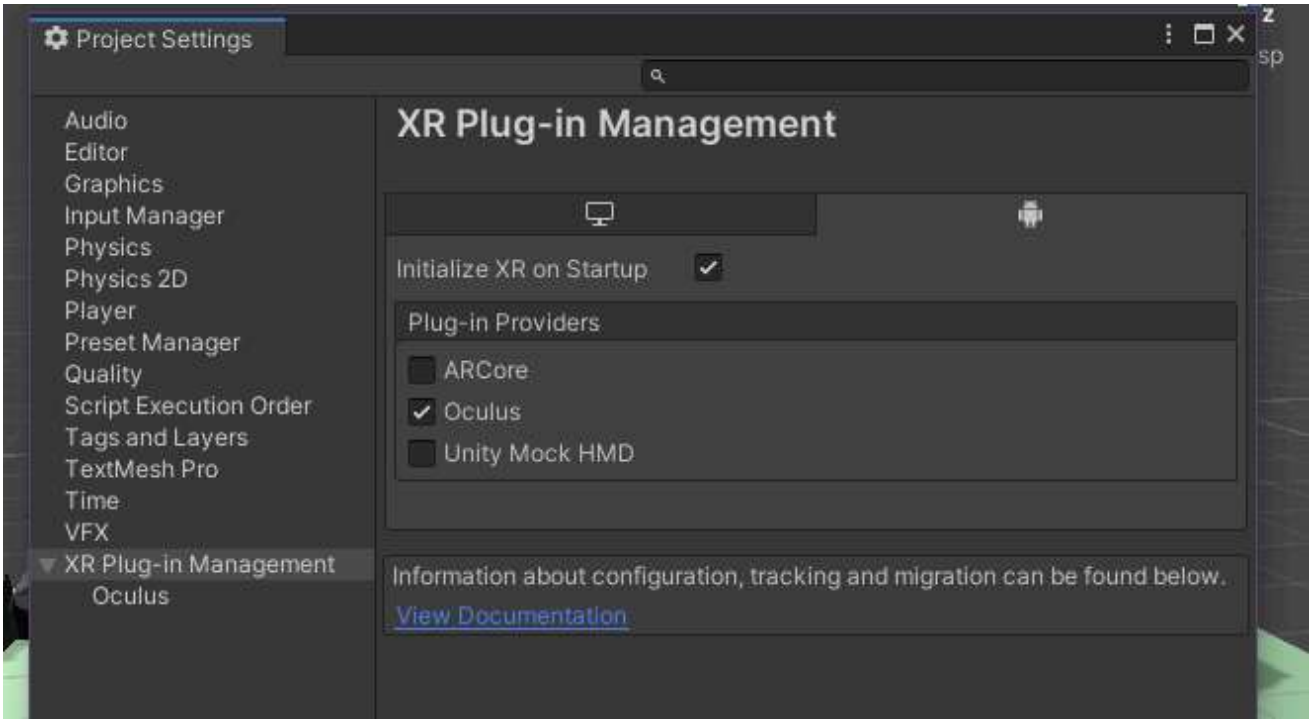
Oculus Quest

1. Install android build tools. In Unity Hub, click `Installs` on the left-hand menu. Click the three dots in the top right corner of the box for your Unity installation and select `Add modules` from the dropdown. Select `Android Build Support` and both subsequent options (`Android SDK & NDK Tools` and `OpenJDK`). Wait for installation to complete, then re-open your project.



2. In Unity, open the project settings window (Edit/Project Settings/...) and go to the XR Plug-in Management menu.

3. Click the Android tab and check the boxes for Oculus and "Initialize XR on Startup".



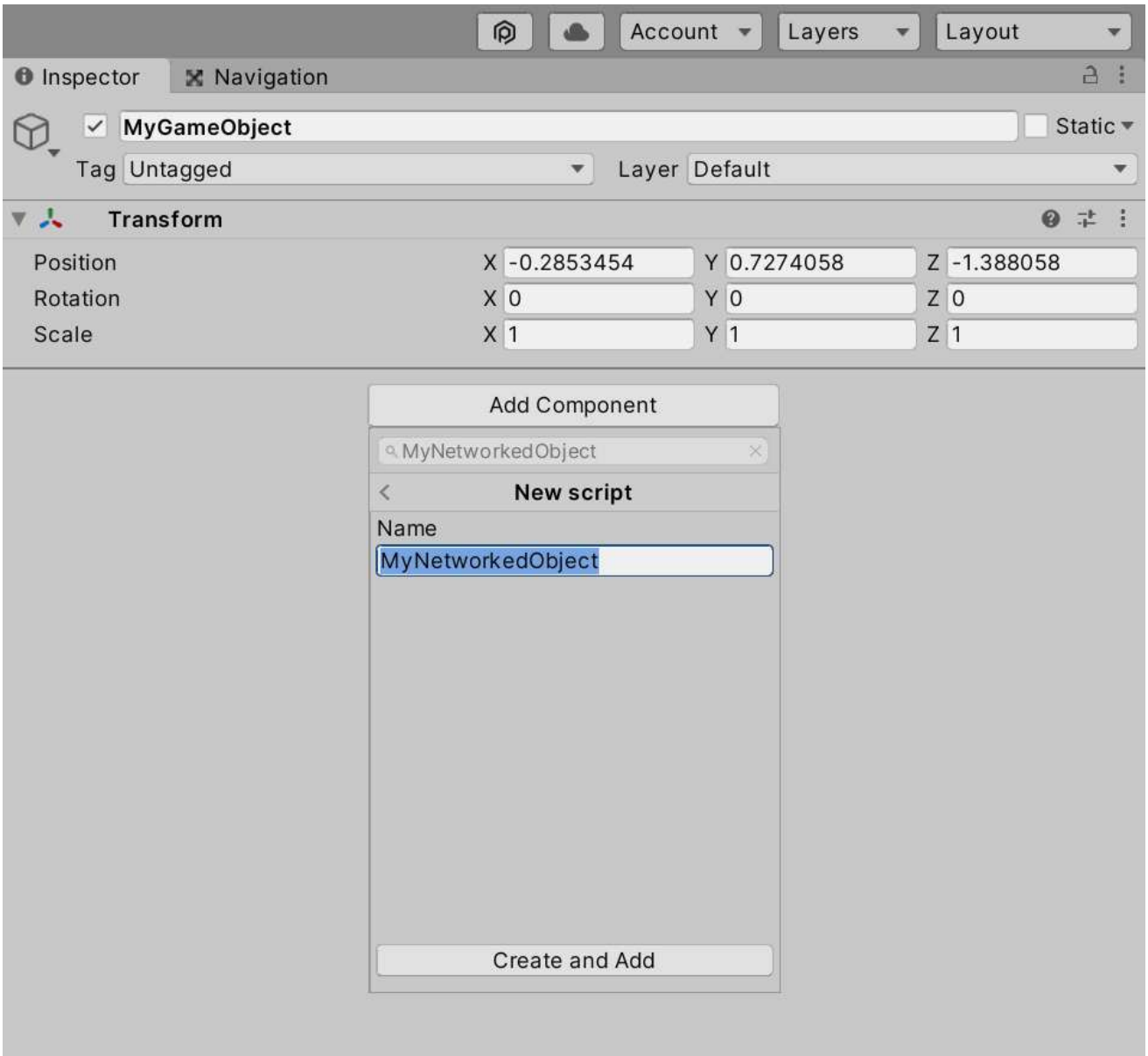
If your Quest is connected with Oculus Link, you can now test your applications by entering Play Mode in the Unity Editor.

4. (Optional) To create your first build for the Quest, follow the Oculus `Enable Device for Development and Testing` guide: https://developer.oculus.com/documentation/unity/unity-enable-device/

Building a Basic Networked Object

Networked objects are Components that can keep themselves synchronised by exchanging messages over the network. You can create new Networked Objects to implement your own networked behaviour.

1) Create a new Unity Script and add it to the GameObject that you want to be networked. You can do this via the inspector by clicking on "Add Component" and typing the new name.



2) Include Ubiq.Messaging

```
    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;
    using Ubiq.Messaging;

    public class MyNetworkedObject : MonoBehaviour
    {
        // Start is called before the first frame update
        void Start()
        {
        }

        // Update is called once per frame
        void Update()
        {

        }
    }
```

3) Create a new member, `context`. `context` will hold the address of your object on the network, and allow you to send messages.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

4) Declare a method called `ProcessMessage`, which takes a `ReferenceCountedSceneGraphMessage`. This is where messages to your Component will come in.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {

    }
}
```

5) In your `Start()` method, call `NetworkScene.Register()`. This registers your Component with Ubiq and gets it an address on the network. The return value is a `NetworkContext` which you can store in the member created previously.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {
        context = NetworkScene.Register(this);
    }

    // Update is called once per frame
    void Update()
    {

    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {

    }
}
```

6) Define what a message between instances of your Component will look like. In the message, write the variables that you want to send. Below we create a message to send the object's position.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {
        context = NetworkScene.Register(this);
    }

    // Update is called once per frame
    void Update()
    {

    }

    private struct Message
    {
        public Vector3 position;
    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {

    }
}
```

7) Add code to parse and process incoming messages to `ProcessMessage`. Below, we convert the ReferenceCountedSceneGraphMessage into a Message, and then access the position member to set the object's position in world space.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {
        context = NetworkScene.Register(this);
    }

    // Update is called once per frame
    void Update()
    {

    }

    private struct Message
    {
        public Vector3 position;
    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        // Parse the message
        var m = message.FromJson<Message>();

        // Use the message to update the Component
        transform.localPosition = m.position;
    }
}
```

8) Messages will only be sent to your Component, from other instances of your Component, so you also need to Send messages as well. This is done through the NetworkContext you recieved when the Component was registered.

Below, we check if the position of the object has changed in the last frame, and if so, send the new position to all other instances of the object. We detect if the position has changed by keeping track of the position in the last frame in a new member, `lastPosition`.

We also modify `ProcessMessage` slightly, to update `lastPosition` when a message is received - otherwise, an incoming message will generate an outgoing message, and two Components will send messages back and forth in an endless cycle even if the player hasn't changed the objects position!

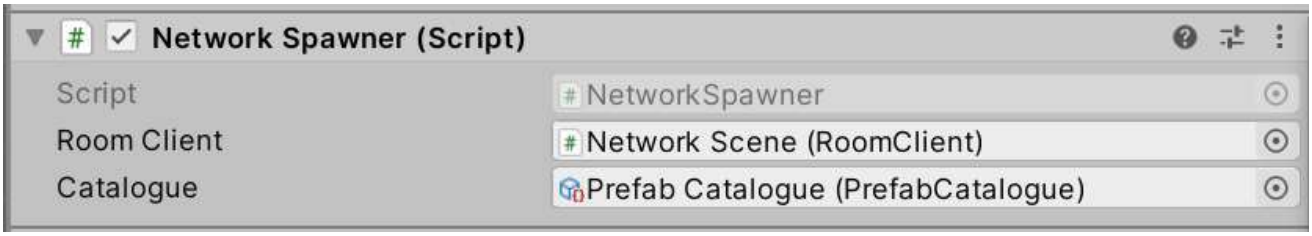
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;

public class MyNetworkedObject : MonoBehaviour
{
    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {
        context = NetworkScene.Register(this);
    }

    Vector3 lastPosition;

    // Update is called once per frame
    void Update()
    {
        if(lastPosition != transform.localPosition)
        {
            lastPosition = transform.localPosition;
            context.SendJson(new Message()
            {
                position = transform.localPosition
            });
        }
    }

    private struct Message
    {
        public Vector3 position;
    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        // Parse the message
        var m = message.FromJson<Message>();

        // Use the message to update the Component
        transform.localPosition = m.position;

        // Make sure the logic in Update doesn't trigger as a result of this message
        lastPosition = transform.localPosition;
    }
}
```

9) Your first networked object is now complete!

Add a cube to your object so you can see it in the scene. Continue with the tutorials to see it in action!

Spawning Objects

You can create objects at runtime on all Peers using the `NetworkSpawner`.



Objects are spawned by calling `SpawnWithPeerScope()` or `SpawnWithRoomScope()`. The `NetworkSpawner` is static and so is accessible anywhere in the code. However, it needs a Unity `GameObject` to find the `NetworkScene` in which to spawn the object.

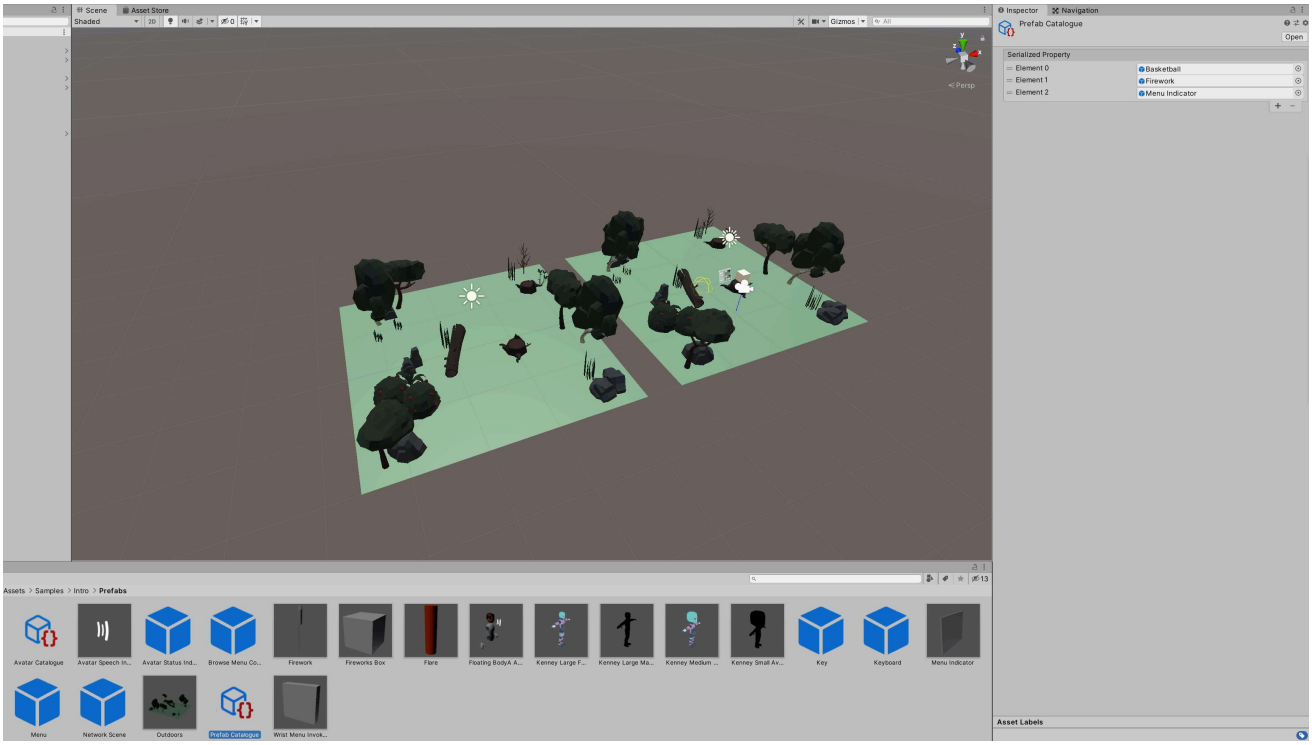
There are two Spawn methods: `SpawnWithPeerScope` and `SpawnWithRoomScope`.

- `SpawnWithPeerScope` keeps objects around so long as the Peer that spawned them is in the room.
- `SpawnWithRoomScope` keeps objects around so long as the room exists, regardless of who created them.

Rooms only exist as long as they have at least one member in them. Once all Peers leave a Room, the Room is destroyed along with all the objects within it.

Before a GameObject can be Spawned, it must be added to the `PrefabCatalogue` of the `SceneManager`.

Do this by adding a new entry to the `Prefab Catalogue` Asset in the Inspector.



Once the Prefab GameObject has been added it can be spawned by passing the same reference to `SpawnWithPeerScope` or `SpawnWithRoomScope`:

```
public void Use(Hand controller)
{
    NetworkSpawnManager.Find(this).SpawnWithPeerScope(FireworkPrefab);
}
```

Creating Spawnable Objects

When objects are spawned they need to be given the same network Id so they can communicate.

When objects are created at Design Time, this is done based on their position in the graph. When they are created at Runtime, they may not end up in the same position so this needs to be done explicitly.

When using the `NetworkSpawner`, Ids are assigned automatically, for every Network Object in the Prefab. For this to happen, Networked Objects must implement the `INetworkSpawnable` interface.

Update your networked object to use `Ubiq.Spawning`, then define `INetworkSpawnable` for your class. This interface requires you to have a property called NetworkId of the type NetworkId that can be read and written to. Create such a property with the default get and set accessors.

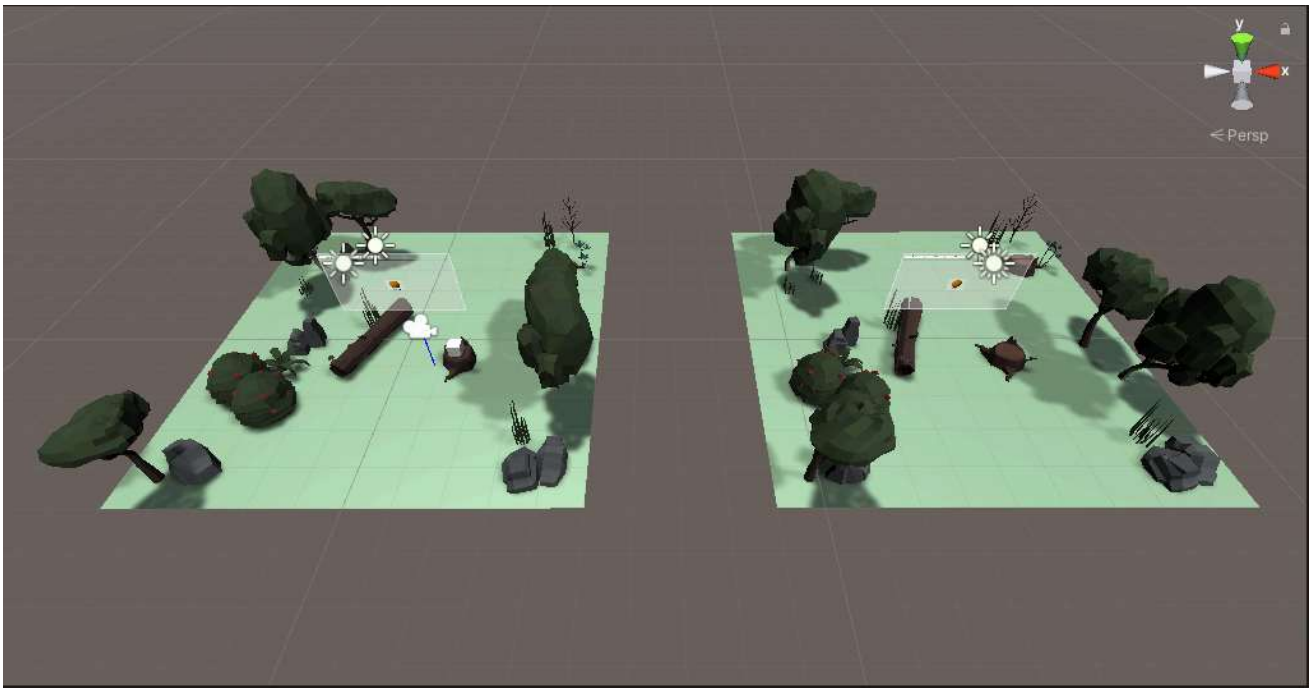
You don't have to do anything with the property - just have it in your class to satisfy `INetworkSpawnable`. Spawned instances of your object will now synchronise correctly.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Ubiq.Messaging;
using Ubiq.Spawning;

public class MyNetworkedObject : MonoBehaviour, INetworkSpawnable
{
    public NetworkId NetworkId { get; set; }

    NetworkContext context;

    // Start is called before the first frame update
    void Start()
    {
        context = NetworkScene.Register(this);
    }

    Vector3 lastPosition;

    // Update is called once per frame
    void Update()
    {
        if(lastPosition != transform.localPosition)
        {
            lastPosition = transform.localPosition;
            context.SendJson(new Message()
            {
                position = transform.localPosition
            });
        }
    }

    private struct Message
    {
        public Vector3 position;
    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        // Parse the message
        var m = message.FromJson<Message>();

        // Use the message to update the Component
        transform.localPosition = m.position;

        // Make sure the logic in Update doesn't trigger as a result of this message
        lastPosition = transform.localPosition;
    }
}
```

Create a Prefab by dragging your Networked Object from the "Creating Networked Objects" tutorial into the Assets view, then add this to the Catalogue.

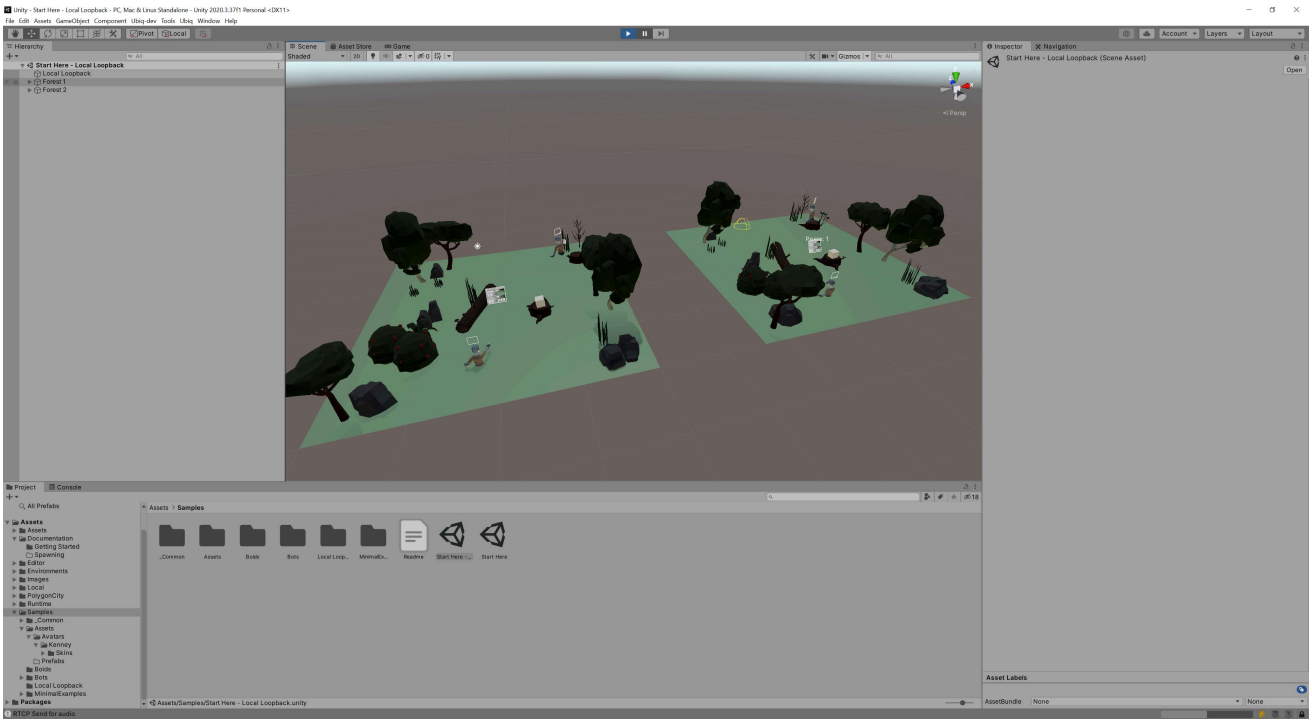
Testing Networked Objects

A Local Loopback scene is included in the Ubiq Samples. Local loopback is helpful for testing your networked objects in only one copy of Unity. The scene contains two clients that join a new room when you enter Play Mode.



The scene is in the Samples folder (`Start Here - Local Loopback`). Click the play button to get started.

In the Scene View you should see two Avatars. One avatar is you - controlled via the Game View - and the other is Bot, which joins the other NetworkScene.



Each environment is a separate Ubiq Peer. The script attached to the Local Loopback GameObject tells both instances to join the same Room. You can see on your in-game display the Join Code for this Room. Try and join it with a third Ubiq client and see what happens!

Forests

Objects, such as Avatars and Spawned Items, are associated with a `NetworkScene`. They find their Network Scene by searching the Scene Graph for the closest one.

The Local Loopback scene has two Network Scenes, each under a top level GameObject, called a *Forest*. These GameObjects separate the Unity scene into branches, where all Components in one branch will find that branch's Network Scene.

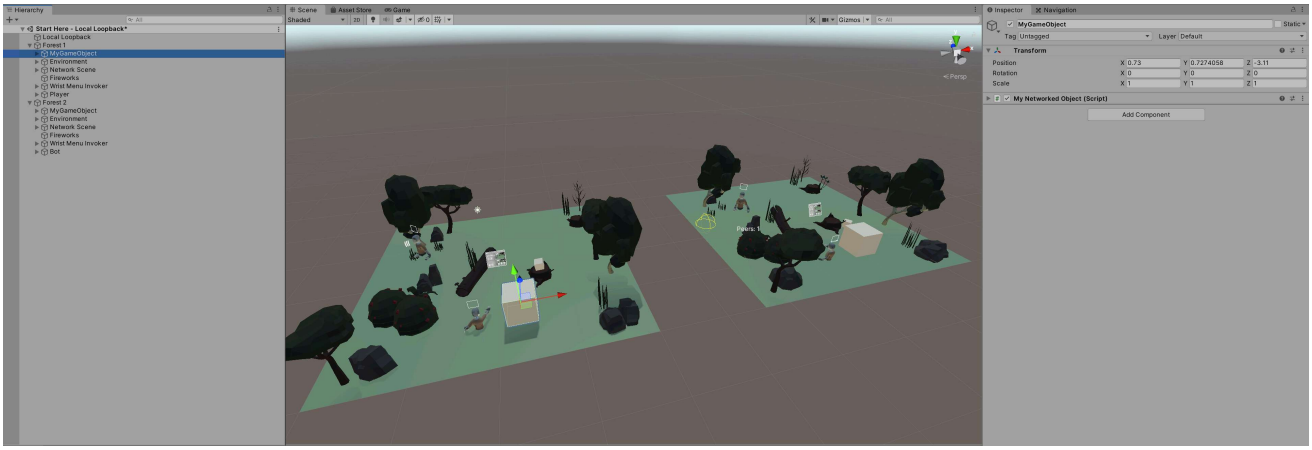
You can add more Forests to the scene to create more Peers.

Only one Peer can be controlled by the PlayerController at any time however, since Unity only supports one set of Input Devices. The other Peers are controlled by Bots.

Synchronisation

Try out your Prefab

Add a copy of your Prefab from the first tutorial page into each Forest and press Play.



Drag one of your GameObjects around, and see it change in the other scene. Try and drag the second instance around, and see the first one change position.

Your MyNetworkedObject Component synchronises the object's positions by looking for *external* changes each frame. This achieved in the `Update` method:

```
if(lastPosition != transform.localPosition)
{
    lastPosition = transform.localPosition;
    context.SendJson(new Message()
    {
        position = transform.localPosition
    });
}
```

If the position doesn't change within a frame, then no updates are sent.

The position only changes in one of two ways:

1. Something outside the script moves the object in *this Peer*.
2. A message from another instance moves the object.

In the second case, the position is updated but no message is sent, because it means something moved the object in *another Peer*.

Ownership

This approach of sending only local changes is simple, but isn't perfect.

Have a friend join a Room containing your object, and see what happens when you both try and move it at the same time.

In networked environments, two users can try to change the same thing at the same time. How those conflicts are resolved is up to you.

Another straightforward approach is to use Ownership Based Synchronisation.

In this case, a variable specifies who "owns" an object. Only the owner can send updates, and all other instances only receive them. This is how the Firework example is implemented:

```
private void Update()
{
    if(attached)
    {
        transform.position = attached.transform.position;
        transform.rotation = attached.transform.rotation;
    }
    if(owner)
    {
        context.SendJson(new Message(transform, fired));
    }
    if(owner && fired)
    {
        body.isKinematic = false;
        body.AddForce(transform.up * 0.75f, ForceMode.Force);

        if (!particles.isPlaying)
        {
            particles.Play();
            body.AddForce(new Vector3(Random.value, Random.value, Random.value) * 1.1f, ForceMode
        }
    }
    if(!owner && fired)
    {
        if (!particles.isPlaying)
        {
            particles.Play();
        }
    }
}
```

See how the behaviour of the script changes depending on the state of the `owner` flag. Only the owner can apply forces to the object, or send its position onto the network. Other instances simply attempt to make their local object match the state sent to them by the owner.

Spawning

When you create your own objects, you (or your code) decides which instance is the owner.

For example, in the case of the Firework, the Fireworks Box sets the Owner flag when it spawns the object.

Since the `Owner` flag defaults to `false`, spawned objects on remote instances will have this set to `false` and all objects will be configured correctly.

Making an Object Graspable

If you want to have your object being grabbable with the user's hands, you will have to inherit from IGraspable and implement Grasp(...) and Release(...)

Making an Object Usable

If you want your object to do something when the user presses the trigger button while holding it, you need to inherit from IUsable and implement Use(...) and UnUse(...)

Servers and Rooms

Ubiq clients can connect directly to eachother, but most applications will use a room server.

The room server is a central service which clients connect to and use to rendezvous with eachother. A server can host multiple *Rooms*. Rooms are a collection of users in a scene, who can talk to eachother and exchange messages. The `RoomClient` is used to find, join and leave rooms.

The RoomClient must be provided with a server to connect to. The VECG team runs a public server, `nexus.cs.ucl.ac.uk`, or you can set up your own.

`RoomClient` will connect to the server in the `Default Server` property on start-up, but you must join a room before you can communicate with other users. This can be done in the Editor through the button in the Inspector, through the RoomClient's API or through the example UI.

Logging

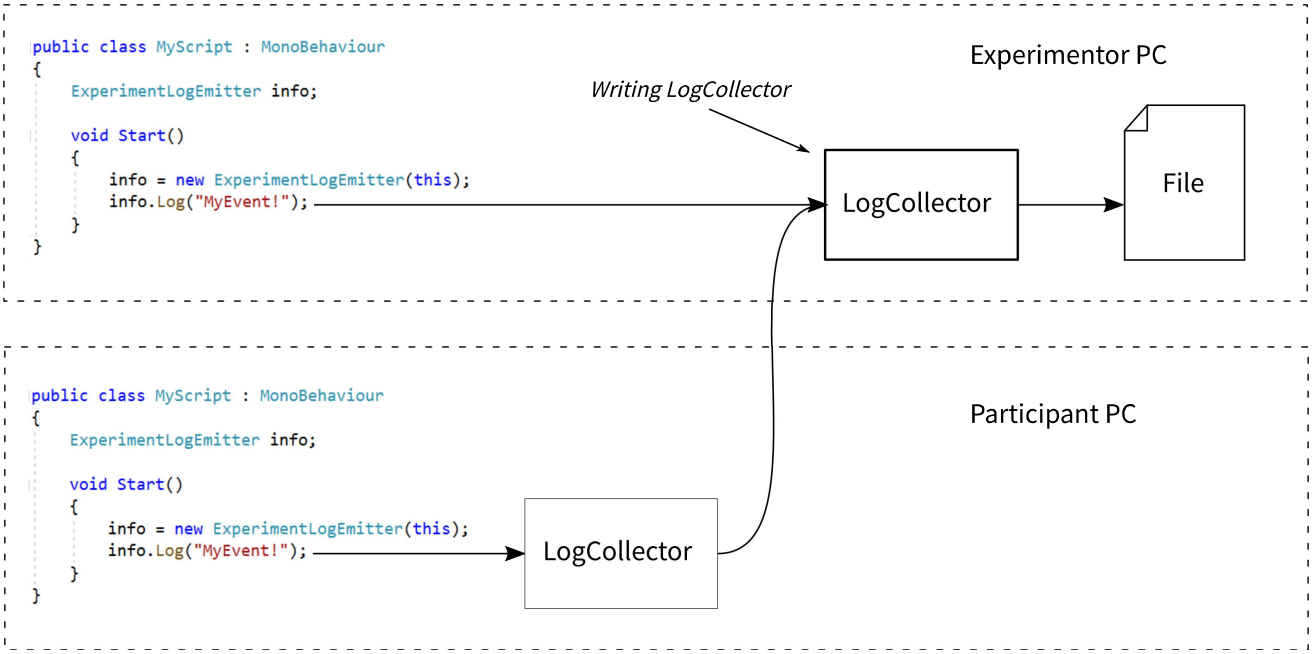
Ubiq has the ability to record, forward and store logs. Ubiq itself generates logs, and custom components can create them too.

For example, the logging system could be used to record the answers to a questionnaire, or the direction of a user's gaze, and forward them to an experimentor.

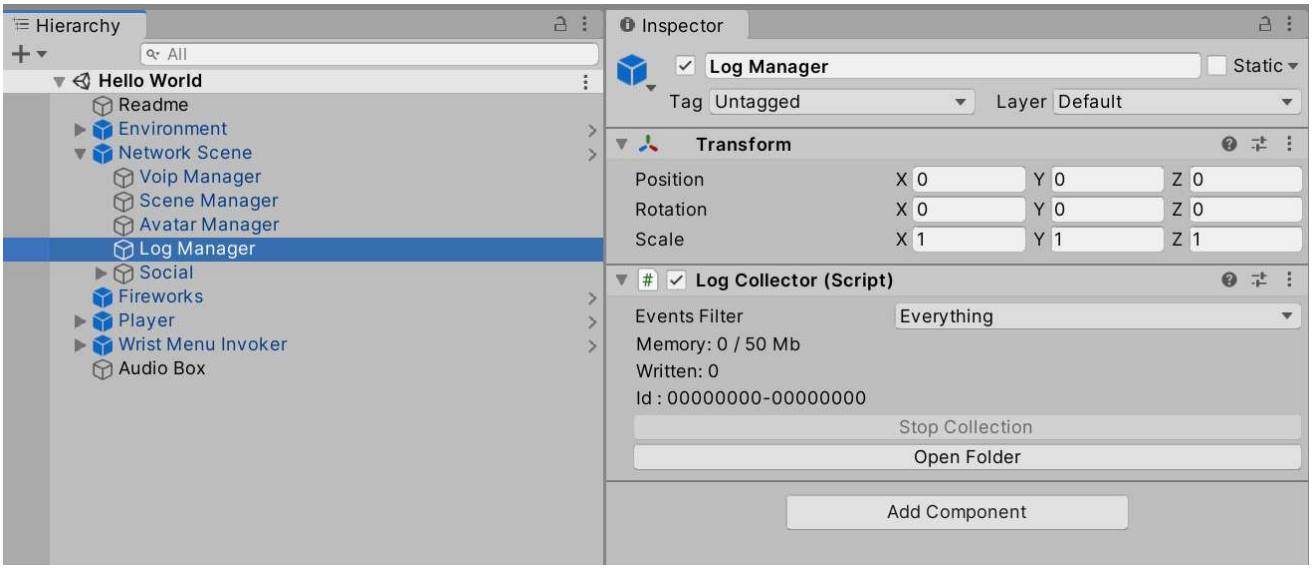
This guide shows how to set up and log some simple data in the *Hello World* scene.

Log Flow

Log events (such as answering a question) are generated by Log Emitters with a simple call, e.g. `debug.Log("MyEvent")`. These events are received by a Log Collector. `LogCollector` is a Component belonging to a `NetworkScene`. Depending on where logs are finally written, the local `LogCollector` may forward events to another `LogCollector` in the same room, or write them directly to a file.



There can be many Log Emitters in an application. There should be one `LogCollector` per Peer. Only one `LogCollector` should be *writing* at a time.



Creating a Questionnaire Button

Log events can come from any source. In this guide, they will be generated when a user presses a button.

Create a new `Button` in the scene. Below, a new `GameObject` was added to the *Main Menu*. Create a new script, `ButtonLogger`, and add it to the Button as well.

The script for `ButtonLogger` is below.

```
using Ubiq.Logging;
using UnityEngine;
using UnityEngine.UI;

public class ButtonLogger : MonoBehaviour
{
    ExperimentLogEmitter events;

    // Start is called before the first frame update
    void Start()
    {
        events = new ExperimentLogEmitter(this);
        GetComponent<Button>().onClick.AddListener(OnButtonClicked);
    }

    void OnButtonClicked()
    {
        events.Log("Button Pressed");
    }
}
```

First, an `ExperimentLogEmitter` is declared. This is the object that will be used to emit log events. It is declared in the class but initialised in `Start()`. This is because it has to find the local Log Collector to communicate with, which can't be done until the scene initialisation begins.

Event Types

Events can be given a Type. The type hints at the meaning of the event. For example, `Info` events record how the application itself is working. `Experiment` events could record data for experiments. `Debug` events could record simple debugging information.

Any code can create any type of event. The type is used to filter events.

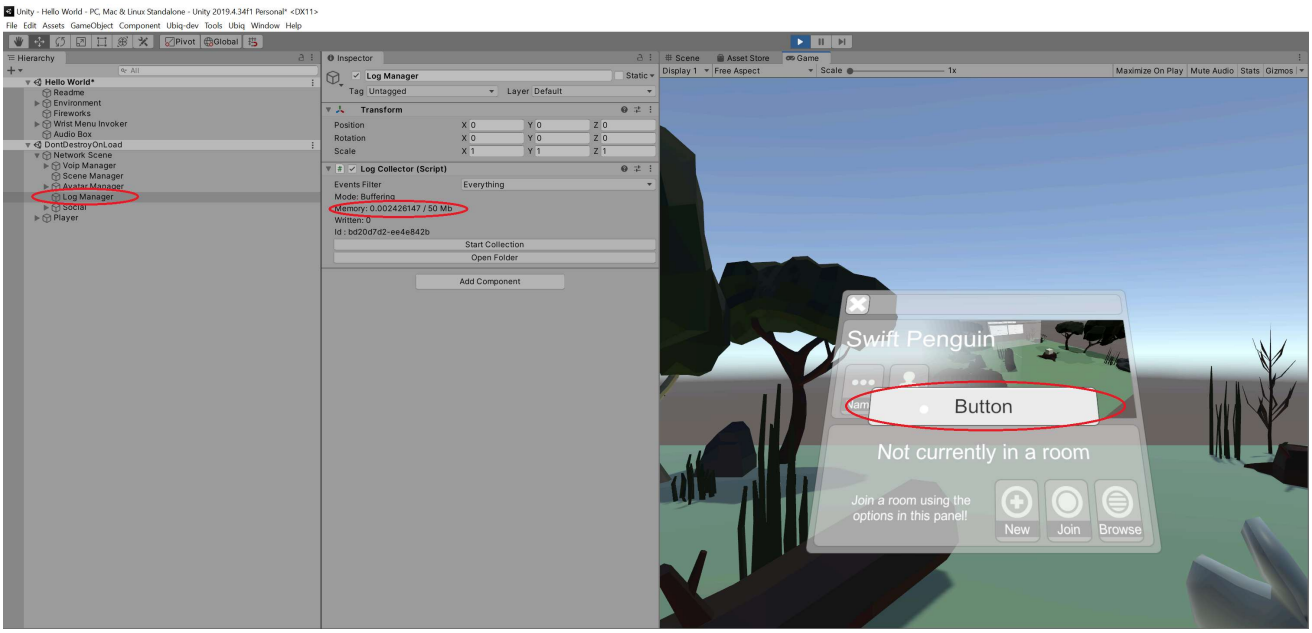
The Emitter created in the `ButtonLogger` script will generate `Experiment` events.

Log Collector

The default NetworkScene Prefab already contains a `LogCollector`, so there is no need to add this.

A callback is registered with the Button's `OnClick` event by the `ButtonLogger` script. When this is raised by the user clicking the button, a log event ("Button Pressed") is emitted.

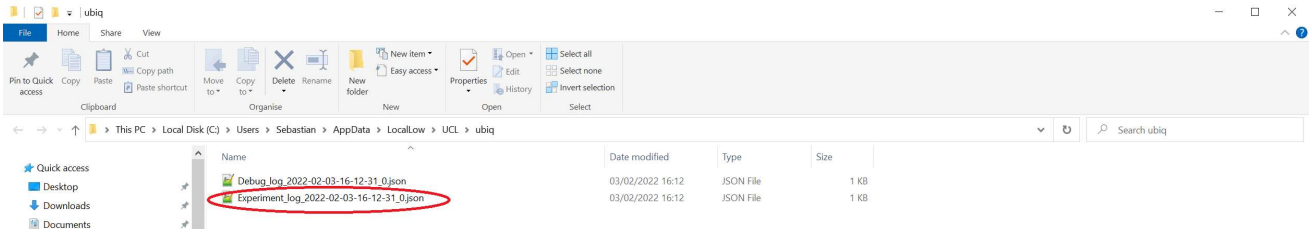
Start the Scene and look at the Log Collector in the Inspector. As the Button is clicked the memory usage of the collector will increase, indicating that the Button is generating events.



Writing Logs

The events will remain in the `LogCollector` until they are requested.

Click *Start Collection*. The Entries count will increase, and opening the log folder will reveal an Experiment log, with a number of *Button Pressed* events.




```
[{"ticks":637795015469208026,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015470638029,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015471998382,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015473438942,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015474853269,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015476313220,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015477743218,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015479173222,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015480802962,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"},
{"ticks":637795015482232892,"peer":"f6aa7d01-24da1cf2","event":"Button Pressed"}
```

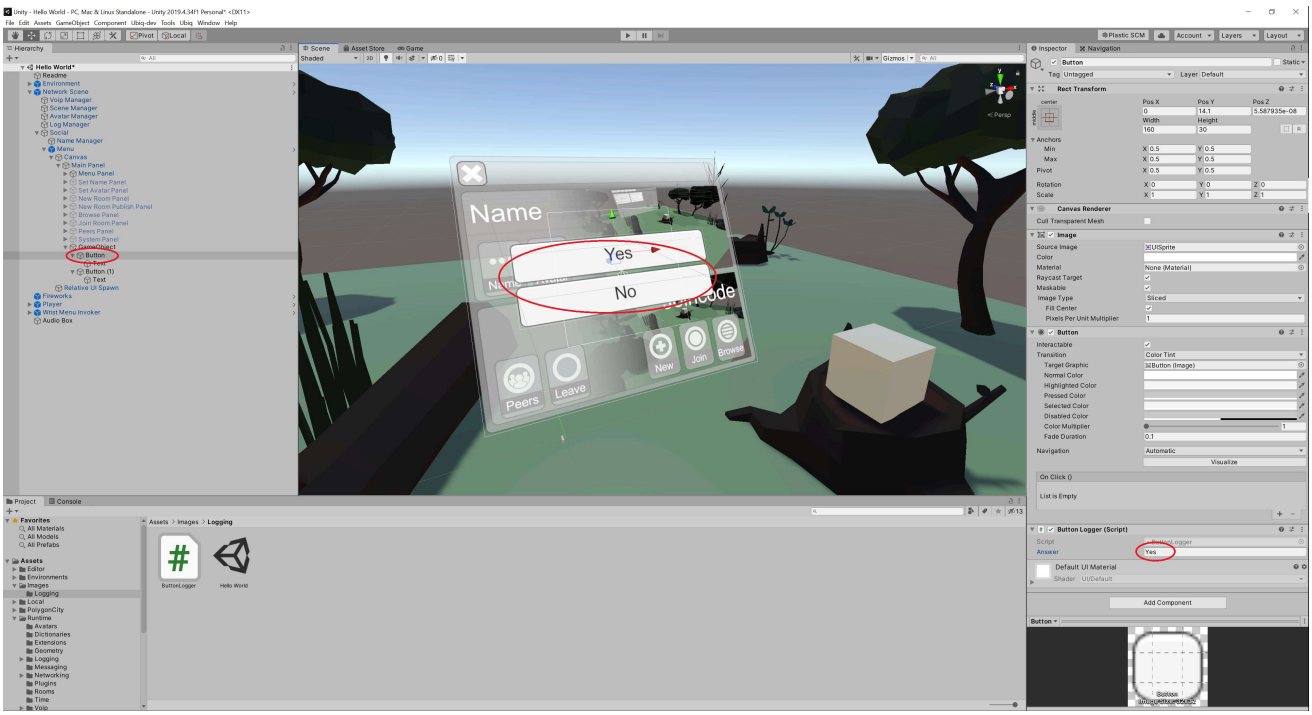

Logging Arguments

The `LogEmitter::Log()` method can take a number of arguments in addition to the event name.

Add a new member to the ButtonLogger, `AnswerName`, and pass it in as an argument.

```
    public string Answer;

    void OnButtonClicked()
    {
        events.Log("Button Pressed", Answer);
    }
```

The value of Answer can be set up in the inspector. Duplicate the Button and set two different values of Answer for each.



Now, when looking at the log after pressing the buttons it will show the value of Answer as well.


```
[{"ticks":637795019171902297,"peer":"ba742247-1415eb07","event":"Button Pressed","arg1":"Yes"},
{"ticks":637795019174622297,"peer":"ba742247-1415eb07","event":"Button Pressed","arg1":"Yes"},
{"ticks":637795019180442303,"peer":"ba742247-1415eb07","event":"Button Pressed","arg1":"No"},
{"ticks":637795019183882312,"peer":"ba742247-1415eb07","event":"Button Pressed","arg1":"No"},
{"ticks":637795019201742296,"peer":"ba742247-1415eb07","event":"Button Pressed","arg1":"Yes"}
```


Practically any variable that can be turned into a string can be logged this way.

Collecting from a Distributed Experiment

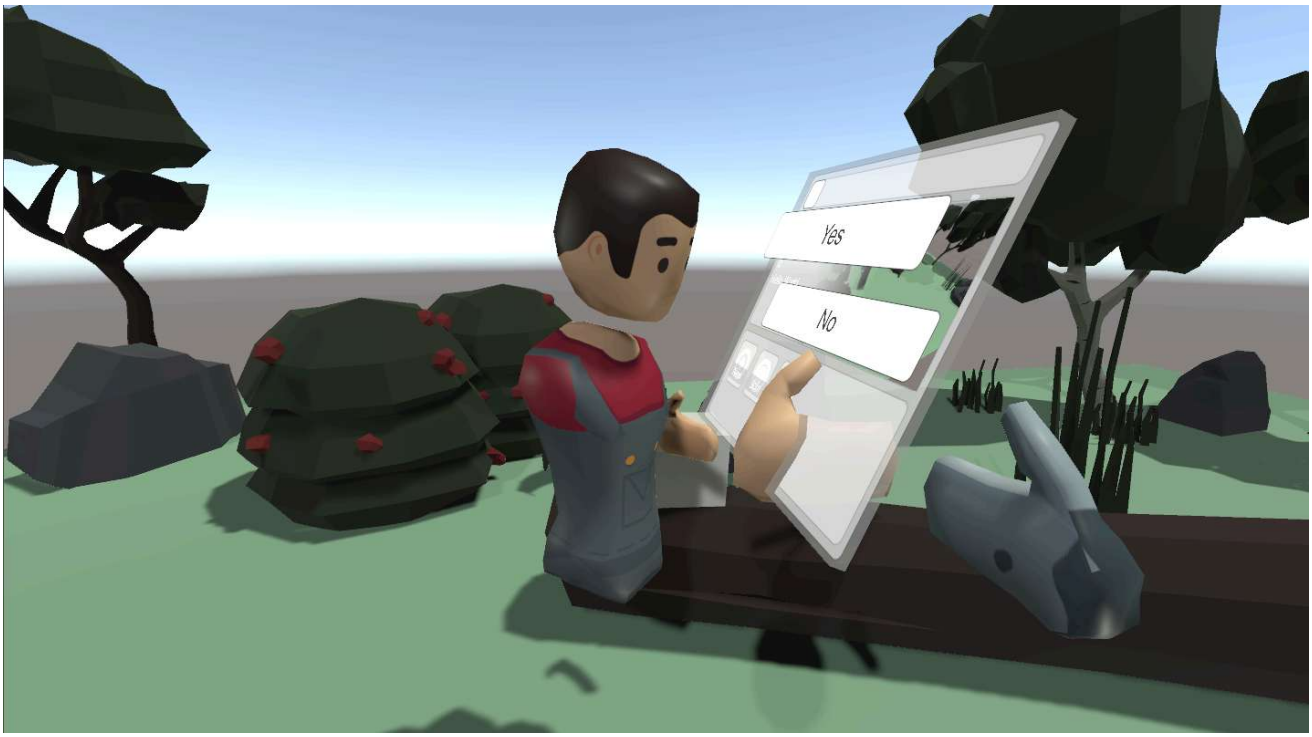
So far the `LogCollector` has just collected from the local player.

Create a Build of the *Hello World* application and run it, then press the buttons a few times.

Note that so far, the application has not even joined a room. This is OK because the `LogCollector` will hold all logs until they are requested.

Next, press *Play* to load the *Hello World* Scene in the Editor.

Now have both Peers join the same room (new or old, in any order). When both have joined, the Avatars of the other Peer should be visible in each.



In the Editor, navigate to the `LogCollector` and click on *Start Collection* in the Inspector.

The Entries count will increase, and an Experiment Log file will appear in the default Logs Folder, containing any answers entered in both the Editor and Standalone Build.

Considerations

To find out more about the logging, see the Logging section in the Advanced topics.

Log events can be generated from user actions, but also other external events, or at a regular frequency (e.g. to log the `Transform` of dynamic objects)

You can change the active `LogCollector` at runtime losslessly, so long as no Peers fail or unexpectedly disconnect.

Collection can also be started programmatically, in addition to clicking *Start Collection*. This allows experiment code to start collection other ways, including in Standalone builds.

See the Samples/Single/Questionnaire sample for a complete Questionnaire implementation.

Creating a 3D Pen for Ubiq

In this short tutorial we'll make a pen that lets us draw shapes in mid-air. We'll add simple networking with Ubiq so you can share your drawings with others!

> This tutorial is intended for Ubiq v1.0.0-pre.7 and may not apply to other versions. In particular, versions of Ubiq prior to v1.0.0-pre.1 do not use Unity's XR Interaction Toolkit.

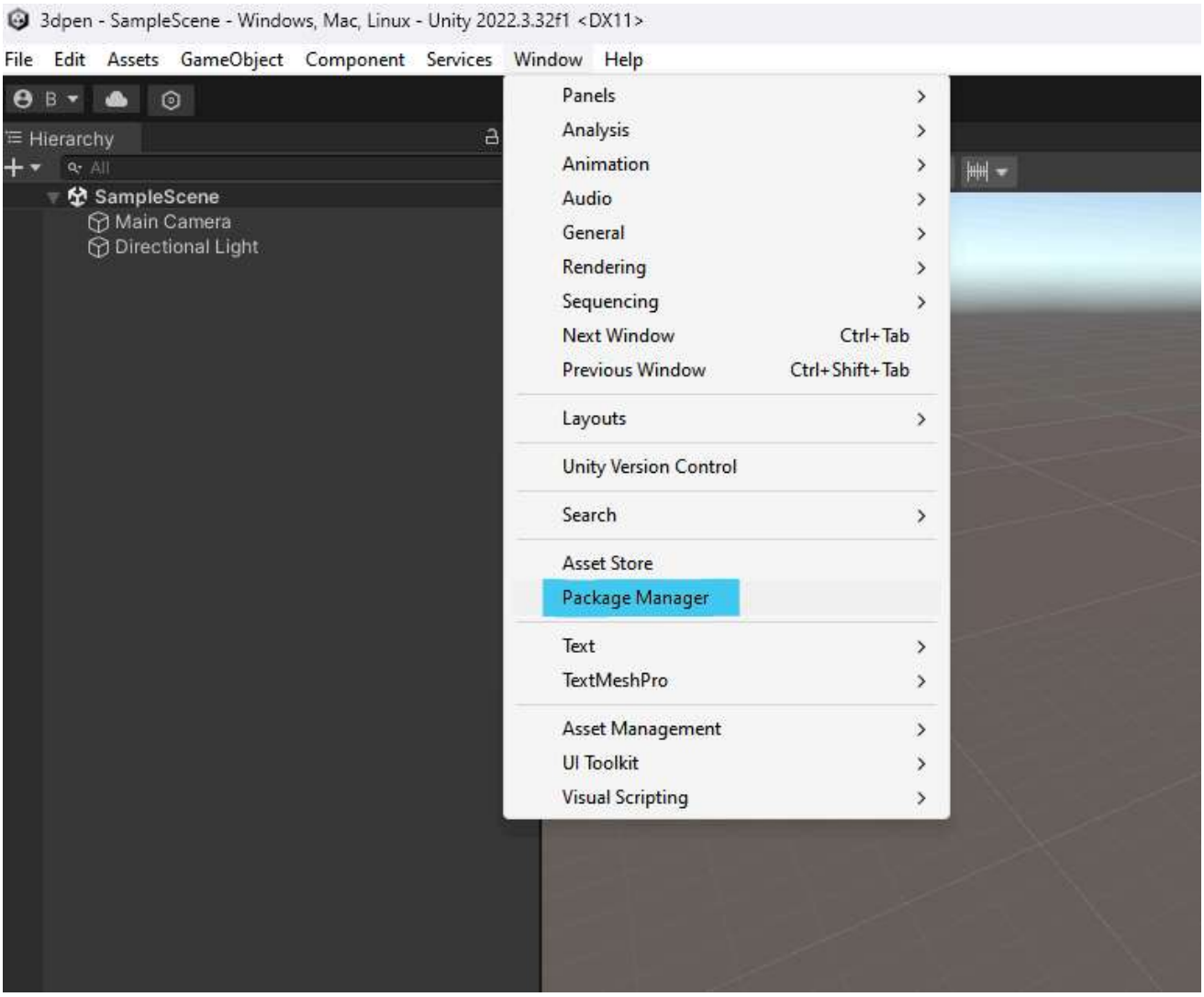
Setting up a new Unity project

Download and install the Unity editor. We are using Unity 2022.3.32f1. Ubiq v1.0.0-pre.7 is compatible with Unity 2021.3.22 LTS and later.

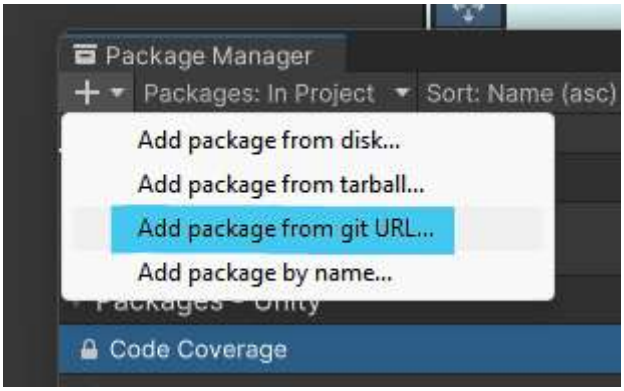
Create a new Unity project with the 3D template and wait for the editor to open the new project.

Installing Ubiq as a package

Open the Package Manager by selecting 'Window > Package Manager' in the Menu bar at the top of the Unity Editor.



Click the '+' button in the top-left corner of the Package Manager window and select 'Add package from git URL...'



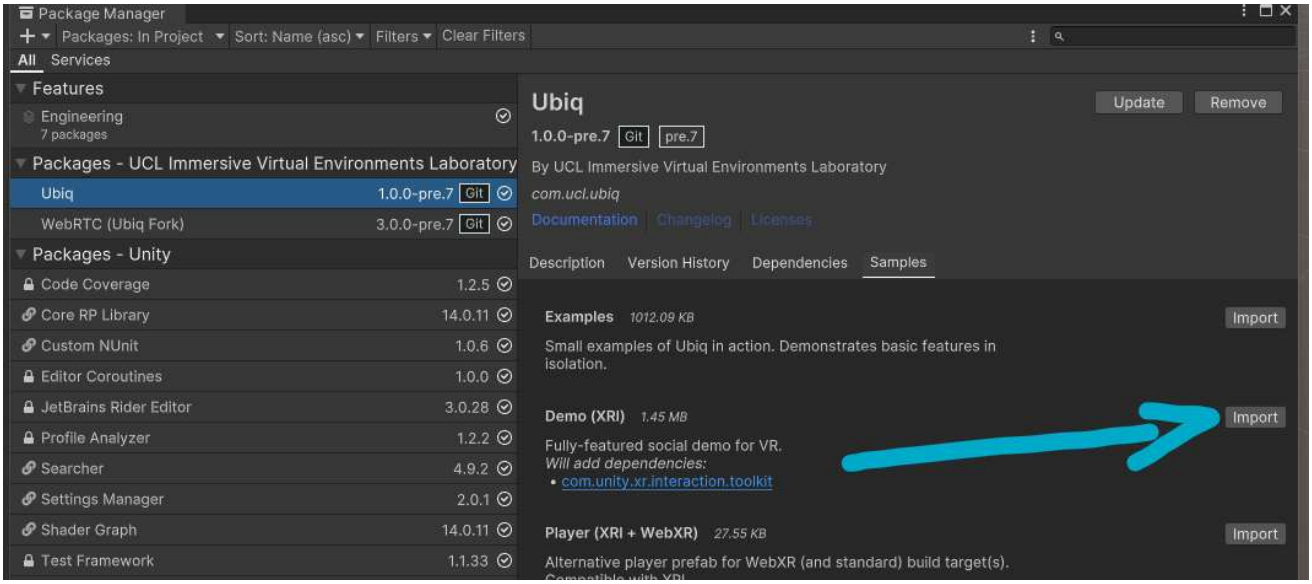
Enter the url `https://github.com/UCL-VR/ubiq.git#upm-unity-v1.0.0-pre.7` and select 'Add' or press enter. This is the URL to install Ubiq v1.0.0-pre.7 specifically.

Wait for Ubiq to be downloaded and imported.

Ubiq will install further dependencies in the background. When this is complete, you will see 'Ubiq successfully modified project requirements.' in the editor console.

Import the Ubiq XRI Demo sample

Back in the Package Manager, select Ubiq from the list on the left. In the pane on the right, switch to the Samples tab, then Import to the 'Demo (XRI)' sample. Wait for the import to complete. If a Unity popup asks about enabling the new Input backend, select Yes.



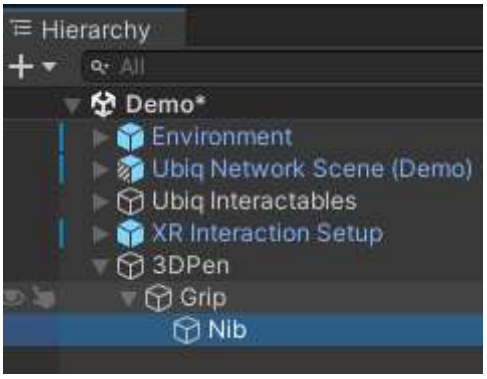
Open the Ubiq demo scene from the Samples: Assets/Samples/Ubiq/1.0.0-pre.7/Demo (XRI)/Demo.unity

Setting up a simple pen with primitive shapes

Now we have everything we need, let's make a simple object for the pen. We will make a simple stand-in with one big cylinder for the grip and a tiny one for the nib.

Right click in the hierarchy window and select Create Empty. Right click on the object in the hierarchy, select Rename, and give it the name '3DPen'. This will be our parent object. We can use this to customize how our object is picked up.

Now right click on 3DPen in the Hierarchy window and select 3D Object/Cylinder. Rename this cylinder 'Grip'. Right click on Grip and again select 3D Object/Cylinder. Rename this new cylinder 'Nib'. Your hierarchy should now read 3DPen/Grip/Nib, as in the image.



Scale, and translate the objects so they (sort of!) resemble a pen. Do not translate the top object (3DPen), as this will get moved when the user grabs it. Focus on just moving and scaling Grip and Nib. Do not worry about rotation now; we'll deal with this later.

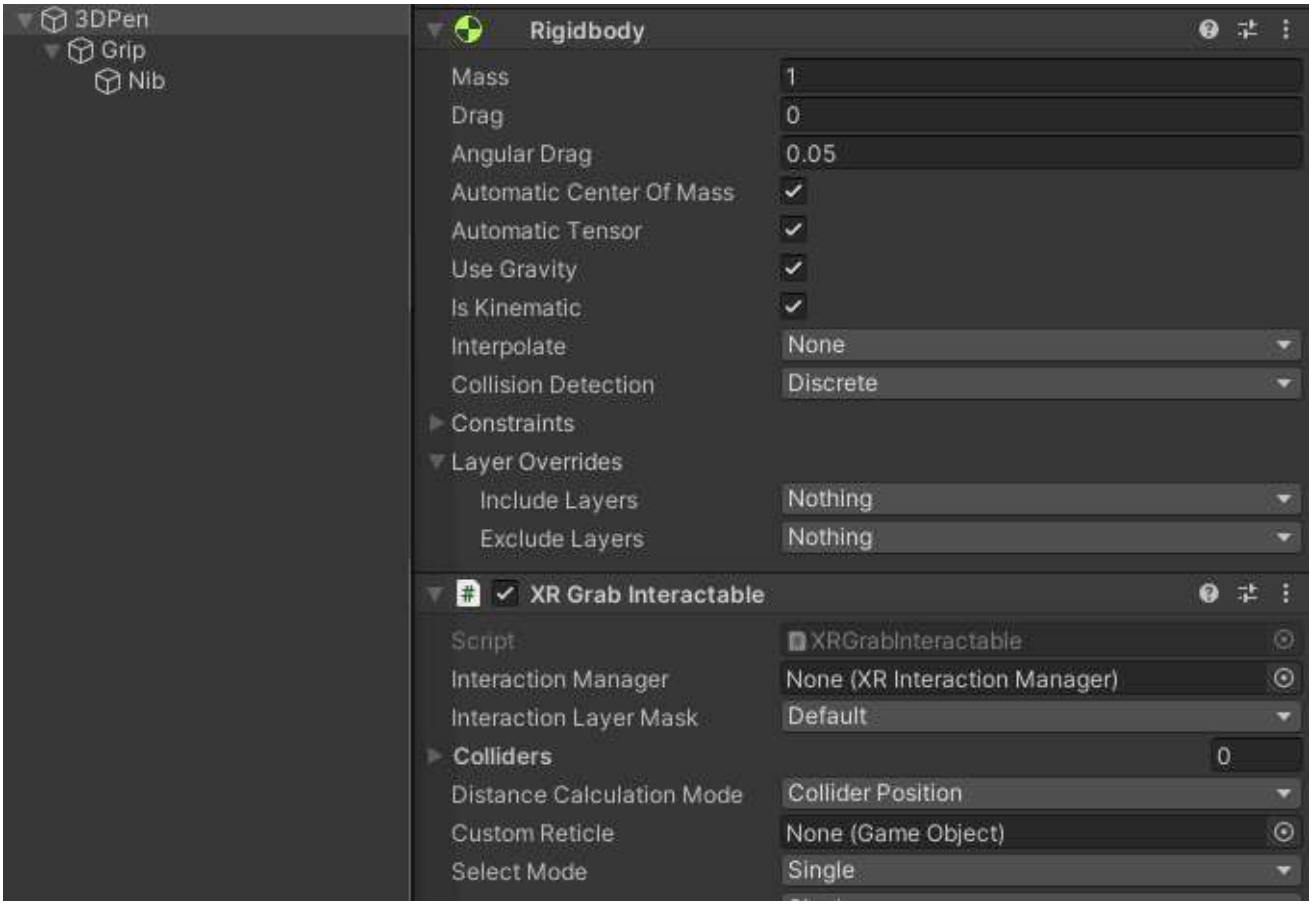
> Tip: the scale of the sample is 1m equals 1 unit.

Finally, position the pen somewhere you can easily reach it. You could try next to the menu. Here's what we ended up with:

Adding physics and connecting to the XR Interaction Toolkit

Select 3DPen in the hierarchy window. Now, in the Inspector window, select Add Component and add an XRGrabInteractable. This will automatically add a Rigidbody. In the Rigidbody, tick 'IsKinematic'. In the XRGrabInteractable, untick 'Throw on Detach'.

> IsKinematic prevents the GameObject's transform being driven by the Physics system. This makes it easier for us to drive it with networking code later on.



The XRGrabInteractable tells the XR Interaction Toolkit how to treat the 3D pen. It now knows it can be picked up and moved around.

Now let's test what we have made so far.

Testing our work so far - Headset

If you have a virtual reality headset to hand, you can test the application by connecting that headset and entering play mode in the editor. Setting up a project for XR is a big topic and varies slightly by headset. A full guide is beyond the scope of this tutorial, but if you are new to working with XR, we provide some recommendations below.

If you have a desktop headset which supports OpenXR we recommend following the Unity instructions here. With a correctly set up project, you should then be able to enter Play mode in the editor and Unity will connect to your device.

If you have a mobile or standalone headset, you can create builds for testing but this is slow. As changes will not be reflected in the editor debugging also becomes much more time consuming. You may be able to configure some form of streaming from your development machine to your headset, which can speed up development immensely. See Meta's guide for this feature on the Meta Quest here. Other standalone headsets (e.g., the XR elite) have similar streaming capabilities.

Once you have your headset set up, enter Play mode and experiment with picking up the pen and moving it around using the 'grab' interaction for your device.

(Fallback) Testing our work so far - Simulator

If you do not have a virtual reality headset for testing, the Demo scene in Ubiq is setup to use Unity's XR Device Simulator by default. The Simulator takes a bit of practice to use but does allow you to functionally test simple interactions without the need for a headset.

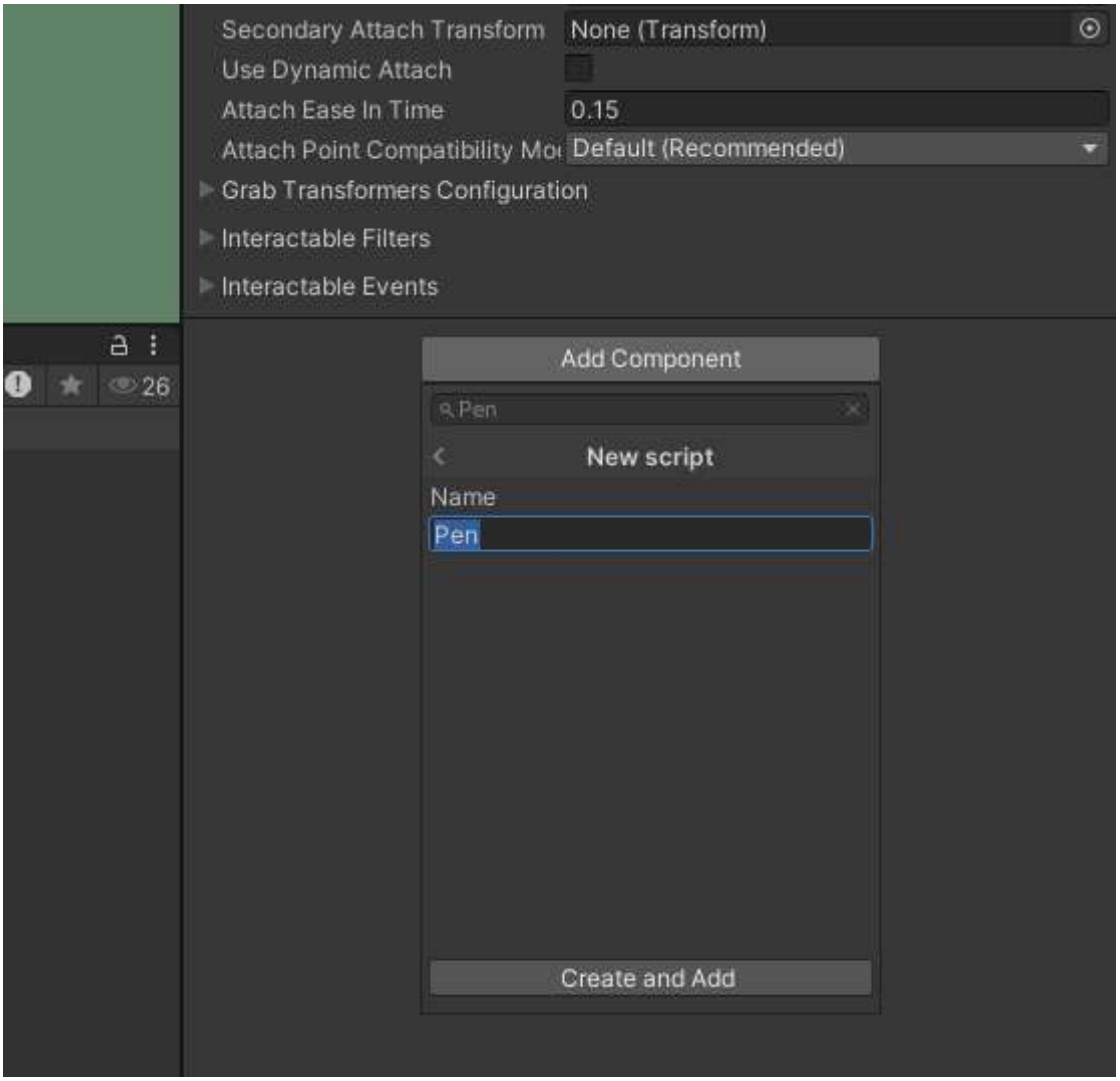
The XR Device Simulator control help window will popup in the lower left corner of the Game window when you enter Play mode in the editor. If it does not expand by default, click the '+' symbol on the right hand side of the minimised bar to expand them.

Enter Play mode and experiment with the simulator controls until you can pick up the pen and move it around.

Drawing in mid-air with the 3D Pen

Either using a headset or the simulator, you should now be able to grab and move the pen.

Next let's write a script to get the pen drawing in mid-air. Select 3DPen in the Hierarchy window again. In the Inspector window, select Add Component, and type Pen. Unity will prompt you to add a new script with that name - select New Script, then Create and Add.



This will create the Pen script in your Assets folder and attach that script to the object. Open the file (Assets/Pen.cs) and replace its contents with the following:

```
using UnityEngine;
using UnityEngine.XR.Interaction.Toolkit;

public class Pen : MonoBehaviour
{
    private Transform nib;
    private Material drawingMaterial;
    private GameObject currentDrawing;

    private void Start()
    {
        nib = transform.Find("Grip/Nib");

        var shader = Shader.Find("Particles/Standard Unlit");
        drawingMaterial = new Material(shader);

        var grab = GetComponent<XRGrabInteractable>();
        grab.activated.AddListener(XRGrabInteractable_Activated);
        grab.deactivated.AddListener(XRGrabInteractable_Deactivated);
    }

    private void XRGrabInteractable_Activated(ActivateEventArgs eventArgs)
    {
        BeginDrawing();
    }

    private void XRGrabInteractable_Deactivated(DeactivateEventArgs eventArgs)
    {
        EndDrawing();
    }

    private void BeginDrawing()
    {
        currentDrawing = new GameObject("Drawing");
        var trail = currentDrawing.AddComponent<TrailRenderer>();
        trail.time = Mathf.Infinity;
        trail.material = drawingMaterial;
        trail.startWidth = .05f;
        trail.endWidth = .05f;
        trail.minVertexDistance = .02f;

        currentDrawing.transform.parent = nib.transform;
        currentDrawing.transform.localPosition = Vector3.zero;
        currentDrawing.transform.localRotation = Quaternion.identity;
    }

    private void EndDrawing()
    {
        currentDrawing.transform.parent = null;
        currentDrawing.GetComponent<TrailRenderer>().emitting = false;
        currentDrawing = null;
    }
}
```

Test using the simulator or a headset. The pen will draw in the air when 'activated' - on most devices, this is when the primary trigger is pulled. A new drawing is created every time the user activates the pen.

Testing the networking - Building a test application

Let's build your application to test networking functionality.

First, go to the top bar, then Edit > Project Settings. In the Project Settings window, select Player from the list on the left. In the pane on the right, expand the 'Resolution and Presentation' foldout, then click the dropdown next to Fullscreen Mode and select Windowed, then set the window to something small, like 640 x 480. This helps us test because we can see both the editor and the application running in a small window.

Now, again in the top bar, go to File/Build and Run. Select a location for the build, and wait for the build to complete.

Note that this assumes your build target is currently Desktop. If you've followed this tutorial from the start, this should be the case by default.

Testing the networking - Joining a room

Now you can run your application in both the editor and as a windowed standalone app. To connect the two, we'll need them both to join the same room.

We do this by interacting with the Ubiq UI panel in the scene. If you are using the simulator, you can use your mouse to interact with the UI (press Escape to stop controlling the simulator). If you are using a headset, you can point at the option you want to select and 'activate' (most likely pull the primary trigger).

Enter play mode in the editor. Select the 'New' button on the Ubiq UI panel in the scene. Leave the name as default, and click the arrow at the top right of the UI panel. Finally, select 'No, keep my room private'. The panel will change to show you a three letter code. This is the 'joincode' for your room. In your standalone windowed application, click Join on the Ubiq sample UI, enter this code, then click the arrow to submit.

Testing the networking - Comparing the applications

Now you have two applications, both connected to the same room. On both, you should now see another avatar in the room with you. Try picking up the object as before. You'll see that it moves for the user who picked it up, but in the other application it stays still. And when you draw with the pen, the other user will not see these drawings either! We'll need to add some networking for both these behaviours.

Adding networked movement

Replace your Pen script (Assets/Pen.cs) with the following:

```
using UnityEngine;
using UnityEngine.XR.Interaction.Toolkit;
using Ubiq.Messaging; // new

public class Pen : MonoBehaviour
{
    private NetworkContext context; // new
    private bool owner; // new
    private Transform nib;
    private Material drawingMaterial;
    private GameObject currentDrawing;

    // new
    // 1. Define a message format. Lets us know what to expect on send and recv
    private struct Message
    {
        public Vector3 position;
        public Quaternion rotation;

        public Message(Transform transform)
        {
            this.position = transform.position;
            this.rotation = transform.rotation;
        }
    }

    private void Start()
    {
        nib = transform.Find("Grip/Nib");

        var shader = Shader.Find("Sprites/Default");
        drawingMaterial = new Material(shader);

        var grab = GetComponent<XRGrabInteractable>();
        grab.activated.AddListener(XRGrabInteractable_Activated);
        grab.deactivated.AddListener(XRGrabInteractable_Deactivated);

        // new
        grab.selectEntered.AddListener(XRGrabInteractable_SelectEntered);
        grab.selectExited.AddListener(XRGrabInteractable_SelectExited);

        // new
        // 2. Register the object with the network scene. This provides a
        // NetworkID for the object and lets it get messages from remote users
        context = NetworkScene.Register(this);
    }

    // new
    private void FixedUpdate()
    {
        if (owner)
        {
            // 3. Send transform update messages if we are the current 'owner'
            context.SendJson(new Message(transform));
        }
    }

    // new
    public void ProcessMessage (ReferenceCountedSceneGraphMessage msg)
    {
        // 4. Receive and use transform update messages from remote users
        // Here we use them to update our current position
        var data = msg.FromJson<Message>();
        transform.position = data.position;
        transform.rotation = data.rotation;
    }

    private void XRGrabInteractable_Activated(ActivateEventArgs eventArgs)
    {
        BeginDrawing();
    }

    private void XRGrabInteractable_Deactivated(DeactivateEventArgs eventArgs)
    {
        EndDrawing();
    }
```

```
    // new
    private void XRGrabInteractable_SelectEntered(SelectEnterEventArgs arg0)
    {
        // 5. The pen has been grabbed. We hold the pen currently, so we're
        // the owner
        owner = true;
    }

    // new
    private void XRGrabInteractable_SelectExited(SelectExitEventArgs eventArgs)
    {
        // 5. The pen has been released. We no longer hold the pen, so we're
        // no longer the owner
        owner = false;
    }

    // Note about ownership: 'ownership' is just one way of designing this
    // kind of script. It's sometimes a useful pattern, but has no special
    // significance outside of this file or in Ubiq more generally.

    private void BeginDrawing()
    {
        currentDrawing = new GameObject("Drawing");
        var trail = currentDrawing.AddComponent<TrailRenderer>();
        trail.time = Mathf.Infinity;
        trail.material = drawingMaterial;
        trail.startWidth = .05f;
        trail.endWidth = .05f;
        trail.minVertexDistance = .02f;

        currentDrawing.transform.parent = nib.transform;
        currentDrawing.transform.localPosition = Vector3.zero;
        currentDrawing.transform.localRotation = Quaternion.identity;
    }

    private void EndDrawing()
    {
        currentDrawing.transform.parent = null;
        currentDrawing.GetComponent<TrailRenderer>().emitting = false;
        currentDrawing = null;
    }
}
```

New lines are marked with a comment. This script does a number of important things, marked in the code.

Test again as in the section 'Testing the networking'. You should now see that when the object is grasped and moved in one application, it also moves in the other.

Adding networked drawings

Time to add networking to our drawings! Replace Pen.cs with the following:

```
using UnityEngine;
using UnityEngine.XR.Interaction.Toolkit;
using Ubiq.Messaging;

public class Pen : MonoBehaviour
{
    private NetworkContext context;
    private bool owner;
    private Transform nib;
    private Material drawingMaterial;
    private GameObject currentDrawing;

    // 1. Amend message to also store current drawing state
    private struct Message
    {
        public Vector3 position;
        public Quaternion rotation;
        public bool isDrawing; // new

        public Message(Transform transform, bool isDrawing) // new
        {
            this.position = transform.position;
            this.rotation = transform.rotation;
            this.isDrawing = isDrawing; // new
        }
    }

    private void Start()
    {
        nib = transform.Find("Grip/Nib");

        var shader = Shader.Find("Sprites/Default");
        drawingMaterial = new Material(shader);

        var grab = GetComponent<XRGrabInteractable>();
        grab.activated.AddListener(XRGrabInteractable_Activated);
        grab.deactivated.AddListener(XRGrabInteractable_Deactivated);

        grab.selectEntered.AddListener(XRGrabInteractable_SelectEntered);
        grab.selectExited.AddListener(XRGrabInteractable_SelectExited);

        context = NetworkScene.Register(this);
    }

    private void FixedUpdate()
    {
        if (owner)
        {
            // new
            // 2. Send current drawing state if owner
            context.SendJson(new Message(transform,isDrawing:currentDrawing));
        }
    }

    public void ProcessMessage (ReferenceCountedSceneGraphMessage msg)
    {
        var data = msg.FromJson<Message>();
        transform.position = data.position;
        transform.rotation = data.rotation;

        // new
        // 3. Start drawing locally when a remote user starts
        if (data.isDrawing && !currentDrawing)
        {
            BeginDrawing();
        }
        if (!data.isDrawing && currentDrawing)
        {
            EndDrawing();
        }
    }

    private void XRGrabInteractable_Activated(ActivateEventArgs eventArgs)
    {
        BeginDrawing();
    }
```

```
    private void XRGrabInteractable_Deactivated(DeactivateEventArgs eventArgs)
    {
        EndDrawing();
    }

    private void XRGrabInteractable_SelectEntered(SelectEnterEventArgs arg0)
    {
        owner = true;
    }

    private void XRGrabInteractable_SelectExited(SelectExitEventArgs eventArgs)
    {
        owner = false;
    }

    private void BeginDrawing()
    {
        currentDrawing = new GameObject("Drawing");
        var trail = currentDrawing.AddComponent<TrailRenderer>();
        trail.time = Mathf.Infinity;
        trail.material = drawingMaterial;
        trail.startWidth = .05f;
        trail.endWidth = .05f;
        trail.minVertexDistance = .02f;

        currentDrawing.transform.parent = nib.transform;
        currentDrawing.transform.localPosition = Vector3.zero;
        currentDrawing.transform.localRotation = Quaternion.identity;
    }

    private void EndDrawing()
    {
        currentDrawing.transform.parent = null;
        currentDrawing.GetComponent<TrailRenderer>().emitting = false;
        currentDrawing = null;
    }
}
```

This adds simple networking to the 3d pen. The approach used is to draw locally when a remote user tells us they are drawing, and stop drawing locally when a remote user tells us they are not.

And we're done! Test it again as in 'Testing the networking', and if you have a headset available, see how it feels in VR!

You might notice a few things about the pen after testing. There's always more functionality to be added! If you're interested, read on for some further exercises and hints.

Extending this tutorial - new joiners

Drawings are not visible to new joining users. A more advanced implementation would store the points of the drawings in Peer properties. This way new users could see them when they join, and they'll be cleaned up when the user that created them leaves. See the Examples sample in the Ubiq package to get you started.

Extending this tutorial - ownership

If you try this tutorial out with another user, you might realise you get some strange behaviour when you pass the pen back and forth. This is because the simple version of 'ownership' used in this tutorial is incomplete. When two users both grab the pen, they both consider themselves the owner, and try to control the pen's position.

Ubiq provides the tools to create robust ownership. One option is setting the owner in the Room properties. These are stored on the server, which resolves conflicts and eventually synchronises the same owner with all room peers. See the Examples sample on properties to get an idea of how this works.

Quick Start

The server code is included in the Ubiq repository in the `Node` directory.

After checking out the code, run,

```
npm install
```

in the `Node` directory. The server can then be started with,

```
npm start
```

This will start a server with the default TCP port - the same one running on Nexus - that Unity can connect to. Continue below if you also need to support Browser Clients.

Advanced

Supporting Secure WebSockets

If you are intending to create Browser Clients with Ubiq's JavaScript library, you will need to support Secure WebSockets.

This is done by providing a certificate for the server.

The quickest way to do this is to create a self-signed certificate using, e.g. OpenSSL. In the `Node` directory, give the command,

```
openssl req  -nodes -new -x509  -keyout key.pem -out cert.pem
```

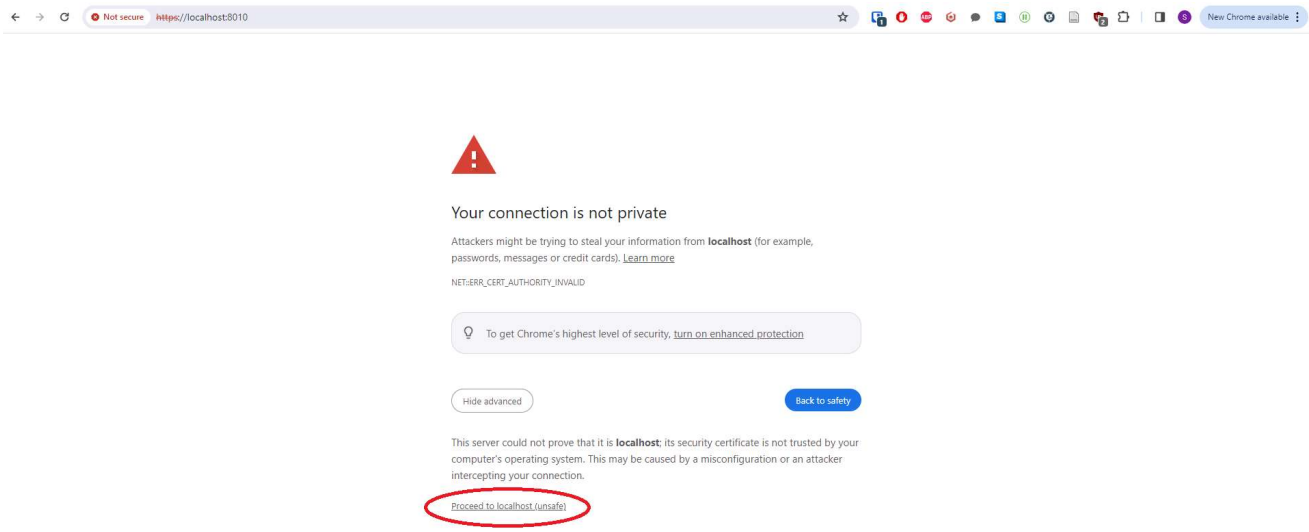
If successful, starting the server will look something like:

```
Added RoomServer port 8009
Added RoomServer port 8010
Added status server on port 8011
```

Self-signed certificates are unlikely to be accepted by default on most browsers. You will need to visit,

```
https://localhost:8010
```

And agree to proceed, before that browser will establish Ubiq connections.



Using VSCode

If you would like to run the server in VSCode, open the `Node` Folder in VSCode, and create the following launch configuration,

```
{
    "type": "pwa-node",
    "request": "launch",
    "name": "Launch Server",
    "skipFiles": [
        "<node_internals>/**"
    ],
    "env": {"NODE_OPTIONS": "--loader ts-node/esm"},
    "program": "${workspaceFolder}\\app.ts",
    "console": "integratedTerminal"
}
```

Status Module API Keys

If you want to limit access to the sensitive status module pages/APIs, you need to specify one or more API keys, for example:

```
{
    "status":{
        "apikeys": ["bf692145-80b1-40da-ba53-9c9dfdf7a5a7"]
    }
}
```

You can copy this snippet into a new `config\local.json` file, or add your keys into the existing empty `apikeys` member in `config\default.json`.

Alternatives ways to start the server

The server can also be started with:

```
node --loader ts-node/esm app.ts
```

or

```
NODE_OPTIONS="--loader ts-node/esm"
node app.ts
```

This is because server project is written in TypeScript and is set up to use the Node ESM Loaders feature to execute TypeScript (`.ts`) files directly without transpilation.

This requires the correct loader to be specified, which is done using the `--loader` parameter, either each time Node is started, or through the `NODE_OPTIONS` environment variable. (`npm start` is simply an alias for `node --loader ts-node/esm app.ts`).

XR Interaction

Ubiq includes a straightforward XR interaction framework. This supports high level actions such as *Using* and *Grasping* 2D and 3D objects, as well as interacting with the Unity UI system.

Ubiq is not dependent on its own interaction system, and it is expected users may utilise the Unity XR Interaction Toolkit, MRTK, VRTK or another system for advanced functionality.

The Ubiq system is intended to support common XR requirements however, while being very simple to use, and transparently cross-platform. All the Ubiq Samples are created with the Ubiq interaction system.

Cross Platform Support

The Ubiq interaction system is designed to work on both the desktop and in XR. This is achieved by maintaining two sets of input processing Components, that only respond to keyboard & mouse interactions, and XR controller interactions, respectively.

These Components are designed to co-exist on one set of GameObjects, allowing the same Player Prefab to be used for desktop and XR applications with no change.

For the interactables - 3D objects and 2D controls - identical events are recieved regardless of the source, so user code will work both for XR and the desktop transparently.

3D Interaction

Interacting with 3D objects is *action based*. Users can *Use* or *Grasp* objects. What these actions do is entirely up to user code.

For example, users could Use a button which spawns an object or turns on a light. They could Grasp a box which attaches to their hand, or a door which swings around an axis.

To implement these behaviours, Components implement `IUsable` or `IGraspable`. They will then recieve callbacks to `Use()` / `UnUse()` and `Grasp()` / `Release()`, respectively.

In XR, Players Use or Grasp objects by putting their controllers on an object and using the Trigger and Grip buttons. On the Desktop, users can use the cursor to click on objects with the Left or Middle mouse buttons.

Components implementing `IUsable` and `IGraspable` must be attached to objects with Colliders, though they do not need a RigidBody.

Hands

The methods of `IUsable` and `IGraspable` are passed `Hand` instances. `Hand` represents an entity in 3D space (a `MonoBehaviour`) that can interact with other 3D objects.

Other than existing in 3D, the `Hand` type is very abstract. It mainly exists to provide a 3D anchor, and to allow Components to distinguish between different controllers. A `Hand` does not have to have any physical presence (a RigidBody, or Collider).

Implementations of `IUsable` and `IGraspable` should be prepared to be used or grasped by multiple `Hand` instances simultaneously.

Graspers and Users

Calls to the `IUsable` and `IGraspable` implementations are made by the `UsableObjectUser` and `GraspableObjectGrasper` Components. These rely on the `HandController` Component, which implements the XR controller tracking and input processing based on the Unity XR namespace. There are desktop equivalents of `UsableObjectUser` and `GraspableObjectGrasper`.

These references are to the concrete types and so it is not currently possible to use `UsableObjectUser` or `GraspableObjectGrasper` with custom hand implementations. Though it is possible to re-implement them and pass a custom `Hand` subclass to the `IUsable` and `IGraspable` implementations.

2D Interaction

The Ubiq XR interaction integrates with Unity's UI system. Players can raycast from their hands to interact with Unity Canvases and controls.

To enable Ubiq XR interaction with a Canvas, add the `XRUICanvas` component to it. Once this Component is added users can interact with the Unity controls using raycasts from the controllers, or the mouse cursor on the desktop.

When using the `XRUICanvas` an `EventSystem` is no longer required. Cameras are not required either on World Space Canvases, allowing them to be declared in Prefabs and instantiated dynamically.

Raycasters

The 2D and 3D interaction mechanisms are separate. UI interaction is performed through UIRaycasters. There are XR and Desktop raycasters (`XRUIRaycaster` and `DesktopUIRaycaster`). Instances of both are attached to the sample Player Prefab.

Player Controller

Users move through the world using Player Controllers. The Samples contain a Player Prefab with a camera and two hands. Player Controllers can move linearly or teleport. Interaction always occurs through a `Hand` instance, so a Player Controller is not technically necessary.

Introduction

Networked VR applications require different types of logging, such as:

1. Debug Logs
2. Experiment Logs
3. Network Traces

1. Refers to logging expected and exceptional events that occur during a regular session. The purpose is post-hoc debugging of high-level application code.
2. Refers to logging application-specific data, such as measurements or questionnaire responses for an experiment.
3. Refers to captures of network traffic to investigate reproducible low-level netcode bugs.

(1) & (2) are handled by Ubiq's Event Logging system. (3) has distinct performance implications, so is handled seperately.

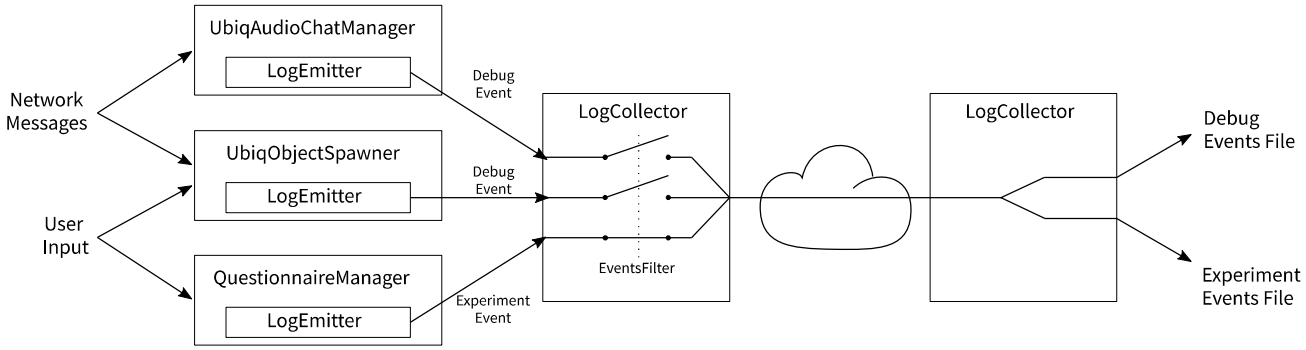
Use Case

The Event Logging System is for collecting low or medium frequency events from multiple peers. The Event Logging system can log both Ubiq and third-party events, which can then be extracted and analysed.

Events are discrete, but otherwise have very few restrictions. It is up to the user to ensure that event logging in their application doesn't negatively affect performance.

Overview

Events are generated by `LogEmitter` instances placed throughout the application. Events generated by these components are passed to a local `LogCollector` instance. The `LogCollector` then writes them to disk (or database, or other endpoint), or forwards them to another `LogCollector` that will.



LogEmitter

`LogEmitter` instances are lightweight objects that the application uses to log events. Calls to a `LogEmitter` are expected to be placed throughout the system persistently, rather than gated with pre-processor defines.

The most common types of event logger are the `ContextEventLogger`, which is designed to work with Components that have a `NetworkContext`, and the `ExperimentEventLogger`, designed for logging measurements in experiments.

```
public class VoipPeerConnectionManager : MonoBehaviour
{
    private ContextLogEmitter debug;

    private void Start()
    {
        context = NetworkScene.Register(this);
        debug = new ContextLogEmitter(context);
    }

    public void ProcessMessage(ReferenceCountedSceneGraphMessage message)
    {
        var msg = JsonUtility.FromJson<Message>(message.ToString());
        switch (msg.type)
        {
            case "RequestPeerConnection":
                debug.Log("CreatePeerConnectionForRequest", msg.objectid);
                break;
        }
    }
}
```

The snippet above demonstrates the creation and use of a `ContextEventLogger`. The `VoipPeerConnectionManager` declares the `ContextLogEmitter` `debug` and initialises it with a `ContextEventLogger` once a context has been created. The `Log` method can then be called to log the receipt of a specific message.

Events are recorded as Json objects, for example:

```
{
"ticks":637794978937526996,
"peer":"aa9a2c8c-0c6f11c7",
"type":"Ubiq.Voip.VoipPeerConnectionManager",
"objectid":"aa9a2c8c-0c6f11c7",
"componentid":50,
"event":"CreatePeerConnectionForRequest",
"arg1":"9020d814-45c41c19"
}
```


`LogEmitter` instances attach to a single `LogCollector`. The emitter constructors find the closest `LogCollector` automatically.

`LogEmitter` methods can be safely called from outside the Unity main thread. They should not be called from outside CLR threads however.

`LogEmitter` instances are designed to have zero overhead when logs are not actually written. The `Log` method has many overloads to avoid boxing, and serialisation only runs when logging is on. Logs are only written when there is a listening `LogCollector` in the scene. If the `LogCollector` is disabled, non-existent, or the Collector's EventFilter ignores the emitter's event type, the emitter will not do anything.

It is encouraged to make as many `LogEmitter` instances as needed. Individual emitters are simple, with few options. Use multiple `LogEmitter` instances within a class to get fine-grained control over logging, for example different log levels.

Events and Filters

A `LogEmitter` can tag events with a flag. A number of flags are pre-defined in `Ubiq.Logging.EventType` enum. The underlying type is an `sbyte`, so additional values can be used in an unchecked context.

```
[Flags]
enum EventType
{
    Application = 1,
    Experiment = 2,
    Debug = 4,
    Info = 8
}
```

A `LogCollector` can be set to ignore one or more of these types with its `EventsFilter` member. If a flag is *set* the event is logged. If it is *unset* the event is *not* logged. Filtering applies to local events. `LogCollectors` will forward and write all external events regardless of the filter. Filtered events are not cached, but lost permanently.

`LogCollector` writes events of different types to different streams/files. Prefixes are defined for the entries in `Ubiq.Logging.EventType`. Other tags will display as their numerical equivalent.

The `LogCollector` `EventsFilter` member and the `Tag` member of any `LogEmitter` can be changed at any time.

`ComponentEventLogger` and `ContextEventLogger` have their type set to `Debug` by default. Convenience classes are defined for `ExperimentLogEmitter` and `InfoLogEmitter`.

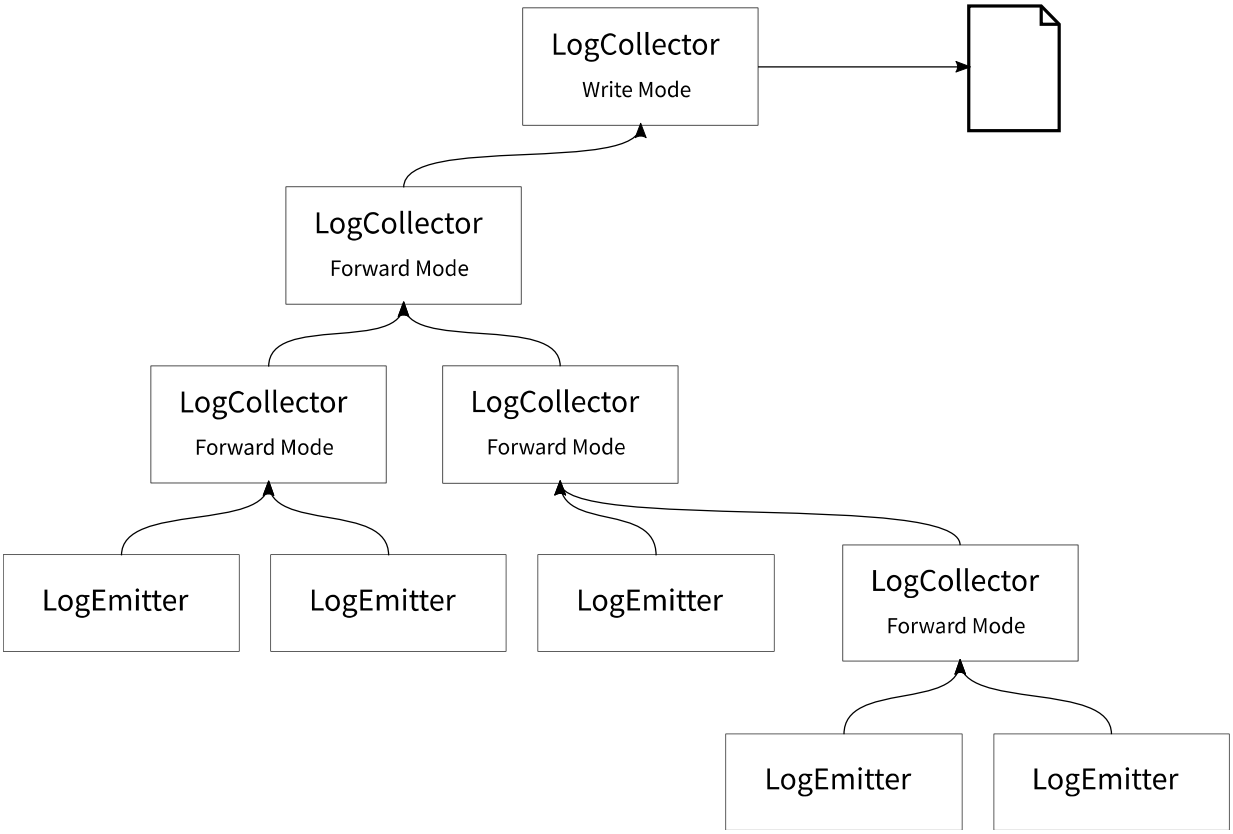
LogCollector

`LogCollector` instances receive events from local `LogEmitter` instances and remote `LogCollector` instances. They do one of three things with these events:

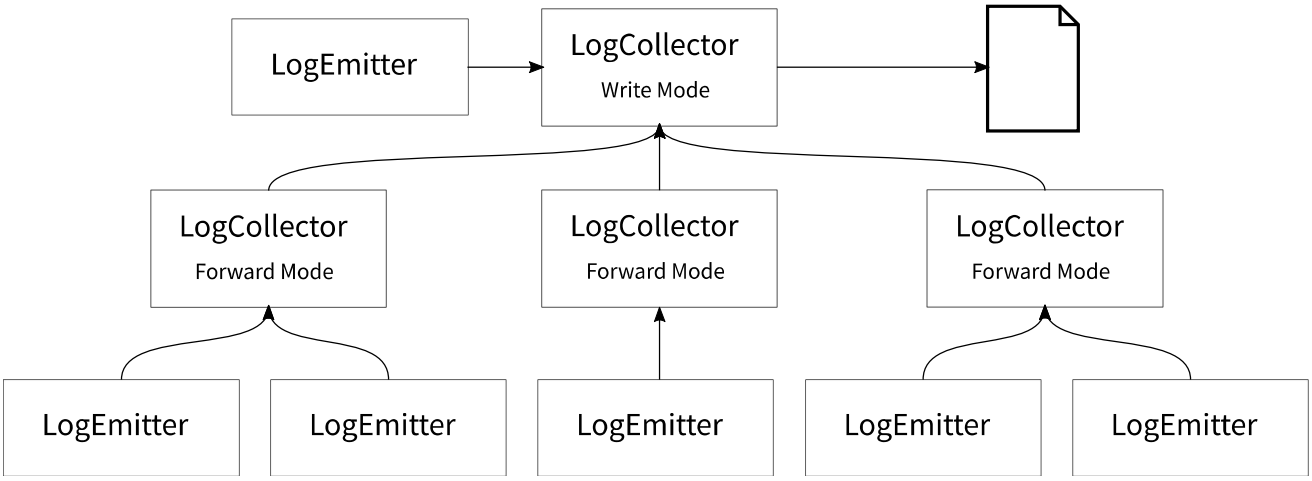
1. Cache them (Buffering Mode)
2. Write them to a local file (Writing Mode)
3. Forward them to another LogCollector (Forwarding Mode)

The behaviour depends on the value of the `destination` member, which determines where the events should go.

`LogCollector` instances can be organised this way into complex tree arrangements.



However, the expected use case is that Peers in a Group forward events directly to one Collector, that also writes them to disk.



`LogCollector` writes events to files in the Persistent Data folder of whatever platform it is running on. It does this through a set of `Ubiq.Logging.LogCollector.IOutputStream` instances (one for each event type, created on demand).

The filenames include the event type and a timestamp.

Buffering

When a `LogCollector` is not Forwarding or Writing, it will cache or Buffer all events received (whether local or remote). If a `LogCollector` was previously Writing or Forwarding and stops, it will resume Buffering. When a `LogCollector` begins Writing or Forwarding, it will start with all the events in its buffer.

Events only leave the buffer when being sent to a safe destination (another `LogCollector`, or a file). In this way logging is lossless, regardless of the initialisation order and even when changing the Active `LogCollector`.

However, events that leave the buffer are gone permanently. If a `LogCollector` writes events, then another `LogCollector` becomes active at a later time, the second will only output events received after it became active. The file from the first `LogCollector` must be retrieved out-of-band to have the complete record of events.

This can be avoided by ensuring only one `LogCollector` becomes active during the lifetime of the Peer Group.

Encodings

Events are written as Json (Utf8 strings).

`LogCollector` attempts to make the output files Json compliant, by placing events in a top-level Json Array, with comma seperated entries. If the file is being read live however, or the process terminates unexpectedly, the file will not have the closing bracket ( ]).

In this case the application reading the files should either add the bracket to the end of the stream or perform its own tokenisation.

Changing the LogCollector

Only one `LogCollector` in a Peer Group may be in Writing Mode at any time.

The Active Collector can be changed by calling `StartCollection` on the new Collector. This can be done at any time.

When the Active Collector changes, the Peers must agree on the new Collector. This is a Distributed Agreement problem. `LogCollector` solves this using the Global Snapshot method [1].

The new Collector broadcasts a message with a logical clock value and atomically updates its cut state. When Peers receive the message, they compare the clock against their own. If it is greater, they update their cut state, otherwise the message is ignored. If a Collector is attempting to become Active, and receives a message with the same clock value as its own, a collision has occured. In this case, the Collector re-initialises its clock to a random value, and re-transmits its state. If a message is received with a greater clock value, it updates its own State as if it were any other Peer.

This algorithm assumes that Peers are fully connected, and message passing is reliable, which is true in Ubiq. The logical clock ensures that all Peers converge on the same State, regardless of the order the messages were recieved in. The Peers have converged when all messages have been delivered.

When the Active Collector leaves the Peer Group, the Clock is reset.

Limitations

As long as Peers remain connected, the system is lossless, even during convergence.

However, during convergence different events may flow to the old and new Active Collectors at the same time, resulting in events being spread between the sets of log files. This will occur until convergence. There is no upper bound on this time, and there is no mechanism to check if convergence has been achieved, so `StartCollection` should be called with care.

To surrender position as the Active Collector, a Collector may call `StopCollection`, however the time between this and the Peers converging to null is again indeterminate.

When the Active Collector changes, the delay between the previous Collector receiving the new state, and other Peers in the Group receiving the new state, may cause it to behave as a relay for a time.

Process Failure

If the Active Collector process fails, log events will be lost until the system converges on a new Collector or null.

If the Collector was part of a Room, the Rooms system can detect disconnection in some cases, and Collectors will update their cut state independently. The Collectors in this case will cache until a new Collector volunteers itself.

If a process that was not the Active Collector fails, then that process will simply not emit events. If that process was previously an Active Collector, it is still possible to lose events if the Collector was acting as a relay when the process failed.

For collecting data such as experimental logs, it is recommended to only ever have one collector on a process that can be monitored. For example, the `logcollectorservice`.

Verification

The `LogCollector` method `GetBufferedEventCount` returns the number of Events currently buffered. As `LogCollector` is multi-threaded, this count may change even while it is being returned. However, if it is known, for example, that an application will not generate any new Events of a particular type, it can be used to check whether all of those events have finished writing.

The `LogCollector` method `Ping` will ping the Active Collector.

All Log messages are delivered in order. Therefore these mechanisms can be used together to verify that all log messages for a particular application have been delivered successfully before it exits.

To do this, an application could:

1. Write the last Event, e.g. a Questionnaire result.
2. In the same thread, wait until the number of buffered events of that type is zero.
3. Ping the Active Collector

When a response is receieved, the Event, and all preceeding it, will have been successfully delivered and the application can safely exit.

This protocol is implemented in the `WaitForTransmitComplete` method.

This does not protect against process failure of an intermediary `LogCollector` however, a condition which is irrecoverable.

Reliability

In order for `WaitForTransmitComplete` to confirm that an event has been successfully delivered, the integrity of the LogCollector processes must be fully visible. One way to achieve this is to ensure only one LogCollector is ever active, and that that LogCollector never forwards. In this case, if the Ping is received, then the logs must also be successfully delivered as they cannot have taken any other path to the collector.

`LogCollector` will keep track of this by default, and warn any caller of `WaitForTransmitComplete` if this is the case.

Analysis

A `LogCollector` outputs a stream of structured logs in compliant Json. These logs can be fed to a stack like the ELK, processed with third-party tools like Matlab or Excel, or processed programmatically on platforms such as Python.

See the Analysis section for examples of how to process the logs.

[1] Kshemkalyani, A. D., & Singhal, M. (2008). Distributed Computing: Principles, Algorithms and Practice. Cambridge University Press.

Analysis

The Event Logger outputs *structured* logs, as Json objects. These can be processed on any platform that can read Json files.

A sample log file is shown below.


```
[
{"ticks":637799309335620180,"peer":"088edbc1-1d1f09b5","type":"Ubiq.Messaging.NetworkScene","ever
{"ticks":637799309384207356,"peer":"088edbc1-1d1f09b5","type":"Ubiq.Voip.VoipPeerConnectionManage
{"ticks":637799309384277353,"peer":"088edbc1-1d1f09b5","type":"Ubiq.Voip.VoipPeerConnectionManage
{"ticks":637799309087959820,"peer":"26a6ee77-3cec71fe","type":"Ubiq.Messaging.NetworkScene","ever
{"ticks":637799309303272560,"peer":"26a6ee77-3cec71fe","type":"Ubiq.Voip.VoipPeerConnectionManage
]
```


In this example, two peers - a desktop PC (Unity Editor) and an Oculus Quest - join a room. The `NetworkScene` and `VoipPeerConnectionManager` both log events.

Some Json members are defined by the `Emitter` type. For example, the `ContextLogger` writes the `objectid` of the context passed to it on creation. The `arg` members correspond to those passed to the `Log()` method. All entries include a timestamp and the Id of the Peer that generated the log. Timestamps are given in .Net Ticks.

Python

Python can be used to analys logs programmatically. The Jupyter notebook below shows how to import and process logs using Pandas, a powerful data analysis library for Python.

Print the version to check that Pandas is available. If the module is not found, install pandas with `pip install pandas`.

```
In [ ]: import pandas as pd
        print(pd.__version__)
```

```
1.4.0
```

Import the Json file with read_json. This will make a Pandas *DataFrame* (a table).

```
In [ ]: df = pd.read_json("Debug_Log.json")
```

Print the first five rows to see what the table looks like.

```
In [ ]: df.head(5)
```

Out[ ]:

	ticks	peer	type	event	arg1	arg2	arg3	objectid	componentid
0	637799313741476033	2990c448-6701d991	Ubiq.Messaging.NetworkScene	Awake	DESKTOP-F1J0MRR	System Product Name (ASUS)	f73fe01b1e21031d49274a1491d1d6b5714c92e9	NaN	NaN
1	637799313890915697	2990c448-6701d991	Ubiq.Voip.VoipPeerConnectionManager	CreatePeerConnectionForPeer	0b6034cb-5c980872	21119b9e-9028aafa	NaN	2990c448-6701d991	50.0
2	637799313890975713	2990c448-6701d991	Ubiq.Voip.VoipPeerConnectionManager	RequestPeerConnection	0b6034cb-5c980872	21119b9e-9028aafa	NaN	2990c448-6701d991	50.0
3	637799313891015701	2990c448-6701d991	Ubiq.Samples.NetworkSpawner	SpawnObject	2	b0edec0e-fcf7792a	True	7725a971-a3692643	49018.0
4	637799313891055695	2990c448-6701d991	Ubiq.Samples.NetworkSpawner	SpawnObject	2	43b53edd-7c900c8f	False	7725a971-a3692643	49018.0

We can use Pandas to filter and process the structured logs. Use Unique to find all the event types seen during the session.

```
In [ ]: df.type.unique()
```

```
Out[ ]: array(['Ubiq.Messaging.NetworkScene',
               'Ubiq.Voip.VoipPeerConnectionManager',
               'Ubiq.Samples.NetworkSpawner'], dtype=object)
```

Pandas can perform vector comparisons, and filter DataFrames by row indices. Select all the SpawnObject events.

```
In [ ]: df[df.event == "SpawnObject"].head(5)
```

Out[ ]:

	ticks	peer	type	event	arg1	arg2	arg3	objectid	componentid
3	637799313891015701	2990c448-6701d991	Ubiq.Samples.NetworkSpawner	SpawnObject	2	b0edec0e-fcf7792a	True	7725a971-a3692643	49018.0
4	637799313891055695	2990c448-6701d991	Ubiq.Samples.NetworkSpawner	SpawnObject	2	43b53edd-7c900c8f	False	7725a971-a3692643	49018.0

- See the Notebook in full
- Download Jupyter Notebook
- Download Example Log File

Excel

Structured event logs are amenable to being viewed in a table. Microsoft Excel PowerQuery can import Json files and load events into Excel Worksheets.

To do this:

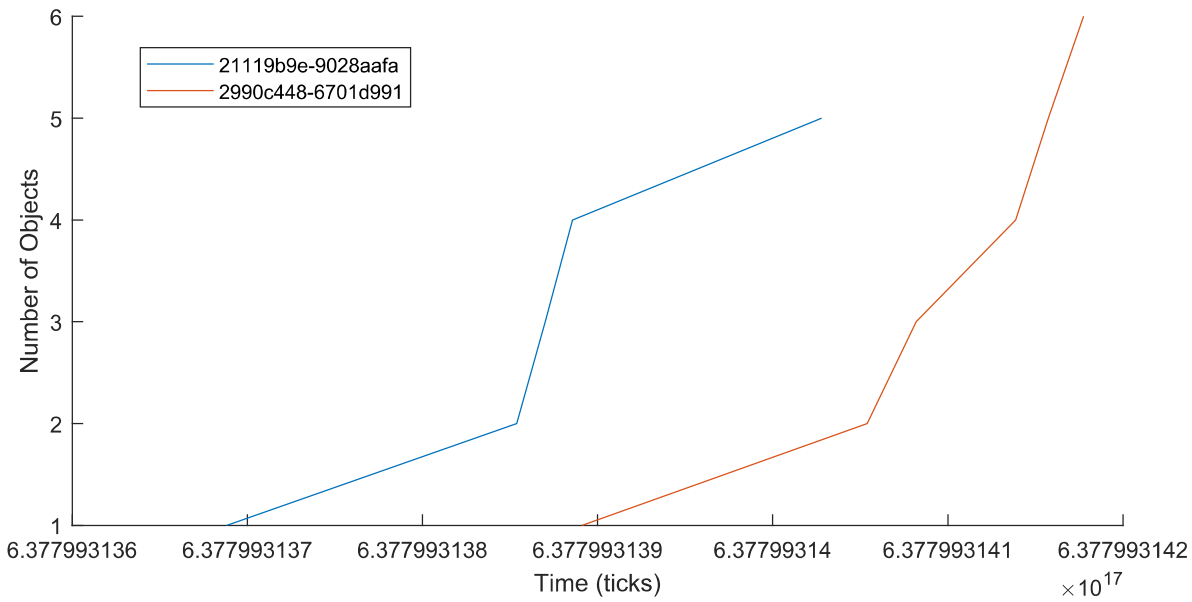
1. Open a new `Workbook`
2. From the `Data` tab, choose `Get Data` -> `From File` -> `From Json`
3. Open the log file, for example `Application_log_2021-04-23-10-56-03_0.json`
4. Select the `List` header and click `Convert To Table`. This will instruct Excel to treat each entry as a row.
5. Leave the Default Values in place and Click `OK`. The `View` will now appear as a `Column`.
6. Use the button in the top right to add the `Expand Column` step. This will split each record into a set of columns. Make sure to click `Load More...` if visible to ensure you get every possible field in the table.
7. Click `OK`
8. Click `Close & Load` to build your table.

You can now order by Ticks, and filter columns such as Events.

Matlab

Like Python, Matlab can load Json using the jsondecode function.

```
% Read the text file and use jsondecode to produce a cell array of
% structures.

events = jsondecode(fileread("Debug_Log.json"));

% The structures will have different fields, so we must use loops to filter
% them before they can be combined into a single struct array or table.

% Below, find all the events of type SpawnObject, and combine them into a
% new array.

spawn = [];

for i = 1:numel(events)
   % The curly braces access the contents of the cell i, which is the
   % struct itself.
   s = events{i};
   if categorical(cellstr(s.event)) == categorical("SpawnObject")
       spawn = [spawn; s];
   end
end

% Convert the new array into a table
T = struct2table(spawn);

% Use the table to change the type of the sceneid column so we can easily
% split the events by which peer they are from.
T.peer = categorical(T.peer);

% Filter the events to keep only those emitted by the Peer that initiated
% the spawn
T = T(T.arg3,:);

% Plot the number of objects spawned over time, by each Peer
figure;
hold all;
peers = unique(T.peer);
for p = peers'
   spawned = T(T.peer == p,:);
   plot(spawned.ticks,1:size(spawned,1));
end

xlabel("Time (ticks)");
ylabel("Number of Objects");
legend(peers);
```



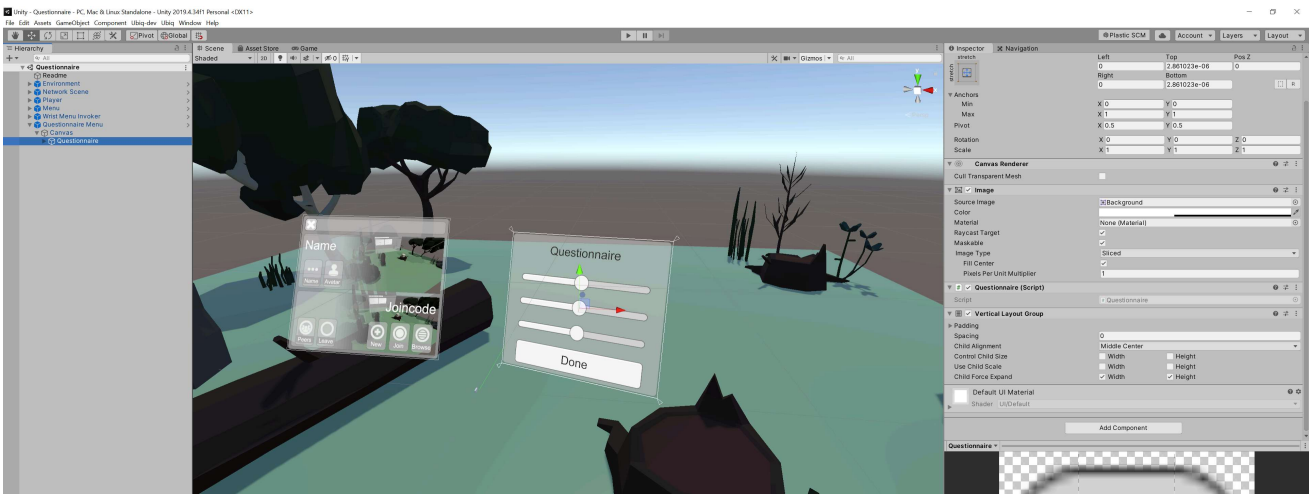
- Download Matlab Source
- Download Example Log File

Questionnaire

The Questionnaire Sample (Samples/Single/Questionnaire) shows how the Event Logging System may be used to collect questionnaire responses.

This scene contains a panel with an example Component, `Questionnaire` attached to it. The Component iterates over all `Slider` instances under its `GameObject`, and uses an `ExperimentLogEmitter` to write their values when the user clicks *Done*.

```
public class Questionnaire : MonoBehaviour
{
    LogEmitter results;

    // Start is called before the first frame update
    void Start()
    {
        results = new ExperimentLogEmitter(this);
    }

    public void Done()
    {
        foreach (var item in GetComponentsInChildren<Slider>())
        {
            results.Log("Answer", item.name, item.value);
        }
    }

}
```



The Questionnaire can be completed locally in Play Mode. Alternatively, the scene can be run remotely, and the experimentor in the Editor can join the same room as the remote copy. In either case, the experimentor in the Editor can click *Start Collection* on the `NetworkScene` > `Log Manager` > `LogCollector` to recieve the Questionnaire results.

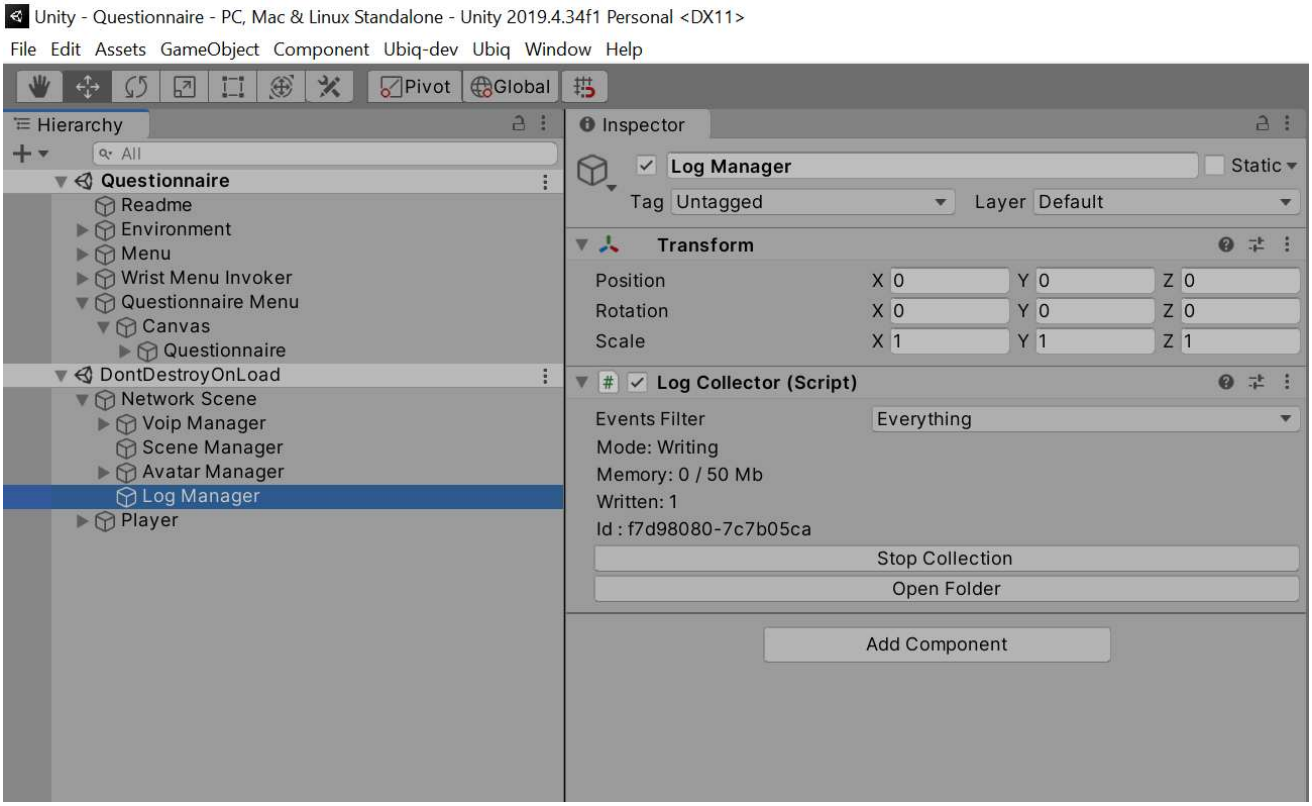
The experimentor can click *Start Collection* before or after the questionnaire has been completed, and the participant can complete the Questionnaire before or after joining the room. In all cases the results will be receieved correctly.

Sample Output

Below is the resulting *Experiment* log file from an application built with the Questionnaire scene.


```
[
{"ticks":637795003787005516,"peer":"f7d98080-7c7b05ca","event":"Answer","arg1":"Slider 1","arg2":
{"ticks":637795003787045512,"peer":"f7d98080-7c7b05ca","event":"Answer","arg1":"Slider 2","arg2":
{"ticks":637795003787045512,"peer":"f7d98080-7c7b05ca","event":"Answer","arg1":"Slider 3","arg2":
]
```


The Questionnaire was filled in on an Oculus Quest, after joining the same room as a user running the same scene in the Unity Editor. As soon as the Questionnaire was completed, the Unity Editor user could find the *Experiment* log by clicking the *Open Folder* button of the `LogCollector` Component in the Editor.



Since no filters were set up on the `LogManager`, a *Debug* log for the session is also created in the same folder.

```
[
{"ticks":637795043778071253,"peer":"cbc6f82b-24ec48b3","type":"Ubiq.Messaging.NetworkScene","ever
{"ticks":637795044161926844,"peer":"cbc6f82b-24ec48b3","type":"Ubiq.Samples.NetworkSpawner","obje
{"ticks":637795044161966844,"peer":"cbc6f82b-24ec48b3","type":"Ubiq.Voip.VoipPeerConnectionManage
{"ticks":637795044162026839,"peer":"cbc6f82b-24ec48b3","type":"Ubiq.Voip.VoipPeerConnectionManage
{"ticks":637795044162066856,"peer":"cbc6f82b-24ec48b3","type":"Ubiq.Samples.NetworkSpawner","obje
{"ticks":637795043937235620,"peer":"4641730f-148936d7","type":"Ubiq.Messaging.NetworkScene","ever
{"ticks":637795044080181550,"peer":"4641730f-148936d7","type":"Ubiq.Samples.NetworkSpawner","obje
{"ticks":637795044152929360,"peer":"4641730f-148936d7","type":"Ubiq.Voip.VoipPeerConnectionManage
{"ticks":637795044153061850,"peer":"4641730f-148936d7","type":"Ubiq.Samples.NetworkSpawner","obje
]
```


Graceful Exit

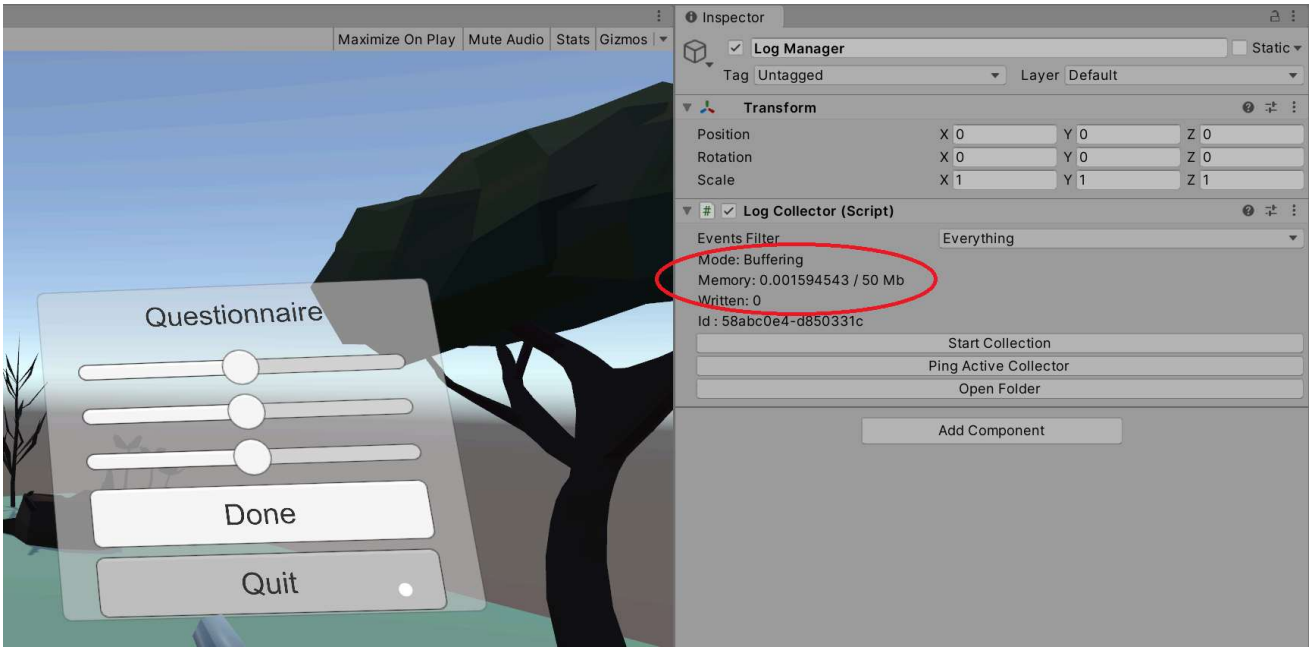
The Questionnaire Panel also has a Quit button. This button makes use of the LogCollector WaitForTransmitComplete method to quit the application, but only when the questionnaire results have been successfully delivered.

```
        public void Quit()
        {
            LogCollector.Find(this).WaitForTransmitComplete(results.EventType, ready =>
            {
                if(!ready)
                {
                    // Here it may be desirable to to save the logs another way
                    Debug.LogWarning("ActiveCollector changed or went away: cannot confirm logs h
                }
#if UNITY_EDITOR
                UnityEditor.EditorApplication.isPlaying = false;
#else
                Application.Quit();
#endif
            });
        }
```

The callback will only be called once the Experiment logs have left the local LogCollector.

Enter Play Mode, click Done, then click Quit.

Since the LogCollector is buffering, the application won't exit because the Log Events are still in Memory. Click Start Collection and the application will immediately write the logs and exit.



Try as well again entering Play Mode, and clicking Done and Quit. This time however join a Room with a LogCollector on another Peer. As soon as that Peer's LogCollector is Started, the Questionnaire will quit.

If the LogCollector on the other Peer is already active (e.g. in the case of a running logcollectorservice), the application will quit almost as soon as it joins the Room.

Event Logging Unit Tests

The LoggingDiagnostics scene (Samples/Single/Logging) performs stress testing of the Logging system.

This scene contains a Component, LoggingDiagnostics, and corresponding UI in the scene to control it outside the Editor.

The Component creates a number of LogEmitters that generate arbitrary, deterministic events. Additionally, the Component will instruct the Peer's LogCollector to become the primary Collector at random.

When multiple instances of this Peer are running, each will create and forward events to eachother as the primary mode moves between them.

As events are deterministic, the log files can be checked once all Peers have shut down to verify that all events from all peers were logged correctly, despite changing the primary Collector and starting at different times.

The function to check the logs is also in the LoggingDiagnostics Component and is invoked in the Inspector in the Editor.

This code assumes that all test Peers were running on the same machine, as it will check all files in the systems persistent data path.

If some Peers were running elsewhere (e.g. on an Oculus Quest), those log files should be copied to the persistent data path first so their events will be included.

The user should quit the peers using the Finish & Exit UI button in the scene. Pressing this button will shut down all Peers in the Peer Group running the LoggingDiagnostics scene. This mechanism must be used as it delays the shutdown for a few seconds after the last log messages have been transmitted, to ensure enough time for the data to be logged and preventing false negatives when checking the data for integrity.

Log Collector Service

The LogCollectorService is an example NodeJs application that joins a room and writes all the Experiment Log Events (0x2) in that Room to disk.

The sample is located in the `Node/samples/logcollectorservice` directory.

Use Case

The Log Collector Service can be used to automatically collect data from self-directed experiments.

To do this, an experimentor would create a build that automatically joined a pre-defined Room. Additionally, the Log Collector Service would be started on a server and configured to join the same Room.

As participants ran their builds and completed the experiment, they would all join the same room and forward their events to the `LogCollector` running in the Log Collector Service, which would write those events to disk on the server.

Questionnaire Scene

A good way to try the Log Collector Service is to use the *Questionnaire* Sample (Samples/Single/Questionnaire).

Add the Join Room Component to the NetworkScene, and set the Room GUID to the same one as in the `logcollectorservice` app.

Be sure to generate a new GUID to avoid collisions with others potentially trying the same demonstration.

Then, start and stop the Questionnaire scene, submitting the Questionnaire a few times each run.

In the logcollectorservice directory, a number of log files should be created, with the Ids that the Peer took on each time it started.

Configuration

The logcollectorservice application is configured by changing the source code of app.js. The two variables that are likely to change are the log event type, and the room GUID.

Serialiser

The logging functionality uses a custom Json serialiser that facilitates building Json objects across multiple function calls.

This is based on neuecc's Utf8Json, but with modifications to track memory usage and remove code generation requirements.

The Utf8Json serialiser is in the `Ubiq.Logging.Utf8Json` namspace. It is not recommended to use the serialiser for purposes other than logging; import an unmodified version of the library separately instead.

Formatters

Libraries such as Utf8Json typically have methods that serialise and deseralise specific types by sequentially reading and writing tokens to and from streams. (In this case, the tokes are read and written using the `JsonReader` and `JsonWriter` structures.)

Utf8Json finds the appropriate method to use using `FormatterResovler` classes. These classes return a cached `Formatter<T>` class, which is an object with two methods to read and write objects of type `T` as Json.

The included version of Utf8Json includes formatters for a number of known types, including all the basic primitives, and enums. Enums are serialised as names.

Code Generation

To serialise types that do not have an explicit formatter defined, libraries such as Utf8Json usually build serialisation methods at runtime using code generation. This is not supported on platforms that use IL2CPP however.

To avoid code generation, unknown types are serialised by the Unity JsonUtility and embedded as objects.

Resolvers and Formatters

When a type is serialised, Utf8Json will use the `DefaultResolver` to find a formatter. The `DefaultResolver` is defined in the `JsonSerializer` class as a static member and returns a `StandardResolver`, a type of composite resolver. This resolver will search each resolver

registered to it in turn, and return the first `Formatter` that matches the type. The `StandardResolver` includes formatters for the built-in types, and the dynamic formatter fallback.

Caching

Utf8Json makes common use of the following design pattern.

```
public IJsonFormatter<T> GetFormatter<T>()
{
    return FormatterCache<T>.formatter;
}

static class FormatterCache<T>
{
    public static readonly IJsonFormatter<T> formatter;

    static FormatterCache()
    {
        formatter = (IJsonFormatter<T>)BuiltinResolverGetFormatterHelper.GetFormatter(typeof(T));
    }
}
```

This snippet leverages the behaviour of generics in C# to replace formatter references in code, without using code generation. In C#, when a generic type is first constructed, the runtime will produce the concrete type and substitute it in the appropriate locations in the MSIL. The static constructor is called before the formatter is referenced for the first time.

That is, the generic FormatterCache type is replaced in the MSIL and the formatter member it returns is resolved on demand (when the `FormatterCache<T>` is first constructed).

Memory Management

The Utf8Json namespace manages its own global memory pools to minimise GC allocations. It does not track memory usage directly however.

Instead, `LogCollector` instances track how many bytes of pooled memory they have in their queues at any time, and use this to control whether new events are buffered or dropped.

Memory is rented from the pool on demand by JsonWriter objects created by `LogEmitter` instances. Outstanding memory is returned to the pool when a JsonWriter is disposed. JsonWriters are disposed by the `LogCollector` they are fed to, either after being copied for transmission or discarded when the buffer reaches capacity. `LogEmitter` instances only create JsonWriters if a `LogCollector` has been registered to recieve (and dispose of) the completed object.

Introduction

Ubiq is built around exchanging exchanges between instances of networked Components. A message sent by one Component is received by all other similar Components (multicasting).

This logical behaviour is decoupled from the actual connection architecture, which is built around unicast TCP connections intended to operate over the internet. Messages are variable size with binary payloads. All messages passed by Ubiq have the same header.

The Medium is the Message

Ubiq delivers individual messages between instances of networked objects (most commonly Unity Components). How messages get from Component A to Component B is the responsiblity of the Messaging Layer, and is abstracted from the developer.

Network Components only recieve messages addressed to them directly, so they can rely on knowing the type of the message by virtue of having recieved it.

The expected programming model is that Components implement send and recieve functionality in the same script, which is also where the format is defined. By the time a message reaches a Component, it is received as the exact raw sequence of bytes sent by the Component's counterpart. Individual components choose the best serialisation method and transmission frequency for their use case.

Scene Graph as a Bus

Ubiq closely matches Unity's programming model. Since user code is placed in Components in Unity, networked objects in Ubiq are also Unity Components.

Each Component has a Network Id which uniquely identifies that instance on the Peer. The Id is shared by equivalent instances on other Peers.

All Components are associated with one `NetworkScene`. This is the networking equivalent to the root of the scene graph. Components find their Network Scene based on the distance through the scene graph to the closest instance of a `NetworkScene` Component.

Fan-Out at the Network Layer

When a Component transmits a message, it is recieved by every other Component with the same address (Network Id). This is the case regardless of how many instances there are with the same combination. Fanning out - or multicasting - the message is done at the network layer. Individual Components don't know how many counterparts they have.

Network Ids

Network Ids are analogous to the *instance Id* of a Unity Component. This is how Component instances address eachother.

A Component type may exist on one or more GameObjects, or it may exist multiple times on one GameObject. Each of these instances within the Unity Scene or process will have a different *instance Id*, and by default a different *network Id*.

So, if there are two copies of a Prefab in a Scene, they will talk to their respective copies, and only their respective copies, on other Peers.

Default Network Id

The default behaviour of Ubiq is to generate a Network Id for a Component when it calls Register(). The Id is based on the Component's address is in the scene graph. This is done predominantly by name.

If you have two scenes with the same hierarchy, and two GameObjects with the same name, and same networked Components, those Components will end up with the same Id. This is true even if the scenes are different Unity scenes, or if the scenes are on different devices.

There is some flexibility. For example, if you had two GameObjects with different names, their order in the scene graph could be inverted and the addressing would still work. You couldn't invert the order of GameObjects with the same name however, or put them in different branches of the same scene graph.

Network Id Gizmo

If you want to see the addresses of your Networked Objects, the NetworkIdGizmo Component can be used to show them at Runtime in the Scene View.

In the image above, `NetworkIdGizmo` is attached to an Avatar. The Avatar has one networked Component - `ThreePointTrackedAvatar` - with an Id of `8eef84fa-5d1ec5ee`.

Specifying a specific Network Id

The default behaviour is intended to be used where one Unity Scene is created for an application, and that same scene is used by all Peers. In this case, as there is only one scene the addresses between all Peers' Components will always be the same and there is no need to worry about keeping them synchronised.

However, there are many cases where you may want to directly control the Network Id. For example, you may want to pre-share a Network Id to create a service. You may want to send messages from one Component to a Component with a different Network Id.

In these cases you can call `NetworkScene::Register(MonoBehaviour, NetworkId)`. This overload will register the Component with the specified Network Id, instead of generating one automatically, allowing you to use any communication pattern you like for your objects.

Generating Ids

You can generate a new, unique network Id by calling `NetworkId::Unique()`.

Network Context & Send

`Register` returns a `NetworkContext` object with a `Send` method. This method sends messages to the same Id as stored in the context (the one registered to the networked Component).

If you want to send messages to a different Network Id, you can call `NetworkScene::Send()` directly, which takes a message and a Network Id to send that message to.

Unique Ids

One of the most important considerations when generating Ids is the probablity of collisions.

Collision Probability

Each client must be able to generate new Ids independently, to avoid dependencies on a server in peer-to-peer situations, and to avoid introducing deadlocks with the Unity programming model (such as needing a server to respond before all `Start()` calls can finish).

The probability of collisions between two independently well-generated identifiers is the *Birthday Problem*. The probability of a collision is given by,

$$p = \frac{2^b!}{(2^b)^n(2^b - n)!}$$

On regular desktops, it is not possible to compute this for the typical numbers involved due to floating point quantisation. However there are approximations, such as the following which will work in Matlab,

- `p = 1 - exp(-(n^2)/(2*(2^b)))`

As the figure below shows, the probability of collisions depends on the number of objects in the scenario `n`, as well as the identifier length `b` (in bits). In Ubiq, the "scenario" is typically a room.

There are no rules defining acceptable chances of collisions. For example, OAuth 2 specifies a probability of less than `10e-160`, which is not achievable even by 128-bit UUIDs.

In the case of a typical Ubiq room, which due to voice chat limitations can support approximately 25 users, a 32 bit value would probably be adequate. The concept of a Room in Ubiq can change depending on how it is used however, so Ubiq tries to avoid making assumptions about the potential collision space. If rooms were considered to be floating regions, for example, then the number of potential users could be much higher.

It is challenging to grasp levels of chance at these scales. The table below by Jeff Preshing presents collision probabilities more intuitively. (The numer of *hash values* below is the number of Objects between all users that could potentially connect in a Ubiq scenario, be that a room, set of rooms, a server, shard etc.)

Number of **32-bit** hash values	Number of **64-bit** hash values	Number of **160-bit** hash values	Odds of a hash collision
77163	5.06 billion	1.42×10^{24}	1 in 2
30084	1.97 billion	5.55×10^{23}	1 in 10
9292	609 million	1.71×10^{23}	1 in 100
2932	192 million	5.41×10^{22}	1 in 1000
927	60.7 million	1.71×10^{22}	1 in 10000
294	19.2 million	5.41×10^{21}	1 in 100000
93	6.07 million	1.71×10^{21}	1 in a million
30	1.92 million	5.41×10^{20}	1 in 10 million
10	607401	1.71×10^{20}	1 in 100 million
	192077	5.41×10^{19}	1 in a billion
	60740	1.71×10^{19}	1 in 10 billion
	19208	5.41×10^{18}	1 in 100 billion
	6074	1.71×10^{18}	1 in a trillion
	1921	5.41×10^{17}	1 in 10 trillion
	608	1.71×10^{17}	1 in 100 trillion
	193	5.41×10^{16}	1 in 10^{15}
	61	1.71×10^{16}	1 in 10^{16}
	20	5.41×10^{15}	1 in 10^{17}
	7	1.71×10^{15}	1 in 10^{18}

Odds of a full house in poker
1 in 693

Odds of four-of-a-kind in poker
1 in 4164

Odds of being struck by lightning
1 in 576000

Odds of winning a 6/49 lottery
1 in 13.9 million

Odds of dying in a shark attack
1 in 300 million

Odds of a meteor landing on your house
1 in 182 trillion

Generating Random Identifiers

The collision probabilities above are only correct if identifiers are well-generated.

RFC 4122 descibes algorithms to generate 128-bit UUIDs. UUIDs are standardised identifiers on many network services. They are very popular as they can be independently generated quickly, while guaranteeing uniqueness and being amenable to storing, sorting and hashing. UUIDs can guarantee uniqueness beyond the probabilities discussed above because some versions include the use of MAC addresses, which *are* centrally registered (by the network card manufacturer) and which act as a "pre-shared secret".

RFC 4122 also presents an algorithm for version 4 which is based purely on random numbers. RFC 1750 discusses random numbers in more detail. Acquiring truly random numbers for identifier generation is challenging, as common computer systems rarely have APIs to access sources of truly random numbers. Most systems use deterministic generators of some kind, initialised with a seed. If there are dependencies between different sections of random number sequences, or covariances in the seed between peers, this will change the probability of collisions.

To make a good identifier, best practice is to use as many data sources as possible, with a strong mixing function. A strong mixer is one where each output bit is a complex, and different, function of all the input bits. The example given in RFC 4122 is to gather data sources into a buffer and use a message digest function such as SHA-1.

Identifiers and Bandwidth

A motivation for minimising id size is to reduce overhead. In a packet sending a `Vector3` a 16 byte (a 128-bit UUID) overhead for the Id would have an overhead of 133% without even the Component Id. Though, on modern networks even a large relative overheads may not be problems so long as the total packet size is low. IPV6, for example, has a 40 byte header alone.

That is, reducing bandwidth by reducing identifier size is unlikely to be an optimal methodology. It should be expected that Ubiq header sizes increase over time, rather than decrease. Since Ubiq is based on a set of point-to-point connections, future works could consider techniques such as asymmetrical per-connection aliases to optimise bandwidth.

Object Ids in Ubiq

Size

Currently, Object Ids are represented by 64 bit numbers. This is because the collision probability for this length is very low for scenarios anticipated by Ubiq in the near future, but it is still easy to handle; 32 bit is too small, while 128 bit would require explicit serialisation/deserialisation, hashing and equality comparisons. 64 bit numbers can be stored in a `long` and nowadays even mobile platforms use 64 bit processors.

Unique Ids in Unity

Unity provides a number of data sources for hashing inside the Application and SystemInfo classes, however these are not accessible in class constructors. Therefore the .NET System.Environment, System.DateTime and System.Random classes are used.

References

- Jesus, P., Baquero, C., & Almeida, P. (2006). ID Generation in Mobile Environments. 1–4. http://hdl.handle.net/1822/36065
- https://neilmadden.blog/2018/08/30/moving-away-from-uuids/
- https://preshing.com/20110504/hash-collision-probabilities/
- https://tools.ietf.org/html/rfc4122

Network and Component Types

Network Ids

Network Ids are 64-bit identifiers. They are represented by the `Ubiq.Messaging.NetworkId` structure in C# and the `Messaging.NetworkId` class in Javascript.

In C# Ids are value types, and the equality operators are overridden. In Javascript they are reference types and must be explicitly compared using the static `NetworkId.Compare` method.

```
NetworkId.Compare(message.objectId, server.objectId)
```

Internally, the types are represented by two 32 bit integers. This is an implementation detail and should not be relied upon. The reason for using two 32 bit integers rather than one 64 bit long, is that Javascript only supports 53 bit integers.

Binary vs Json

In Javascript, `NetworkId`s must be handled in both their binary and Json forms. This is because Ubiq messages include the `NetworkId` in their binary header, while some Javascript services, such as the Room Server, accept `NetworkId`s as arguments.

For example, a Json message would arrive containing the Id of an object that the Javascript code should send a message to. The Javascript code will need to convert that into a binary representation in order to build the header.

Binary `NetworkId`s are converted to `NetworkId` class instances by the `Message` wrapper. From this point on any Javascript code can work with the object in its Json representation.

`NetworkId` instances and generic Json objects representing a network Id can interoperate.

Common Ids

A number of Ids are pre-assigned to common objects in Ubiq. These are listed below.

Network Ids

- RoomServer: 0000-0000-0000-0001

WebRTC and VOIP

Real-time Audio in Ubiq is supported by WebRTC. WebRTC is the media stack supporting RTC (real-time communication) in modern web-browsers. WebRTC allows Ubiq to interoperate with browsers and second-party native applications on multiple platforms.

WebRTC supports everything required for RTC except signalling, by design. The WebRTC native libraries make their own P2P connections using ICE, and drive the audio devices. Ubiq exchanges signalling messages on behalf of the media stack to support this.

WebRTC was chosen as it is the only stack that interoperates with web-browsers. It is also the only way to exchange UDP with browsers.

It is recommended to use Ubiq's Audio API, rather than work with WebRTC directly.

Library

Ubiq currently uses SIPSorcery's .NET WebRTC library. Unlike typical WebRTC implementations, this library does not include a full media stack. Ubiq provides a simple audio endpoint which integrates with the Unity audio system, allowing for microphone input and spatialised audio output.

Operation

WebRTC operates using PeerConnection objects. These objects manage the connectivity with one or more other peers. RTC occurs in a P2P fashion, but PeerConnection objects require an out-of-band signalling system to establish their own RTP connections.

While PCs (PeerConnections) can form a one-to-many mesh themselves by fanning-out offer and answer messages, typically they operate in tandem. We assume PCs always form pairs.

Negotiation

WebRTC peers establish P2P RTP connections to pass audio, video and blob data through a process known as Interactive Connectivity Establishment (ICE). ICE is performed by having an out-of-band signalling system exchange SDP and ICE messages. SDP messages describe what streams a peer would like to send and consume, and in what format. ICE messages test different methods of direct connections that perform NAT and firewall traversal, until a preferred one is found.

The PC API is based on callbacks.

When a track is added or removed from a PC, it raises an event, OnNegotiationNeeded. This signals to the host application that it should initiate an exchange of SDP and ICE messages. The host does this by generating an Offer SDP message, and transmitting it to the counterpart PC. The PCs will themselves generate subsequent messages.

Servers and Rooms

Ubiq provides a Rooms system and a server. These work together to provide a fixed endpoint that allows users to rendezvous over the internet, where service discovery is impossible.

The Rooms system - the RoomClient and RoomServer - provide the concepts of Rooms and Peers. A Peer is a remote player. A Room is a place that multiple Peers can join. The example Server implements a RoomServer that allows Peers to find and join Rooms, and facilitates message exchange between the Rooms members.

Rooms and Messaging

The purpose of rooms is to control which users exchange messages with which.

At the messaging layer, Ubiq transmits messages across a Connection and expects the network to deliver them to matching objects. The connection made by the RoomClient to the Server is the same one used to deliver messages between networked objects. When the server receives a message, it forwards it to all other peers in the room.

After connecting to the server, a client begins in an empty room. Its messages will not be forwarded to anyone.

The RoomClient can join an existing room by exchanging messages with the RoomServer. The RoomServer is a 'virtual' networked object that exists on the server and listens for messages addressed to a particular Object/Component Id.

Once the RoomClient has sucessfully joined a room via the RoomServer, messages will begin to be forwarded between that client and the other members.

Rooms

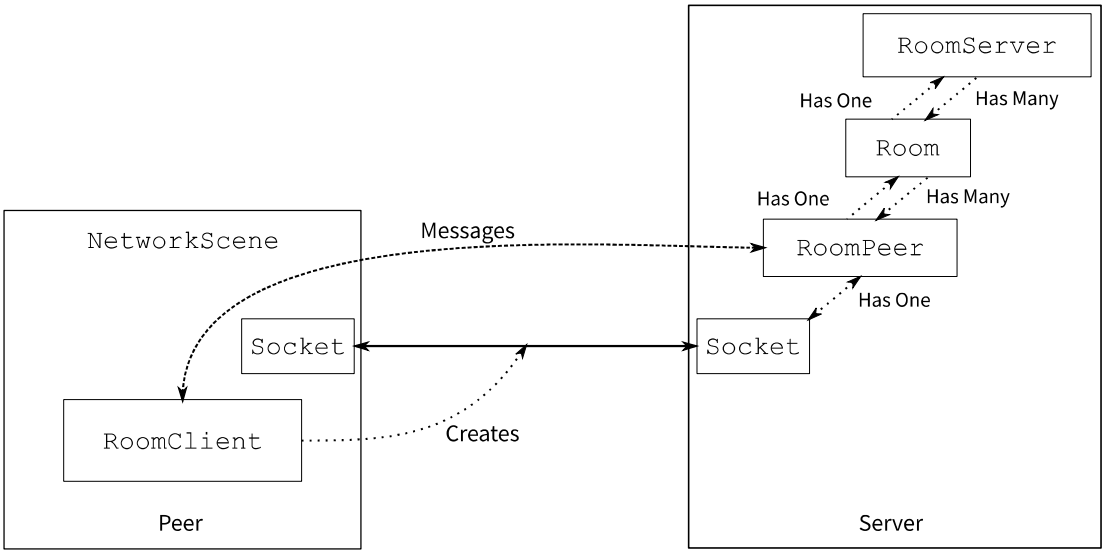
Rooms themselves are objects that exist in the RoomServer. Rooms contain a list of Peers, as well as a general purpose dictionary for storing low-frequency persistent information. Each Peer object contains the Socket used to communicate with the client.

Services

The RoomServer is one service that exists on the server. Other services include simple Blob management.

Overview

The Room Server service consists of a number of objects that work together to route messages to either the command and control interface on the server itself, or to other peers.



RoomClient Connections

The `NetworkScene` itself does not create any connections.

In the Ubiq sample code, the `RoomClient` is responsible for making the initial connection to the rendezvous server. The RoomClient has a pre-shared IP and port, which it assumes points to an endpoint with a Room Server running. The Ubiq server Nexus runs a Room Server.

When `RoomClient` starts up, it instructs the `NetworkScene` to make the connection. The underlying connection may be over TCP, WebSockets, or any other supported protocol. Once established the `NetworkScene` will assume it can be used to exchange Ubiq messages however.

On the server, new connections are wrapped with a `RoomPeer` object. The `RoomPeer` is able to parse Ubiq messages. It also has references to the global `RoomServer` object. The `RoomPeer` acts as a gatekeeper. It parses messages intended for the `RoomServer` and calls the appropriate APIs. Other messages it forwards to the `Room` it is a member of.

The APIs it invokes in response to `RoomClient` messages will cause the `RoomServer` object to move the `RoomPeer` between different rooms (or remove it from all rooms).

Heartbeat

`RoomClient` instances will ping the RoomServer at a 1 Hz. If they do not receive a response after 5 seconds, they consider the server to have timed out, and will notify the user.

Timeouts

If a client drops a connection to a server, the server will inform all other peers by emitting `OnPeerRemoved`.

The server will only remove a client when the underlying TCP connection has been closed. Until this time the client may re-appear.

If a client connection breaks without shutting down cleanly, there may be a delay during which the Peer is in the room, but cannot exchange messages. Components should be robust to this. How they behave will depend on their use-case. For example, if a peer existed in the room but had no effect on it, there would be no need to detect this case at all. The Avatar class uses interruptions in its transform stream to detect a disconnected peer, and will hide an Avatar after a few seconds if no data is received.

An easy way for Components to handle this case is to ensure they are created under a Peer's Avatar, in which case they will be enabled and disabled with the Avatar.

`RoomClient` instances do not necessarily control the connections to `RoomServer` and so cannot detect if these are lost. `RoomClient` instances ping the `RoomServer` routinely and if they do not get a response after a set time, will disconnect the connections they are responsible before and consider the room left.

Leaving and Joining

When a `RoomClient` starts it is automatically a member of an empty, unidentified, room.

A `RoomClient` can create a new room or join an existing one at any time. Joining or leaving a room involes a *room change*. Changing the room is the same whether the client is going to or from an empty room, or between non-empty rooms.

When the room changes `RoomClient` will emit a number of events, in order:

1. `OnPeerRemoved` for any peers that go out of scope
2. `OnPeerAdded` for any peers that come into scope
3. `OnPeerUpdated` for any peers that are in scope but whose properties have changed
4. `OnJoinedRoom` with the new room
5. `OnRoomUpdated` with the new room

Whether a Peer is in *scope* means whether or not it is available to current Peer. If two Peers moved to another room at the same time, they would remain in scope because the Peers themselves remain in the same rooms, even though that room has changed. On the other hand, if a Peer joins a different Room, it will go out of scope, as it is no longer in the same room, even though it may still be connected to the server.

The purpose of rooms is to facilitate message exchanging between specific sets of peers. Most Components should use the Peer events to create, destroy and update objects. The Room events are typically used when code needs to control room membership, for example the UI panels for joining rooms.

There is no such thing as leaving a room; underneath, leaving a Room means joining a new, empty Room.

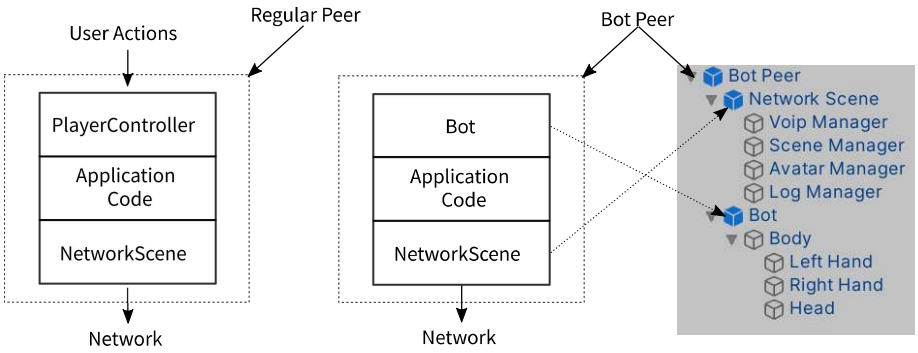
Bots

Ubiq has the ability to simulate large numbers of users. This may be useful when stress testing, for example. This is demonstrated in the *Bots* Application Sample.

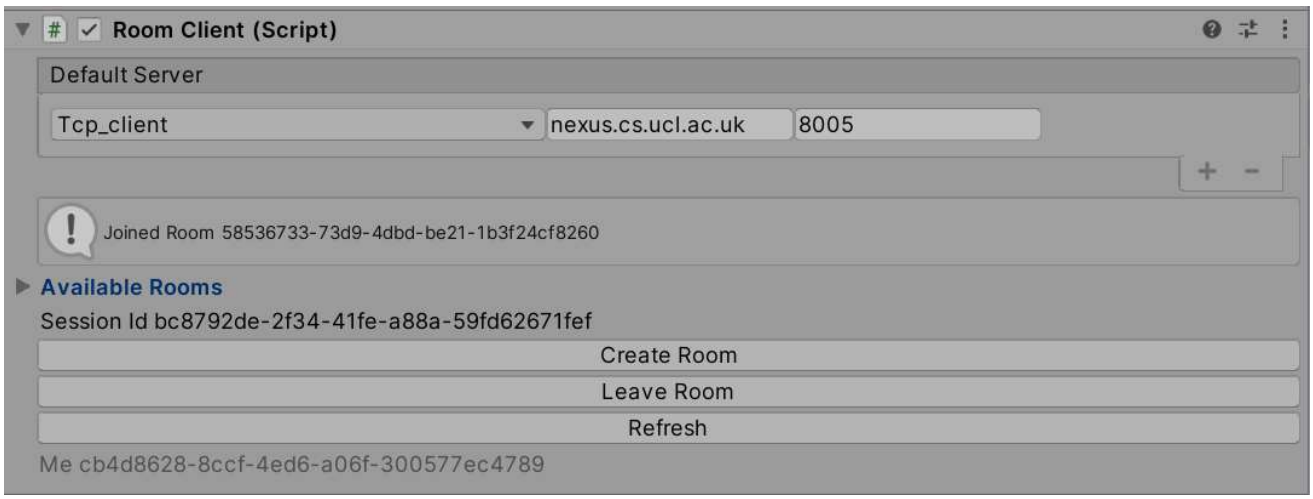
Bot Peer

In a regular application, a user interacts with the world through the `PlayerController`. They connect to the network using a `NetworkScene` and become a *Ubiq Peer*.

It is the same with the Bot and Bot Peer. The Bot GameObject contains Components which interact with the world. Pairing a `Bot` with a `NetworkScene` creates a *Bot Peer*, that can connect to a room and act autonomously.



The Bot Peer is equivalent to a user application. In the Editor, the Bot Peer can be joined to a room by using the RoomClient Editor Controls, just like a regular Peer. Alternatively, the default Bot, and others, can be controlled en-masse using the Bots Controller.



The Bot Peer Prefab can be dropped into new scenes and controlled via the Editor, where small numbers of Bots are required.

Bot

The Bot Prefab implements the behaviour of the non-player controlled character. The sample Bot contains a number of common abilities, which can be extended by adding additional custom Components.

Avatar

The Bot Peer contains a skeleton allowing it to embody an Avatar at remote peers.

Behaviour

The Bot Peer Avatar is controlled by the Bot Component. This uses a nav mesh to move between random points in the environment.

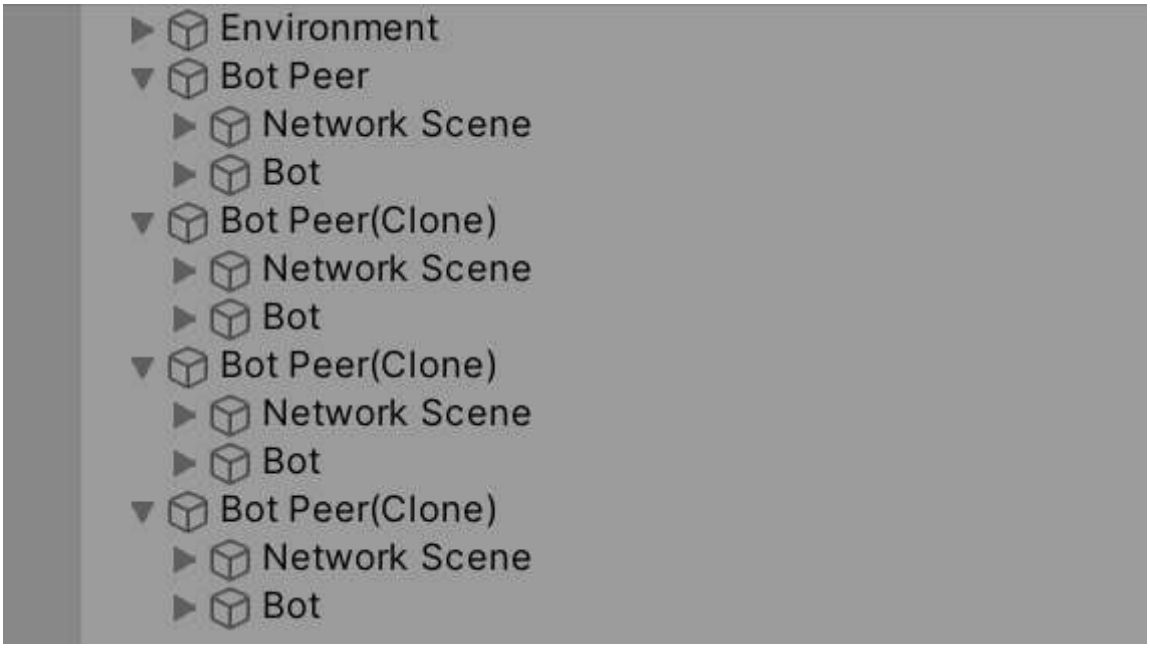
Audio

The Bot Peer can speak with a pre-recorded audio clip. This is through a Component that takes the place of the Microphone input in a player-controlled Peer.

Managing Multiple Bots

Usually, a single Unity process is a single Ubiq Peer, with one NetworkScene. This is because a single PC can usually only drive one set of user input devices at a time.

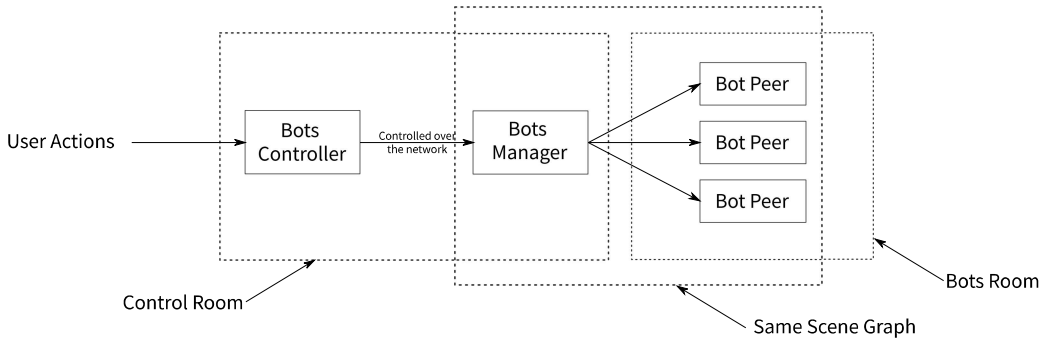
With bots, a single process can host multiple Bots, but each Bot is still a separate Peer, and has its own NetworkScene. This is achived using the same scene-graph Forests as the *Local Loopback* scene.



BotsManager & BotsController

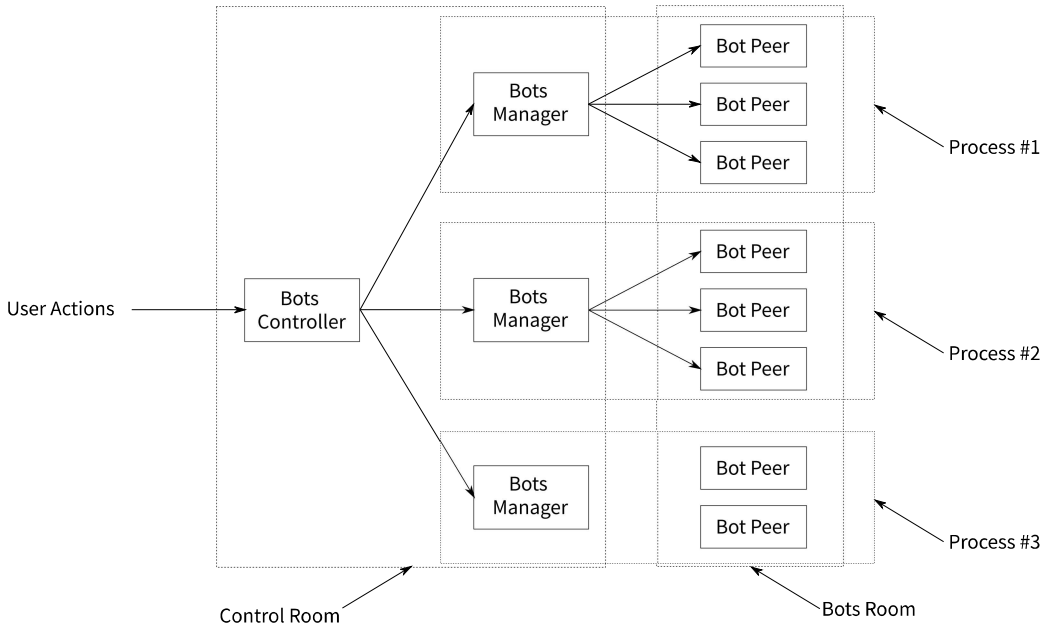
The *Bots* Application Sample contains Components to create and manage large numbers of bots.

The `BotsManager` is the Component that creates and configures Bot Peers within a single process. Bots Managers are controlled remotely by a `BotsController`.



The `BotsController` and `BotsManager` use Ubiq to communicate. The `BotsManager` and `BotsController` join the same *Command and Control Room*. The Bot Peers themselves join different rooms, possibly even on different servers. The `BotManager` communicates with the Bot Peers through the local scene graph.

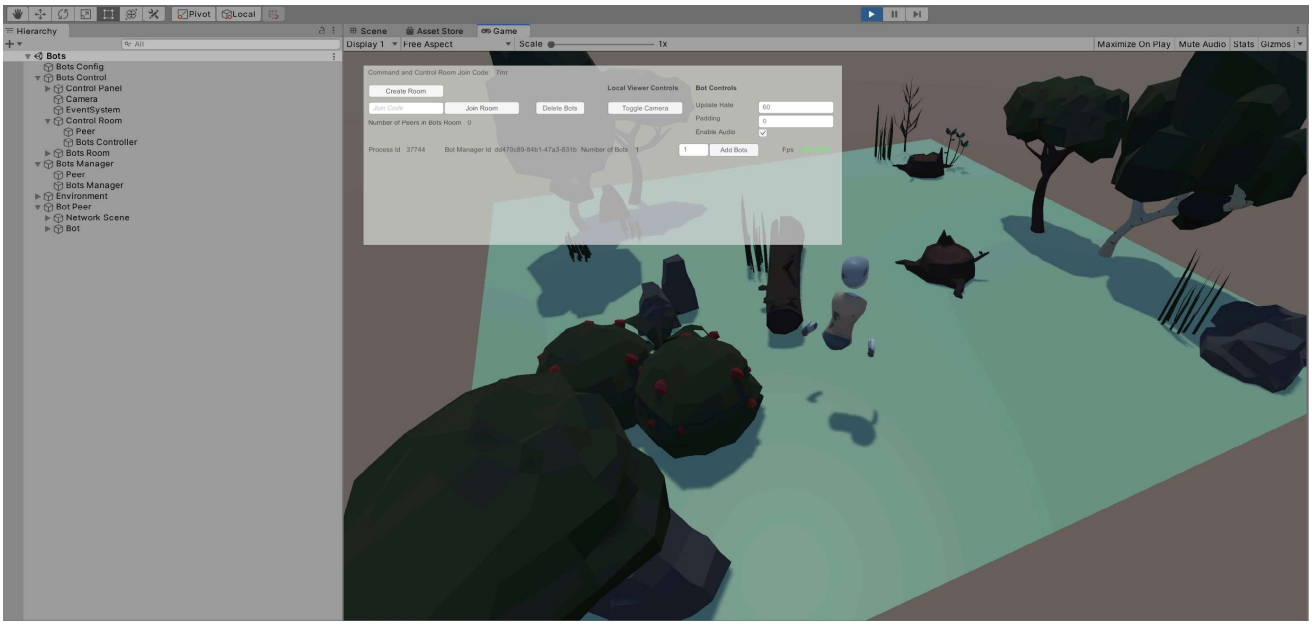
Multiple Bot Managers across different processes can be controlled this way, each managing a number of Bots.



A typical Unity process running on a modern desktop can handle approximately 20 bots. All three Components can run within one scene graph, within one process, as in the Bots Sample Scene, or can be split between multiple machines to control hundreds of bots.

Creating Bots with the Bots Scene

Opening the Bots scene and pressing Play will show a birds-eye view of the local process, and a control panel. This scene is only meant to be used on the desktop.



The Bots scene already has one Bot Peer created. This GameObject's NetworkScene Editor Controls or the Create Room/Join Room Control Panel Buttons can be used to immediately join the Bot into a room with other Peers - bots or normal Peers.

Stress Testing with the Bots scene

One of the uses for Bots is stress testing. The Bots Sample is set up to show this.

The Bots Scene contains a number of Components, and a UI.

Bots Control

This GameObject contains resources to allow a user to control one or more Bot Managers with the UI. It also includes the Camera and Event Manager. The UI contained under Bots Control drives the `BotsController`, which in turn directs Bot Managers.

The `BotsController` is in the Control Room GameObject, which has its own NetworkScene and so forms the Bots Controller Peer.

This branch also contains a Peer (Bots Room Peer) that can be connected to the Bots' room, in order to do things such as take statistics or collect logs.

Bots Manager

The Bots Manager GameObject is the forest for the Bots Manager Peer. Though they are in the same scene graph, the Bots Manager Peer and Bots Controller Peer both join the same command and control room.

Environment

To reduce memory overhead, multiple Bot Peers share the same Environment, though as they have their own NetworkScene they don't directly interact outside of Ubiq networking.

Bot Peer 1

The Scene includes one Bot Peer instance. The Control Panel can be used to add more.

Bots Config

The Bots Config GameObject hosts a helper Component to set which servers the Control Room and Bots Room should be hosted on at design time. By allowing these to be set at design time, the Bots Scene can be built to an executable to be run headless on different machines without needing further configuration.

Make sure to change the Control Room Id before running the sample against Nexus, in case others are running the same sample.

Controlling Bots

The Bots Manager is used to spawn new Bot Peer instances. Bot Peers exist in the same application but are completley independent as far as the network, and other peers, are concerned. The UI in the Bots scene is for the Bot Controller. There is no interface for interacting with the virtual world, VR or otherwise, as there is no Network Scene for players in the Bots example. Bots can be instructed to join any room however, including those with regular players.

Command and Control and Bots Rooms

When the Scene is started, the Control Panel UI will control the local Bot Manager. However, there is a limit to how many Bot Peers a single Unity process can host.

Bot Managers across different Unity processes can work together to control very large numbers of bots.

The Control Panel and Bot Manager are two distinct Ubiq Peers. They communicate using Ubiq Messages through a 'Command and Control' Room.

When the Control Panel starts, it creates a new Room and has the local Bot Manager join it. Additional Bot Managers from other processes can join this room too, and fall under the control of the Control Panel.

The Command and Control Room can be any Ubiq Room, including the one that the Bots join.

The Room can be set via the command line with the -commandroomjoincode argument. This, combined with -batchmode, can allow many headless Unity instances to spawn bots.

In practice though, when doing things like stress testing, the Room should be different, even on a different server if doing server stress testing.

Servers

There are in total four distinct Peers in the single Bots Sample: the Bot Controller (Control Room), the Bot Manager (Control Room), the Bot Room Peer (Bots Room) and the default Bot Peer (Bots Room).

The Bots Config Component is used to change the server(s) for all of these at once.

The default server for Bot Peer 1 is set in the standard NetworkScene Prefab; it is overridden before connecting when the Control Panel is used to control the Bots.

Control Panel

The UI has two sets of controls: Common controls and Instance controls.

Common Controls apply to all Bot Managers, and Instance controls apply to just that Bot Manager.

When only one Bot Manager instance is known, the controls behave identically.

Toggle Camera

As each Peer acts as if it were the only one in the process, each Peer will show all other Peers Avatars. This can create high rendering loads. Avatars can be hidden by toggling the Camera, which changes the Culling Mask. The camera is not completley disabled, as it is needed for the UI.

Enable Audio

Enables or Disables the audio chat channel for new Bots. This can be used to reduce compute load on the clients. This may be desireable if stress testing a server, for example, as audio data doesn't pass through the server.

Create/Join Room

The Create Room button creates a new room for *Bots*, and commands any existing Bots to join it. Alternatively, an existing Join Code can be entered and the Join Room button used. Either can be used as many times as desired to change all the Bots at once, without re-creating the Bots.

Number of Peers

Shows the number of Peers in the Bots Room, including the 'dummy' peer. A process may host a number of Bots that have not yet joined a room. Use this figure to monitor the actual number of bots in a room together.

Bot Manager Instances

Below the Common Controls each Bot Manager that the Controller is aware of is listed. Each line shows the Id of the Manager, as well as a the number of bots it is hosting. The Input Field and Add Bot Button can be used add new Bots to that instance.

If a Bots Room has been set up, new Bots will automatically join it.

The FPS is used to approximate the performance or load of the Bots Manager. The Colour is controlled by the Fps Gradient member of the Bots Manager Control Prefab (between 0-100 Fps).

Bot Scene

The Bots Application Sample also contains a sample scene with a single Bot Peer, along with a Player Peer. This can be used to see what it is like for players to interact with Bots. In this scene, the Player and the Bot should be joined to the same room manually using the RoomClient Editor controls.

Asyncrhonous Design Patterns in Unity

The Unity process manages the main thread, which begins before any user code is executed. Most Unity resources can only be accessed from the main thread; an exception will be thrown otherwise. There are still many possibilities for writing aysnchronous code however.

Design Pattern

Delayed initialisation with callbacks. Mimics the do-then pattern in JS. Methods are called which take Actions. Those Actions are initialised by lambdas. The lambas execution thread depends on the called function.

```
void Start()
{
    factory.GetRtcConfiguration(config =>
    {
        pc = factory.CreatePeerConnection(config, this);
    });
}
```

Design Pattern

Message pumps with Update. Commonly used in the mid-level networking code, this pattern uses a list of actions to execute methods on the main thread.

```
class RoomClient
{
    private List<Action> actions = new List<Action>();

    public void SendToServer(Message message)
    {
        actions.Add(() =>
        {
            SendToServerSync(message);
        });
    }

    private void Update()
    {
        foreach (var action in actions)
        {
            action();
        }
        actions.Clear();
    }
}
```

Design Pattern

Commonly used in webrtc code for objects that take time to initialise because they are waiting on external resources. This pattern uses coroutines to effectively poll a resource, conditionally executing operations on the main thread.

```
    void Start()
    {
        factory.GetRtcConfiguration(config =>
        {
            pc = factory.CreatePeerConnection(config, this);
        });
    }

    private IEnumerator WaitForPeerConnection(Action OnPcCreated)
    {
        while (pc == null)
        {
            yield return null;
        }
        OnPcCreated();
    }

    public void AddLocalAudioSource()
    {
        StartCoroutine(WaitForPeerConnection(() =>
        {
            var audiosource = factory.CreateAudioSource();
            var audiotrack = factory.CreateAudioTrack("localAudioSource", audiosource);
            pc.AddTrack(audiotrack, new[] { "localAudioSource" });
        }));
    }
```

Packaging

Overview

The Unity side of the project can be built as a package for the Unity Package Manager.

Releases

Release numbers use SemVer. The version is to be put in the filename and the package.json file.

The intention is that each release corresponds to a commit. To this end if you are making a new release version of the package:

1. Make sure you have no modified files (check your `git status` - the script does this for you)
2. Add a tag on your current commit with the release number, push the tag to the origin repo

Building (Script)

There's a simple build script included in the project. It's integrated with the Unity Editor. Access it through the taskbar:

`Ubik-dev -> Pack for Unity Package Manager`

Building (Manual)

Building the package manually is quite painless.

1. Check `git status` does not indicate any modified files
2. Make a new folder and copy over the Editor, Runtime and Samples folders and the package.json and package.json.meta files
3. Rename the Samples folder to Samples~ (this stops Unity importing it into the main package)
4. Zip it!
5. Name the zipped file ubik-{versionnumber}.zip

Nexus

Overview

The UCL VECG hosts multiple instances of the rendezvous server on nexus.cs.ucl.ac.uk. Different branches of this repository are checked on nexus and run on different ports.

The checkouts are in `/home/node` and follow the format `ubiq-[branch name]`.

Currently `ubiq-master` is running on `8010`. This is the primary, public server.

It is expected and encouraged that feature branches are created, run on nexus temporarily for development, then removed when no longer needed.

The following sections describe how Nexus is maintained by the VECG team. You do not need to follow this pattern to maintain your own server, but it may be instructive.

Administration

Access to `nexus.cs.ucl.ac.uk` is via SSH.

The nodejs process is managed via pm2. The relevant commands are:

- `pm2 list` (Shows running processes)
- `pm2 log` (Shows the logs)
- `pm2 flush` (Clears the logs)
- `pm2 restart` (Restarts the app, e.g. after an update)

pm2 is responsible for restarting the nodejs process after a server restart. To save the state of the tasks that it will try to restore, give the command `pm2 save`.

nodejs and pm2 run under the `node` account. All maintenance should be performed as `node`. The username/password is node/node. It is not possible to SSH directly as node; login with your CS credentials, then change user with `su` (i.e. `$ su node`).

All VECG members who request access will be given sudo permission. All members will be collaborators on the Github repository. Any member can add new members.

Git

The node account has been given access to the GitHub repository through a Deploy Key. This is a single-use SSH key associated with the repository.

New

When cloning a new copy for a branch, you must specify the folder as git will always clone into the repository name by default. For example: `git clone --depth 1 git@github.com:UCL-VR/ubiq.git ubiq-master`

You can specify the branch name for the clone command (`git clone --depth 1 --branch master git@github.com:UCL-VR/ubiq.git ubiq-master`), or checkout the appropriate branch after.

The `--depth 1` command downloads only the `HEAD`, which is all that is needed to run the server.

After cloning, navigate to `~/ubiq-[branch name]/Node/` and issue the commands:

1. `npm install`
2. `pm2 start npm -- name "ubiq-[branch name]" -- start`

The first installs the nodejs dependencies and the second creates the pm2 job with a unique name to identify the instance.

Updating

To update a checkout, enter the repository and issue `git pull`. `node` cannot write to the repository.

Firewall

Be aware when creating your own clones that you will need to open your chosen ports on the local firewall and reload it.

To see the firewall rules on CentOS, give the command,

```
sudo iptables -L -n -v --line-n
```

The `-n` argument shows the port numbers, rather than showing the names of typical services that run on them. `-v` shows the interface, and `--line-n` shows line numbers, which will be important when adding rules.

To add a new rule give the commands,

```
sudo iptables -I INPUT 5 -p tcp --dport 8010 -j ACCEPT
sudo service iptables save
```

The number after `INPUT` indicates the line that the rule should be added at. The `ACCEPT` rule must come before the catch-all `REJECT` rule. For example, the `REJECT` rule is on line 11 below and will reject all packets not matching any of the rules above it.

```
1    ACCEPT     all  --  0.0.0.0/0           0.0.0.0/0           state RELATED,ESTABLISHED
2    ACCEPT     icmp --  0.0.0.0/0           0.0.0.0/0
3    ACCEPT     all  --  0.0.0.0/0           0.0.0.0/0
4    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           state NEW tcp dpt:22
5    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8006
6    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8005
7    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8004
8    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8003
9    ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8002
10   ACCEPT     tcp  --  0.0.0.0/0           0.0.0.0/0           tcp dpt:8001
11   REJECT     all  --  0.0.0.0/0           0.0.0.0/0           reject-with icmp-host-prohibit
```

The `iptables save` is for the `iptables-service`, which is installed on `Nexus`, and allows saving and automatically reloading firewall rules on reboot.

Be aware also that opening the ports locally will not make them available on the public internet, but only via the CS VPN. Nexus has range `8000-8020` open on the CS and ISD firewalls.

Legacy Versions

When breaking changes are made, legacy versions of the server will be checked out with their last supported version number, e.g. `ubiq-0.0.6`, and instances of these will be left running. The ports that they listen on will be incremented with each breaking change.

Old clients will therefore continue to work, though old and new versions cannot communicate with eachother.

Not all versions include breaking changes, so the sequence of legacy versions running is not continuous. The latest version is always `ubiq-master`.

Currently `ubiq-0.0.6` is running on `8001`.

References

- https://upcloud.com/community/tutorials/configure-iptables-centos/

MkDocs Commands

This page is for the Ubiq developers.

This documentation is generated with MkDocs. For full documentation visit mkdocs.org.

To keep linking between pages simple, try to keep all MD files in the docs/ directory. Images should be placed in images/ and all filenames should be GUIDS. Please delete images if you remove a reference to them. The exception is reusable assets such as logos, which have human-readable names.

There is a special branch of Ubiq ("Documentation") which can be used to store Scenes and other assets that were set up for screenshots.

- `mkdocs new [dir-name]` - Create a new project.
- `mkdocs serve` - Start the live-reloading docs server.
- `mkdocs build` - Build the documentation site.
- `mkdocs -h` - Print help message and exit.