Forge Networking Remastered

Welcome to your new documentation space!
This is the home page for your documentation space within Confluence. Documentation spaces are great for keeping technical documentation organised and up to date.

Next you might want to:

- ☐ **Customise the home page** - Click "Edit" to start editing your home page
- ☐ **Check out our sample pages** - Browse the sample pages in the sidebar for layout ideas
- ☐ **Create additional pages** - Click "Create" and choose "Blank Page" to get started
- ☐ **Manage permissions** - Click "Space Tools" and select "Permissions" in the sidebar to manage what users see

Search this documentation

No labels match these criteria.

Featured Pages

Content by label

There is no content with the specified labels

Popular Topics

Recently Updated Pages

- Removing Examples and Modules
  Dec 14, 2016 • created by Brent Farris
- Removing Buffered RPC Based on Arguments
  Dec 05, 2016 • created by Brent Farris
- Name Collision Issues
  Nov 15, 2016 • created by Brent Farris
- Lobby
  Nov 14, 2016 • created by Brent Farris
- Stand Alone Chat
  Nov 14, 2016 • created by Brent Farris

User Manual (Forge Networking Remastered)

Welcome to Forge Networking Remastered, we would like to personally thank you for electing to become a part of our journey into Networking systems and Software as a whole. You are here because you have become a premium supporter of the Forge Networking project!

NOTICE
This documentation is an on-going process. We have created "stubs" which are basically empty pages, in order to remind us (the developers) of topics we need to be sure to cover in the documentation. If there is a stub (blank page) for a topic, do not fear, we will be getting to filling it out ASAP!

Forge Networking to Remastered Migration Guide

First of all we would like to **thank you** so much for supporting Forge Networking over the past two years and we are proud to bring you **Forge Networking Remastered**!

Forge Networking Remastered (FNR) is a re-imagining of networking for Forge. We took everything we learned, all the feedback we could get and we started **from scratch** on FNR. We kept some of the helper classes such as **ObjectMapper** and **BMSByte** as those were optimized classes just for generic network data transfer. Secondly we started FNR with the thought of **Web Sockets** and WebGL in mind. Though (at the time of this writing) WebGL support is not officially complete, we are working very hard to solidify this platform (mainly in relation to Unity integration and limitations). Thirdly we wanted to completely remove any kind of reflection "magic" that was in the system. In place of reflection we opted for generated class files and code, this makes debugging and tracking execution much easier to maintain and test.

Now for the reveal of the biggest change to Forge Networking, the Network Object. We have replaced the old singular, monolithic **inheritance** net work model in place of an **attachment** network model. Now you can write your code as you wish without overly intrusive and complicated code/reflection and have a more solid extendable model. We have also included the new Network Contract Wizard (NCW) which allows you to easily blueprint and review your network contracts without having to dig through tons of code. We have completely re-imagined networking to be more of a service, attachment, addon, and extension rather than an overly complex integrated model.

Of course you will find many things that you once knew such as Network Instantiation, Remote Procedure Calls RPC, **MainThreadManager** (Threading in Unity) however you will find that some of the older models (such as **NetSync**) have been removed. I know, I know, **NetSync** is awesome and it works kinda like UNET's **SyncVar**; however we believe we have a much simpler, more controlled, less magical, and more powerful approach to serializing network variables. Since we have abstracted all the network code to be an attachment, you will find a new fancy **network Object** variable that is a part of the class you create. This object will house all of your network variables and allow you to set the sync time as well as access any variable from the network easily. This means you can **choose** when to use the network variable, choices are good, yes? 🙂 Also the networkObject is powerful, it automatically detects changes to your variable (as a whole) and serializes only that variable as apposed to the whole network class!

So what are the difference highlights?

- Instantiation is done through **NetworkManager.Instance.Instantiate...**
- Remote procedure calls are done through **networkObject.SendRpc**
- NetSync has become **networkObject.[myVar]**
- PrimarySocket has become **NetworkManager.Instance.Networker** (You can also get the networker from the **networkObject**)
- WriteCustom has become **Binary** (Check out the VOIP module for how to use Binary)
- Networking and NetworkingManager have been replaced by **NetworkManager**

So what are some of the new highlights?

- Network Contract Wizard (NCW)
- Support for NAT hole punching
- Master Server has Matchmaking
- "Cherry picked" network variable updating
- State rewinding
- Network error logging system
- The Network Object

Notes From Community

Do not put NetworkManager into your scene, it will be spawned on it's own.

Authoritative Design

Often you will hear the term **authoritative server** and in most cases it is used to mean **authoritative game logic**. So we would like to first go over the difference between the two as we often state that Forge Networking (including Remastered) is **authoritative server** by design.

Authoritative Server: When all network traffic goes through the server before it is relayed to other clients. In other words, clients do not communicate with each other directly.

Authoritative Game Logic: When all your game code is written in a way where the server is completely in control of the flow and owns all of the network objects. This is unique to your game and how your game plays, it can be complex based on your game and we would suggest exploring the idea of not having authoritative game logic unless your game requires it. Games that require it often are games that have long term tracking (such as Battlefield) or track data for users in a database (such as an MMO). Games that plan to have non-tracked multiplayer (such as Minecraft) would not opt into having full server authority due to the overhead of complexity.

So when we say that Forge Networking (and Remastered) is an **authoritative server** we are talking about the former where all network traffic goes through the server first and then is relayed to all of the clients.

Getting Started

The following is collection of examples/tutorials to get you up and running with Forge Networking. Our goal with these examples are to help you learn the basics of the engine quickly through guided step by step processes. We do cover more in-depth explanations of the various components of the engine outside of the examples, however we do suggest going through these examples first before taking a deeper dive into the complexities and extensions of the engine found in other sections of this documentation.

Basic Moving Cube Example

This example is not a continuation of other examples and should be treated as if it were done in a new project.

- Planning Network Code
- Network Contract Wizard
- Extending Generated Classes
- Scene Setup
- Test

In this example we are going to make a simple game where we have a cube in the scene that is owned by the server and all the clients can see it being moved by the server.

What is covered in this example

1. Plan for networking code
2. Use the **"Network Contract Wizard"** to create a network object type with properties
3. Extend the generated class
4. Implement the network code in your custom class
5. Manage who executes what code in the extended class
6. Host a server
7. Connect with a client directly via IP address and Port

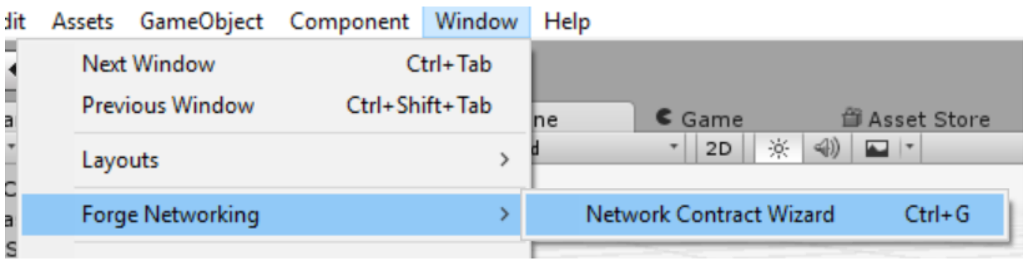
Planning Network Code

So one of the first things we want to think about is our **Network Contract**. This is basically a fancy word for how we design/setup network communication. It is helpful to imagine our final result to our application so that we can properly design for it. In this application when a client connects it will see a cube, then the cube will move around in sync with the server, who is the person actually moving the cube. To accomplish this we need to:

1. Have a cube already in the scene
2. Synchronize the position and rotation of the cube to all the clients from the server
3. The transformations should be smooth so we need to interpolate them

Network Contract Wizard

Now that we know that we need to sync the **position** and **rotation** of a cube, we can design our network contract for that object. We will first open the **Network Contract Wizard** which is a UI provided by the Bearded Man Studios team to make it easy to design your **network contracts** in a easy way. To open this menu, go into Unity and select "Window->Forge Networking->Network Contract Wizard".

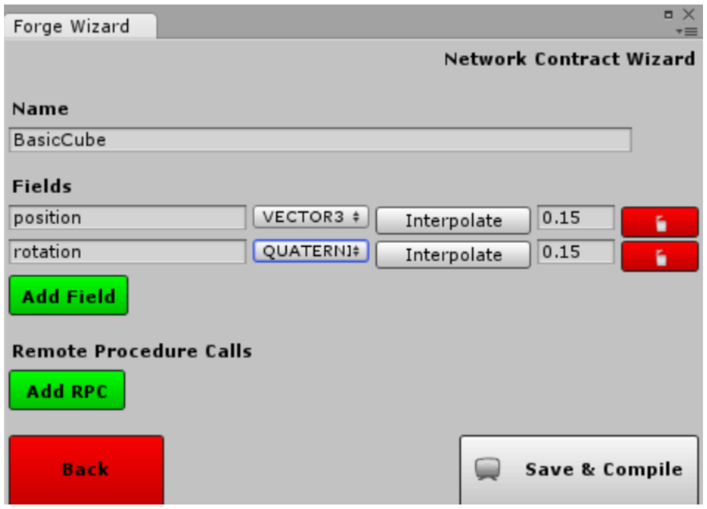


Once you have opened this editor you will be presented with a list of all the Network Objects currently available, to learn more about this please see the document on the Network Contract Wizard as we will just be going over how to create a network object through the contract wizard. To start, click on the "Create" button near the top and you will be presented with the create UI. In here, we have 3 major fields of interest, the **Name** fields, the **Fields** field, and the **Remote Procedure Calls** field.

1. The **Name** field is where we create the name for our Network Object and behavior, this is a friendly name that should be written in "Pascal case" to follow the C# convention since it is going to be a part of the class names that are generated.
2. The **Fields** section shows all of the various fields that our network should be aware of. In this case we are going to want to make a **position** and **rotation** field which are **Vector3** and **Quaternion** respectively. *Note, you can name these fields whatever you like, these are just friendly variable names for you (and your team) to know what they are for when used*
3. The **Remote Procedure Calls** field is where you will design any Remote Procedure Call (RPC) function signatures. We are not going to go over this field in this tutorial as we do not need it for the goal we are attempting to accomplish.

Lets begin by naming our Network Object:

1. Lets set the name for our Network object to "**BasicCube**" (without quotes)
2. Click the **Add Field** button
3. Name the new field **position**
4. Set the type to **Vector3**
5. Click the **Interpolate** button
6. Set the interpolate time (the text field that pops up after clicking the **Interpolate** button) as **0.15**
7. Click the **Add Field** button
8. Name the new field **rotation**
9. Set the type to **Quaternion**
10. Click the **Interpolate** button
11. Set the interpolate time (the text field that pops up after clicking the **Interpolate** button) as **0.15**
12. Click the **Save & Compile** button



Name: BasicCube

position	VECTOR3	Interpolate	0.15
rotation	QUATERNI	Interpolate	0.15

Extending Generated Classes

When we use the **Network Contract Wizard (NCW)** we are actually generating a lot of network code based on what has been input into the editor fields, this actually cuts out a lot of work that you would have to do by hand. There is one class in particular that we want to extend from, this class name will be "**BasicCubeBehavior**". The naming convention for this generated class is _____Behavior where "_____" is the name we typed into the NCW. Lets now create a C# file in Unity and write our basic game logic, we will name this file "**BasicCube**".

1. Open the newly created C# file
2. Add **using BeardedManStudios.Forge.Networking.Generated;** to the using statements
3. Derive the class from **BasicCubeBehavior**
4. Write the rest of the logic for the cube as seen below

BasicCube

```
using UnityEngine;
using BeardedManStudios.Forge.Networking.Generated;

public class BasicCube : BasicCubeBehavior
{
    /// <summary>
    /// The speed that the cube will move by when the user presses a
    /// Horizontal or Vertical mapped key
    /// </summary>
    public float speed = 5.0f;

    private void Update()
    {
        // If we are not the owner of this network object then we should
        // move this cube to the position/rotation dictated by the owner
        if (!networkObject.IsOwner)
        {
            transform.position = networkObject.position;
            transform.rotation = networkObject.rotation;
            return;
        }

        // Let the owner move the cube around with the arrow keys
        transform.position += new Vector3(Input.GetAxis("Horizontal"),
Input.GetAxis("Vertical"), 0.0f) * speed * Time.deltaTime;

        // If we are the owner of the object we should send the new
position
        // and rotation across the network for receivers to move to in the
above code
        networkObject.position = transform.position;
        networkObject.rotation = transform.rotation;

        // Note: Forge Networking takes care of only sending the delta, so
there
        // is no need for you to do that manually
    }
}
```

As you can see from the code snippet above, you can determine if the current player is the owner of the object using the **networkObject.IsOwner** boolean comparison. This will allow you to make code specifically based on the owner of the object. In this case, since the cube is going to be in the scene at start, it's owner is the **server**. In the snippet above the client (non owner) will update the transform position and rotation of the cube (the object this script is going to be attached to) to the position and rotation received from the server. Since we turned on interpolation, all of the smoothing is done "behind the scenes". Now the server in this case will just assign the **position** and **rotation** variables of the networkObject. These **<u>are</u>** the two fields we created in the NCW by the way. All generated network objects from the NCW will have a **networkObject** member variable that you can access from the deriving child. Whenever you assign a field of this object it is replicated across the network if the assigning user is the owner of the object.

Scene Setup

Now that we have done all of the network programming required for our end goal, it is time to setup our scene.

1. Create a new scene in unity
2. Create a cube for your floor
3. Set the position of this cube on the **x**, **y**, and **z** to **0**, **0**, **0** respectively
4. Set this cube's scale on the **x**, **y**, and **z** to **25**, **0.1**, **25** respectively
5. Create another cube which will act as the active moving cube
6. Set the position of this cube on the **x**, **y**, and **z** to **0**, **3**, **0** respectively
7. Attach a 3D **Rigidbody** Component to the cube
8. Attach our **BasicCube** script to this cube
9. Save the scene as **BasicCube**
10. Open up the build properties for Unity
11. Add the **MultiplayerMenu** scene as the first scene
12. Add your **BasicCube** scene as the second scene

Test

Now that we have setup our scene and everything else, it is time to test the game.

1. Open the **Build Settings**
2. Click on **Player Settings...**
3. Open the **Resolution and Presentation** section
4. Turn on **Run In Background***
5. Go back to **Build Settings**
6. Click on **Build And Run**
7. Once the game is open, return to the Unity Editor
8. Open the **MultiplayerMenu** scene
9. Click the play button
10. Click the **Host (127.0.0.1:15937)** button on the bottom of the game view
11. Go back to the built game
12. Make sure the host ip address is set to **127.0.0.1**
13. Make sure the host port is set to **15937**
14. Click the **Connect** button
15. Select the server game instance (Unity Editor)

Now if you move around the cube in the editor, you will see the movements replicated to the clients.

Basic RPC Example

This example is not a continuation of other examples and should be treated as if it were done in a new project.

In this example we are going to go over how to use the built in RPC methods inside of Forge Networking Remastered. In this example we are going to make a scene that already has a cube in it, then if anyone presses the up arrow key it will move the cube up, if anyone presses the down arrow key, it will move the cube down.

What is covered in this example

1. Plan for networking code
2. Use the **"Network Contract Wizard"** to create a network object type with RPC methods
3. Extend the generated class
4. Implement the network code in your custom class
5. Host a server
6. Connect with a client directly via IP address and Port

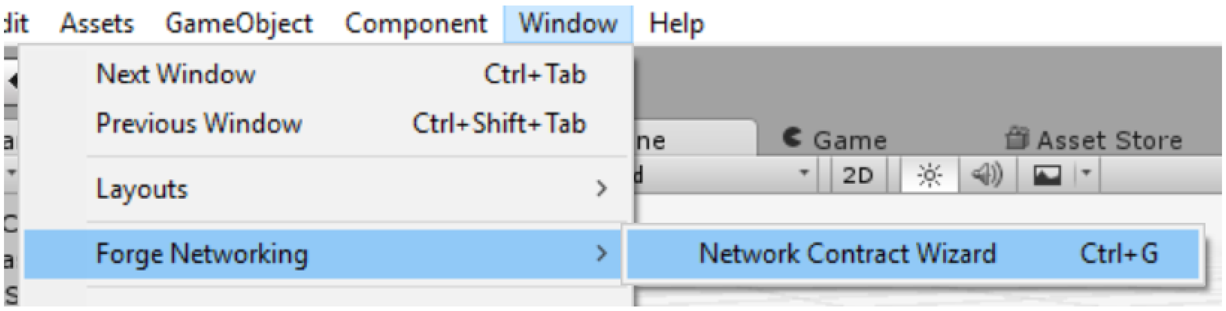
Planning Network Code

So one of the first things we want to think about is our **Network Contract**. This is basically a fancy word for how we design/setup network communication. It is helpful to imagine our final result to our application so that we can properly design for it. In this application when a client connects it will see a cube, if that client or any other connection (server or client) presses the up or down arrow key the cube will move in the appropriate direction for everyone connected:

1. Have a cube already in the scene
2. Press up arrow key to move the cube up via RPC method
3. Press the down arrow key to move the cube down via RPC method

Network Contract Wizard

Now that we know that we need to sync the **position** and **rotation** of a cube, we can design our network contract for that object. We will first open the **Network Contract Wizard** which is a UI provided by the Bearded Man Studios team to make it easy to design your **network contracts** in a easy way. To open this menu, go into Unity and select "Window->Forge Networking->Network Contract Wizard".



Once you have opened this editor you will be presented with a list of all the Network Objects currently available, to learn more about this please see the document on the Network Contract Wizard (NCW) as we will just be going over how to create a network object through the contract wizard. To start, click on the "Create" button near the top and you will be presented with the create UI. In here, we have 3 major fields of interest, the **Name** fields, the **Fields** field, and the **Remote Procedure Calls** field.

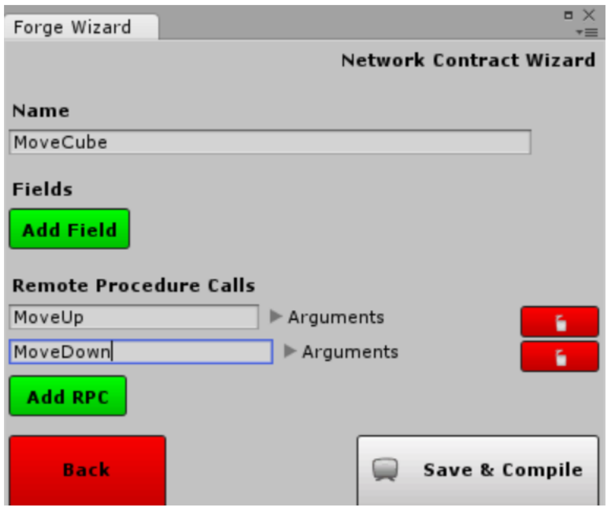
1. The **Name** field is where we create the name for our Network Object and behavior, this is a friendly name that should be written in "Pascal case" to follow the C# convention since it is going to be a part of the class names that are generated.
2. The **Fields** section shows all of the various fields that our network should be aware of.
3. The **Remote Procedure Calls** field is where you will design any Remote Procedure Call (RPC) function signatures. This is going to be our main area of focus for this example.

Setting up the contract option 1

In this option we will create 2 RPC methods with no arguments. One RPC is to move the cube up and the other is to move the cube down. **NOTE: Only pick option 1 or 2 to follow**

Lets begin by naming our Network Object:

1. Lets set the name for our Network object to **MoveCube**
2. Click the **Add RPC** button
3. Name the new RPC **MoveUp**
4. Click the **Add RPC** button
5. Name the new RPC **MoveDown**
6. Click the **Save & Compile** button



Name
MoveCube

Fields

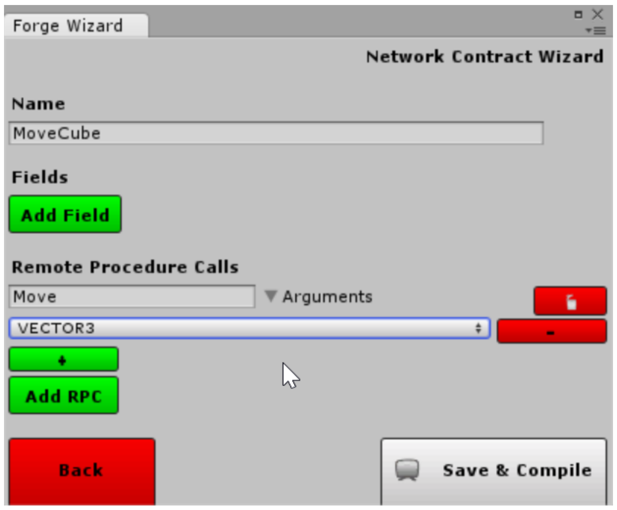
Remote Procedure Calls
MoveUp
MoveDown

Setting up the contract option 2

In this option we will create 1 RPC with 1 argument that denotes the direction to move the cube. **NOTE: Only pick option 1 or 2 to follow**

Lets begin by naming our Network Object:

1. Lets set the name for our Network object to **MoveCube**
2. Click the **Add RPC** button
3. Name the new RPC **Move**
4. Click the arrow to open **Arguments** next to the RPC name input box
5. Click the green button that appears below the RPC name input box
6. Select **Vector3** from the dropdown option list
7. Click the **Save & Compile** button



In this example you may be curious what the **Vector3** selection was for. This particular selection was to set the data types that will be sent as arguments to this method. Since we will be sending a direction to move in, this is a **Vector3**, that means that we would need to pick it. The order of these type selections (if we had more than one) are explicit. That is to say that the order you select them in, is the order that you would pass argument types in, just as if you were writing a method in C#.

Extending Generated Classes

When we use the **Network Contract Wizard (NCW)** we are actually generating a lot of network code based on what has been input into the editor fields, this actually cuts out a lot of work that you would have to do by hand. There is one class in particular that we want to extend from, this class name will be "**MoveCubeBehavior**". The naming convention for this generated class is _____Behavior where "_____" is the name we typed into the NCW. Lets now create a C# file in Unity and write our basic game logic, we will name this file "**MoveCube**".

1. Open the newly created C# file
2. Add **using BeardedManStudios.Forge.Networking.Generated;** to the using statements
3. Derive the class from **BasicCubeBehavior**
4. Write the rest of the logic for the cube as seen below

Code if option 1 was selected

MoveCube

```
using UnityEngine;
using BeardedManStudios.Forge.Networking.Generated;
using BeardedManStudios.Forge.Networking;
using BeardedManStudios.Forge.Networking.Unity;

public class MoveCube : MoveCubeBehavior
{
    private void Update()
    {
        // Move the cube up in world space if the up arrow was pressed
        if (Input.GetKeyDown(KeyCode.UpArrow))
            networkObject.SendRpc("MoveUp", Receivers.All);
        // Move the cube down in world space if the down arrow was pressed
        else if (Input.GetKeyDown(KeyCode.DownArrow))
            networkObject.SendRpc("MoveDown", Receivers.All);
    }

    /// <summary>
    /// Used to move the cube that this script is attached to up
    /// </summary>
    /// <param name="args">null</param>
    public override void MoveUp(RpcArgs args)
    {
        // RPC calls are not made from the main thread for performance,
since we
        // are interacting with Unity enginge objects, we will need to make
sure
        // to run the logic on the main thread
        MainThreadManager.Run(() =>
        {
            transform.position += Vector3.up;
        });
    }

    /// <summary>
    /// Used to move the cube that this script is attached to down
    /// </summary>
    /// <param name="args">null</param>
    public override void MoveDown(RpcArgs args)
    {
        // RPC calls are not made from the main thread for performance,
since we
        // are interacting with Unity engine objects, we will need to make
sure
        // to run the logic on the main thread
        MainThreadManager.Run(() =>
        {
            transform.position += Vector3.down;
        });
    }
}
```

As you can see from the code snippet above an RPC is called using the **networkObject.SendRPC** method. The first argument is the name of the method and the second argument is the receivers of the object which could be set to things like AllBuffered, Others, etc. The moment the RPC method is called it is sent on the network to be replicated to the other clients (including server if called from a client). ***Note**: In this example, it doesn't use a buffered call and it does not actually synchronize the position, so the client should be connected before the cube is moved.* ***Note 2**: Notice that we use **MainThreadManager** to run the move logic in this example, if you have not used the **MainThreadManager** before or would like more information about threading in Unity, please view this page.*

Code if option 2 was selected

MoveCube

```
using UnityEngine;
using BeardedManStudios.Forge.Networking.Generated;
using BeardedManStudios.Forge.Networking;
using BeardedManStudios.Forge.Networking.Unity;

public class MoveCube : MoveCubeBehavior
{
    private void Update()
    {
        // Move the cube up in world space if the up arrow was pressed
        if (Input.GetKeyDown(KeyCode.UpArrow))
            networkObject.SendRpc("Move", Receivers.All, Vector3.up);
        // Move the cube down in world space if the down arrow was pressed
        else if (Input.GetKeyDown(KeyCode.DownArrow))
            networkObject.SendRpc("Move", Receivers.All, Vector3.down);
    }

    /// <summary>
    /// Used to move the cube that this script is attached to
    /// </summary>
    /// <param name="args">
    /// [0] Vector3 The direction/distance to move this cube by
    /// </param>
    public override void Move(RpcArgs args)
    {
        // RPC calls are not made from the main thread for performance,
since we
        // are interacting with Unity engine objects, we will need to make
sure
        // to run the logic on the main thread
        MainThreadManager.Run(() =>
        {
            transform.position += args.GetNext<Vector3>();
        });
    }
}
```

As you can see from the code snippet above an RPC is called using the **networkObject.SendRPC** method. The first argument is the name of the method, the second argument is the receivers of the object which could be set to things like AllBuffered, Others, etc, and the last argument(s) are the arguments for the method. The arguments are mapped to the **object[] args** of the method in the order that they were sent in. The moment the RPC method is called it is sent on the network to be replicated to the other clients (including server if called from a client). ***Note**: In this example, it* *doesn't use a buffered call and it does not actually synchronize the position, so the client should be connected before the cube is moved.* ***Note 2****: Notice that we use* ***MainThreadManager*** *to run the move logic in this example, if you have not used the* ***MainThreadManager*** *before or would like more information about threading in Unity, please view this page.*

Scene Setup

Now that we have done all of the network programming required for our end goal, it is time to setup our scene.

1. Create a new scene in unity
2. Create a cube which will act as the active moving cube
3. Set the position of this cube on the **x**, **y**, and **z** to **0**, **0**, **0** respectively
4. Attach our **MoveCube** script to this cube
5. Save the scene as **MoveCube**
6. Open up the build properties for Unity
7. Add the **MultiplayerMenu** scene as the first scene
8. Add your **MoveCube** scene as the second scene

Test

Now that we have setup our scene and everything else, it is time to test the game.

1. Open the **Build Settings**
2. Click on **Player Settings...**
3. Open the **Resolution and Presentation** section
4. Turn on **Run In Background***
5. Go back to **Build Settings**
6. Click on **Build And Run**
7. Once the game is open, return to the Unity Editor
8. Open the **MultiplayerMenu** scene
9. Click the play button
10. Click the **Host (127.0.0.1:15937)** button on the bottom of the game view
11. Go back to the built game
12. Make sure the host ip address is set to **127.0.0.1**
13. Make sure the host port is set to **15937**
14. Click the **Connect** button
15. Select the server instance (Unity Editor) then press the up and down arrow keys
16. Select the client instance then press the up and down arrow keys

You will see the server movements replicated to the client and the client movements replicated to the server

Basic Instantiation Example

Note: We assume that you have gone through the Basic Moving Cube Example and the Basic RPC Example before going through this example.

This example is not a continuation of other examples and should be treated as if it were done in a new project.

In this example we are going to create a simple cube to act as the player object. The "player" should be able to move their individual cube using the **horizontal** and **vertical** axis input. For this we will create a scene without any cubes in it, then when a connection is made or the server is created, we will spawn a cube "player" for the newly connected client or created server.

What is covered in this example

1. Plan for networking code
2. Use the **"Network Contract Wizard"** to create a network object type with properties
3. Extend the generated class
4. Implement the network code in your custom class
5. Manage who executes what code in the extended class
6. Create a prefab that will be attached to this network code
7. Hook up our prefab to be instantiated
8. Create a global game logic class that will instantiate the player object on the network
9. Host a server
10. Connect with a client directly via IP address and Port

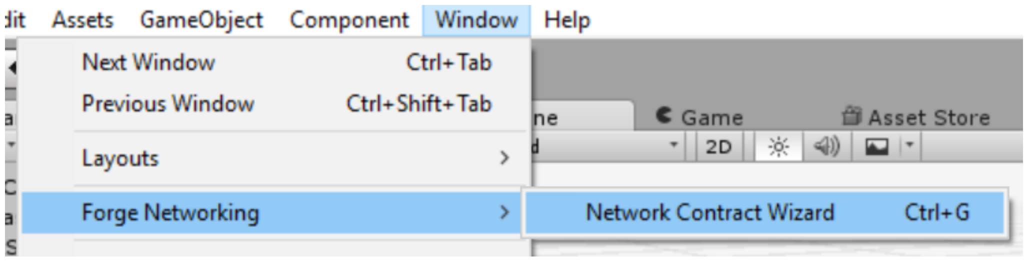
Planning Network Code

So one of the first things we want to think about is our **Network Contract**. This is basically a fancy word for how we design/setup network communication. It is helpful to imagine our final result to our application so that we can properly design for it. In this application when a client connects it will create a player cube for itself, it will also see all of the other connected cubes and their movements:

1. Player creates a cube for itself when connected
2. Synchronize the position and rotation of the player cube to all the clients and server
3. The transformations should be smooth so we need to interpolate them

Network Contract Wizard

Now that we know that we need to sync the **position** and **rotation** of a cube, we can design our network contract for that object. We will first open the **Network Contract Wizard** which is a UI provided by the Bearded Man Studios team to make it easy to design your **network contracts** in a easy way. To open this menu, go into Unity and select "Window->Forge Networking->Network Contract Wizard".

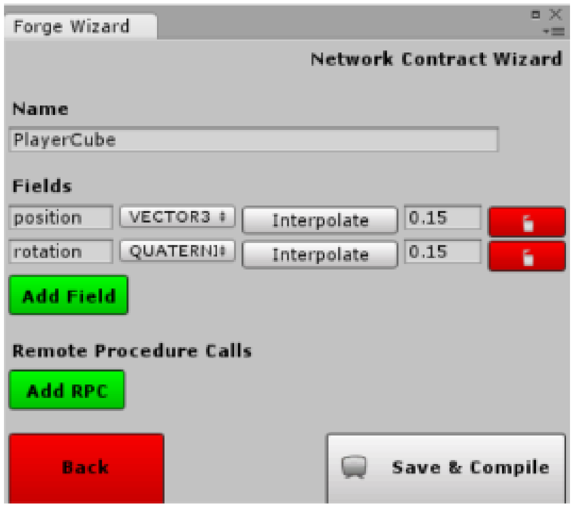


Once you have opened this editor you will be presented with a list of all the Network Objects currently available, to learn more about this please see the document on the Network Contract Wizard as we will just be going over how to create a network object through the contract wizard. To start, click on the "Create" button near the top and you will be presented with the create UI. In here, we have 3 major fields of interest, the **Name** fi elds, the **Fields** field, and the **Remote Procedure Calls** field.

1. The **Name** field is where we create the name for our Network Object and behavior, this is a friendly name that should be written in "Pascal case" to follow the C# convention since it is going to be a part of the class names that are generated.
2. The **Fields** section shows all of the various fields that our network should be aware of. In this case we are going to want to make a **positi on** and **rotation** field which are **Vector3** and **Quaternion** respectively. *Note, you can name these fields whatever you like, these are just friendly variable names for you (and your team) to know what they are for when used*
3. The **Remote Procedure Calls** field is where you will design any Remote Procedure Call (RPC) function signatures. We are not going to go over this field in this tutorial as we do not need it for the goal we are attempting to accomplish.

Lets begin by naming our Network Object:

1. Lets set the name for our Network object to "**PlayerCube**" (without quotes)
2. Click the **Add Field** button
3. Name the new field **position**
4. Set the type to **Vector3**
5. Click the **Interpolate** button
6. Set the interpolate time (the text field that pops up after clicking the **Interpolate** button) as **0.15**
7. Click the **Add Field** button
8. Name the new field **rotation**
9. Set the type to **Quaternion**
10. Click the **Interpolate** button
11. Set the interpolate time (the text field that pops up after clicking the **Interpolate** button) as **0.15**
12. Click the **Save & Compile** button


Forge Wizard

Network Contract Wizard

Name

PlayerCube

Fields

position	VECTOR3	Interpolate	0.15	
rotation	QUATERNI	Interpolate	0.15	

Extending Generated Classes

When we use the **Network Contract Wizard (NCW)** we are actually generating a lot of network code based on what has been input into the editor fields, this actually cuts out a lot of work that you would have to do by hand. There is one class in particular that we want to extend from, this class name will be "**PlayerCubeBehavior**". The naming convention for this generated class is _____Behavior where "_____" is the name we typed into the NCW. Lets now create a C# file in Unity and write our basic game logic, we will name this file "**PlayerCube**".

1. Open the newly created C# file
2. Add **using BeardedManStudios.Forge.Networking.Generated;** to the using statements
3. Derive the class from **PlayerCubeBehavior**
4. Write the rest of the logic for the cube as seen below

PlayerCube

```
using UnityEngine;
using BeardedManStudios.Forge.Networking.Generated;

public class PlayerCube : PlayerCubeBehavior
{
    /// <summary>
    /// The speed that this cube should move by when there are axis inputs
    /// </summary>
    public float speed = 5.0f;

    private void Update()
    {
        // If this is not owned by the current network client then it needs
to
        // assign it to the position and rotation specified
        if (!networkObject.IsOwner)
        {
            // Assign the position of this cube to the position sent on the
network
            transform.position = networkObject.position;

            // Assign the rotation of this cube to the rotation sent on the
network
            transform.rotation = networkObject.rotation;
            // Stop the function here and don't run any more code in this
function
            return;
        }

        // Get the movement based on the axis input values
        Vector3 translation = new Vector3(Input.GetAxis("Horizontal"),
Input.GetAxis("Vertical"), 0);

        // Scale the speed to normalize for processors
        translation *= speed * Time.deltaTime;

        // Move the object by the given translation
        transform.position += translation;

        // Just a random rotation on all axis
        transform.Rotate(new Vector3(speed, speed, speed) * 0.25f);

        // Since we are the owner, tell the network the updated position
        networkObject.position = transform.position;

        // Since we are the owner, tell the network the updated rotation
        networkObject.rotation = transform.rotation;

        // Note: Forge Networking takes care of only sending the delta, so
there
```

```
            // is no need for you to do that manually
        }
    }
```

Setting up prefab

Since we are going to instantiate this object when a player connects, we will need a prefab that identifies this object. To quickly create a prefab that has a cube we will use the standard Unity process.

1. **Right click** on the **Hierarchy**
2. Hover our mouse over **3D Object**
3. Click on **Cube** in the following context menu

Now that we have created a cube we need to set it up with the class **PlayerCube** class we just created

1. Select the cube
2. Click on the **Add Component** button in the **Inspector**
3. Type in **PlayerCube** into the search
4. Click the **PlayerCube** script to attach it

Now that we have setup our player object it is time for us to create the prefab that will be instantiated when clients connect.

1. Select the cube in the **Hierarchy**
2. Drag the cube into the **Project** area to create the prefab
3. Rename this prefab **PlayerCube**, this step is not required, just helps the example stay cohesive

With this we are prepared to setup our **NetworkManager** to support the new instantiation of the object.

Attaching the PlayerCube prefab for instantiation

If you search the project directory you will find a prefab named **NetworkManager**. This is a default prefab we have created for you to get started. You **can** make your own prefab or alter this one if you wish to extend behavior. Now we will go through the process of attaching our created **PlayerCube** prefab to this **NetworkManager**

1. Select the **NetworkManager** prefab in the project
2. Locate the field named **Player Cube Network Object**
3. You will notice that this is an array, set the **Size** to 1
4. Click the radial (circle) on the right of the **Element 0** input field
5. Locate and select the **PlayerCube** prefab

Setup Instantiation & Scene

Now that we have setup our **NetworkManager** we are ready to make our instantiate code. To do this we will create a script that will instantiate our player

1. Open up a new scene
2. Create a new C# script named **GameLogic**
3. Open the newly created C# file
4. Add **using BeardedManStudios.Forge.Networking.Unity;** to the using statements
5. Write the rest of the logic for instantiating the cube (seen below)
6. Attach the newly created script to the **Main Camera** object in the scene (this is just to have the script in the scene, no other particular reason)
7. Save the scene as **GameScene**
8. Add the **MultiplayerMenu** scene as the first scene
9. Add your **GameScene** scene as the second scene

GameLogic

```
using UnityEngine;
using BeardedManStudios.Forge.Networking.Unity;

public class GameLogic : MonoBehaviour
{
    private void Start()
    {
        NetworkManager.Instance.InstantiatePlayerCubeNetworkObject();
    }
}
```

Test

Now that we have setup our scene and everything else, it is time to test the game.

1. Open the **Build Settings**
2. Click on **Player Settings...**
3. Open the **Resolution and Presentation** section
4. Turn on **Run In Background***
5. Go back to **Build Settings**
6. Click on **Build And Run**
7. Once the game is open, return to the Unity Editor
8. Open the **MultiplayerMenu** scene
9. Click the play button
10. Click the **Host (127.0.0.1:15937)** button on the bottom of the game view
11. Go back to the built game
12. Make sure the host ip address is set to **127.0.0.1**
13. Make sure the host port is set to **15937**
14. Click the **Connect** button
15. Select the server game instance (Unity Editor)

Now if you move around the cube in the editor, you will see the movements replicated to the client(s). If you move the cube around in the client(s) you will see the cube moving on the server. Our code has the cube constantly rotating so you will see them doing that as well.

Jump Start Guide

NOTICE
The code and design presented in this example is not intended for production use. Everything in this example **is for learning purposes only**. There may be better ways of doing things, or better network designs that could be done. This example is in no way a reflection of how you should write production code and simply is a tool to help learn the various parts of the Forge Networking Remastered system.

This example will go through all of the major uses of Forge Networking Remastered. You will learn the basic building blocks required to build any online multiplayer game using Forge Networking Remastered (FNR).

Introduction

The game we will be creating for this example is very simple. We will have two or more players that can see each other as capsules and we will have a volume (trigger), in this trigger if our player activates it (triggers it) then it will spawn a sphere to the center of the map that will have a rigidbody and random x and z force. Note that the trigger will need to be destroyed when the ball is spawned in order to prevent it from spawning more than one sphere. When the sphere is collided with y the player, then the sphere is destroyed and we add a point to the player who touched the sphere.

Very First Step

The first step before adding any networking to your game is to identify where the critical points to add networking will be. Below is a list of information we know that will need to be sent across the network in order for the game to play as expected on all connections.

- Spawn player
- Sync player positions
- Sync player names
- Destroy trigger after touch
- Spawn sphere
- Sync sphere position
- Change sphere velocity on spawn
- Add points for colliding with sphere
- Change sphere position after collision

Next we can go through each of our required network fields and determine which ones should be done with an RPC, network instantiate, network destroy, and which ones should be done by synchronizing a variable.

- **Instantiate** Spawn player
- **Variable** Sync player positions
- **RPC** Sync player names
- **Destroy** Remove trigger after touch
- **Instantiate** Spawn sphere
- **Variable** Sync sphere position
- **RPC** Change sphere velocity on spawn
- **RPC** Show the last person who got the ball
- **RPC** Change sphere position and velocity after collision

By doing this we are easily able to see what we should do next when we move onto the Network Contract Wizard (NCW).

Setup the Game

Next, before we setup our networking logic, we will want to setup our base game and all of it's objects and prefabs (No scripting yet).

First we will create the basic world for our players to run around in.

1. Create folder: Scenes
2. Create folder: Scripts
3. Create folder: Prefabs
4. Create folder: Materials
5. Create a new scene
6. Delete the **Main Camera**
7. Save your scene as a new scene into **Scenes** folder
8. Create a material in **Materials** folder
    a. Name: Trigger
    b. Color: 0, 255, 0, 128
    c. Set Rendering Mode to Transparent
9. Create a material in **Materials** folder
    a. Name: GameBall
    b. Color: 0, 0, 255, 255

Do the following in the Scene we just created.

1. Create a cube:
    a. Position: 0, 0, 0
    b. Rotation: 0, 0, 0
    c. Scale: 25, 0.1, 25
    d. Name: Floor
2. Create a cube
    a. Position: -12.5, 5, 0
    b. Rotation: 0, 0, 0
    c. Scale: 0.1, 10, 25
    d. Name: Left Wall
3. Create a cube
    a. Position: 12.5, 5, 0
    b. Rotation: 0, 0, 0
    c. Scale: 0.1, 10, 25
    d. Name: Right Wall
4. Create a cube
    a. Position: 0, 5, -12.5
    b. Rotation: 0, 0, 0

   c. Scale: 25, 10, 0.1
   d. Name: Front Wall
5. Create a cube
   a. Position: 0, 5, 12.5
   b. Rotation: 0, 0, 0
   c. Scale: 25, 10, 0.1
   d. Name: Back Wall
6. Create a cube
   a. Position: 7.5, 5, 7.5
   b. Rotation: 0, 0, 0
   c. Scale: 10, 10, 10
   d. Name: Start Trigger
   e. Set Material to the **Trigger** material we created
   f. Check the **Is Trigger** box in the box collider component on this object
7. Create a sphere
   a. Position: 0, 10, 0
   b. Rotation: 0, 0, 0
   c. Scale: 1, 1, 1
   d. Name: GameBall
   e. Set Material to the **GameBall** material we created
   f. Add a **Rigidbody** component
   g. Save as a Prefab in the **Prefabs** folder
   h. Delete from scene
8. Create an empty Game Object
   a. Position: 0, 0, 0
   b. Rotation: 0, 0, 0
   c. Scale: 1, 1, 1
   d. Name: Game Logic
9. Select the menu item **Game Object>UI>Text**
   a. The **Text** that is created in the **Hierarchy**
   b. Rename it to **Last Scored**
   c. Change the **Text** field to be empty
   d. Set the **Anchor Preset** to **Top Left** *(this is the one with the top left being highlighted in the anchor image)*
   e. Set the **Pivot X** to 0
   f. Set the **Pivot Y** to 1
   g. Set **Pos X** to 0
   h. Set **Pos Y** to 0
   i. Set **Pos Z** to 0
   j. Set **Width** to 500
10. Save the scene

Now we have completed the setup for our game, let's setup our **Build Settings**

1. Open Build Settings from **File>Build Settings** (Ctrl + Shift + B on windows)
2. Add **MultiplayerMenu** scene as the 0th scene index
3. Add your newly saved scene as the 1st index
4. Click **Player Settings...**
   a. Turn on **Run In Background**

Now all that is left to do is setup our **Player** prefab and then we will be ready to jump into setting up our network.

1. In Unity select **Assets>Import Package>Characters**
2. Click the Import button
3. Open the **Standard Assets>Characters>FirstPersonCharacter>Prefabs** directory
4. Drag the **FPSController** prefab into the scene
5. Rename the prefab to **Player**
6. Add a **Mesh Filter** component
7. Add a **Mesh Renderer** component
8. Select the **Capsule** as the **Mesh** in the **MeshFilter** component
9. Open the **Materials** drop down in the **Mesh Renderer**
10. Select **Default-Material** in **Element 0**
11. Add a **Sphere Collider**
    a. Set **Y** to -0.5
    b. Set **Radius** to 0.5
12. Add another **Sphere Collider**
    a. Set **Y** to 0.5
    b. Set **Radius** to 0.5
13. Drag the **Player** GameObject from the **Hierarchy** into the **Prefabs** folder that we made in the beginning
14. Delete the **Player** Game Object from the scene
15. Save the project

Sphere Collider Note
The reason we are adding 2 sphere Colliders is because (at the time of this writing) the Character Controller is not reliable for detecting collisions. You have to "rub" up on the object a bit before it is detected. By adding the sphere colliders to our character, this issue will be resolved.

Now you are ready to start setting up your network contract using the **Network Contract Wizard**.

Network Contract Wizard

The Network Contract Wizard (NCW) is responsible for creating the blueprint of the network communication. We will use this UI to setup the basic classes, fields, and remote procedure calls needed for our network communication. To get started, open the Network Contract Wizard UI by going to **Window>Forge Networking>Network Contract Wizard** within Unity. In this window you will be presented with all the current network classes. If you included the **Bearded Man Studios Inc** examples folder then you should see a couple pre-made options initially.

Now let's get started on making the network contract for our simple game:

1. Click the **Create** button
2. Type in **Player** into the name box
3. Click the **Add Field** button
    a. Type **position** into the text box
    b. Select the drop down
    c. Select **VECTOR3** from the options
    d. Click the **Interpolate** button and leave the value at **0.15**
4. Click the **Add RPC** button
    a. Type **UpdateName** into the text field
    b. Open the **Arguments** cascade
        i. Click the ➕ button
        ii. Type **newName** into the text field
        iii. Click the drop down
        iv. Select **STRING** from the drop down
5. Click **Save & Compile**

What this will do is create and modify some classes in the **Generated** folder of your project. Before we continue to add the other network contracts, let's setup our **Player** class.

1. Open the **Scripts** folder
2. Create a new C# script named **Player**
3. Open the **Prefabs** folder and select the **Player** prefab
4. Click **Add Component**
5. Add the **Player** script to the prefab
6. Select the **Scripts** folder
7. Open the **Player.cs** script you just created
8. Insert the following code

Player.cs

```
// We use this namespace as it is where our PlayerBehavior was generated
using BeardedManStudios.Forge.Networking;
using BeardedManStudios.Forge.Networking.Generated;
using UnityEngine;
using UnityStandardAssets.Characters.FirstPerson;

// We extend PlayerBehavior which extends NetworkBehavior which extends
MonoBehaviour
public class Player : PlayerBehavior
{
    private string[] nameParts = new string[] { "crazy", "cat", "dog",
"homie", "bobble", "mr", "ms", "mrs", "castle", "flip", "flop" };
    public string Name { get; private set; }

    protected override void NetworkStart()
    {
        base.NetworkStart();
        if (!networkObject.IsOwner)
        {
            // Don't render through a camera that is not ours
            // Don't listen to audio through a listener that is not ours
            transform.GetChild(0).gameObject.SetActive(false);

            // Don't accept inputs from objects that are not ours
            GetComponent<FirstPersonController>().enabled = false;

            // There is no reason to try and simulate physics since the
position is
            // being sent across the network anyway
            Destroy(GetComponent<Rigidbody>());
        }

        // Assign the name when this object is setup on the network
        ChangeName();
    }

    public void ChangeName()
    {
        // Only the owning client of this object can assign the name
        if (!networkObject.IsOwner)
            return;

        // Get a random index for the first name
        int first = Random.Range(0, nameParts.Length - 1);

        // Get a random index for the last name
        int last = Random.Range(0, nameParts.Length - 1);

        // Assign the name to the random selection
        Name = nameParts[first] + " " + nameParts[last];

        // Send an RPC to let everyone know what the name is for this
player
        // We use "AllBuffered" so that if people come late they will get
the
        // latest name for this object
        // We pass in "Name" for the args because we have 1 argument that
is to
        // be a string as it is set in the NCW
        networkObject.SendRpc("UpdateName", Receivers.AllBuffered, Name);
    }

    // Default Unity update method
    private void Update()
    {
        // Check to see if we are the owner of this player
        if (!networkObject.IsOwner)
        {
            // If we are not the owner then we set the position to the
            // position that is syndicated across the network for this
player
            transform.position = networkObject.position;

            return;
        }

        // When our position changes the networkObject.position will detect
the
        // change based on this assignment automatically, this data will
then be
        // syndicated across the network on the next update pass for this
networkObject
        networkObject.position = transform.position;
    }

    // Override the abstract RPC method that we made in the NCW
    public override void UpdateName(RpcArgs args)
    {
        // Since there is only 1 argument and it is a string we can safely
        // cast the first argument to a string knowing that it is going to
        // be the name for this player
```

```
            Name = args.GetNext<string>();
        }
    }
```

This is all the code we need to allow for all of the connections to see the movement of the players. The next thing we need is to be able to actually instantiate our **Player** prefab since it will not be in the scene at the start of the game. To do this let's open the **NCW** window again.

1. Click the **Create** button
2. Type in **GameLogic** in the name box
3. Click the **Add RPC** button
    a. Type **PlayerScored** into the text field
    b. Open the **Arguments** cascade
        i. Click the ➕ button
        ii. Type **playerName** into the text field
        iii. Click the drop down
        iv. Select **STRING** from the drop down
4. Click **Save & Compile**

What this will do is create and modify some classes in the **Generated** folder of your project. Before we continue to add the other network contracts, let's setup our **GameLogic** class.

1. Open the **Scripts** folder
2. Create a new C# script named **GameLogic**
3. Select the **Game Logic** Game Object from the **Hierarchy** of our previously saved scene
4. Click **Add Component**
5. Add the **GameLogic** script to the Game Object
6. Select the **Scripts** folder
7. Open the **GameLogic.cs** script you just created
8. Insert the following code

GameManager.cs

```
// We use this namespace as it is where our GameLogicBehavior was generated
using BeardedManStudios.Forge.Networking.Generated;
using BeardedManStudios.Forge.Networking.Unity;
using UnityEngine;
using UnityEngine.UI;

// We extend GameLogicBehavior which extends NetworkBehavior which extends
MonoBehaviour
public class GameLogic : GameLogicBehavior
{
    public Text scoreLabel;

    private void Start()
    {
        // This will be called on every client, so each client will
essentially instantiate
        // their own player on the network. We also pass in the position we
want them to spawn at
        NetworkManager.Instance.InstantiatePlayerNetworkObject(position:
new Vector3(0, 5, 0));
    }

    // Override the abstract RPC method that we made in the NCW
    public override void PlayerScored(RpcArgs args)
    {
        // Since there is only 1 argument and it is a string we can safely
        // cast the first argument to a string knowing that it is going to
        // be the name for the scoring player
        string playerName = args.GetNext<string>();

        // Update the UI to show the last player that scored
        scoreLabel.text = "Last player to score was: " + playerName;
    }
}
```

Now we are able to not only spawn the player, but we are also able to print out the last player that scored to the screen. Talking about scoring, we possibly want to get the ball rolling (if you would excuse the expression) and instantiate the **ball** and serialize it's position to all the clients. However just before that, lets fill out our scoreLabel object on our **Game Logic** Game Object.

1. Select the **Game Logic** from the **hierarchy**
2. Drag the **Last Scored** UI Game Object from the **hierarchy** to the **Score Label** field on the **GameLogic** script

Okay, now we are actually ready to make the ball now! So let's start by opening up the **NCW** window once again.

1. Click the **Create** button
2. Type in **GameBall** into the name box
3. Click the **Add Field** button
    a. Type **position** into the text box
    b. Select the drop down
    c. Select **VECTOR3** from the options
    d. Click the **Interpolate** button and leave the value at **0.15**
4. Click **Save & Compile**

What this will do is create and modify some classes in the **Generated** folder of your project. Before we continue to add the other network contracts, let's setup our **GameBall** class.

1. Open the **Scripts** folder
2. Create a new C# script named **GameBall**
3. Open the **Prefabs** folder and select the **GameBall** prefab
4. Click **Add Component**
5. Add the **GameBall** script to the prefab
6. Select the **Scripts** folder
7. Open the **GameBall.cs** script you just created
8. Insert the following code

GameBall.cs

```
// We use this namespace as it is where our BallBehavior was generated
using BeardedManStudios.Forge.Networking;
using BeardedManStudios.Forge.Networking.Generated;
using BeardedManStudios.Forge.Networking.Unity;
using UnityEngine;

// We extend BallBehavior which extends NetworkBehavior which extends
MonoBehaviour
public class GameBall : GameBallBehavior
{
    private Rigidbody rigidbodyRef;
    private GameLogic gameLogic;

    private void Awake()
    {
        rigidbodyRef = GetComponent<Rigidbody>();
        gameLogic = FindObjectOfType<GameLogic>();
    }

    // Default Unity update method
    private void Update()
    {
        // Check to see if we are the owner of this ball
        if (!networkObject.IsOwner)
        {
            // If we are not the owner then we set the position to the
            // position that is syndicated across the network for this ball
            transform.position = networkObject.position;

            return;
        }

        // When our position changes the networkObject.position will detect
the
        // change based on this assignment automatically, this data will
then be
        // syndicated across the network on the next update pass for this
networkObject
        networkObject.position = transform.position;
    }

    public void Reset()
    {
        // Move the ball to 0, 10, 0
        transform.position = Vector3.up * 10;

        // Reset the velocity for this object to zero
        rigidbodyRef.velocity = Vector3.zero;

        // Create a random force to apply to this object between 300 to 500
or -300 to -500
        Vector3 force = new Vector3(0, 0, 0);
        force.x = Random.Range(300, 500);
        force.z = Random.Range(300, 500);

        // Randomly invert along the number line by 50%
        if (Random.value < 0.5f)
            force.x *= -1;

        if (Random.value < 0.5f)
            force.z *= -1;

        // Add the random force to the ball
        rigidbodyRef.AddForce(force);
    }

    private void OnCollisionEnter(Collision c)
    {
        // We are making this authoritative by only
        // allowing the server to call it
        if (!networkObject.IsServer)
            return;

        // Only move if a player touched the ball
        if (c.gameObject.GetComponent<Player>() == null)
            return;

        // Call an RPC on the Game Logic to print the player's name as the
last
        // player to touch the ball
        gameLogic.networkObject.SendRpc("PlayerScored", Receivers.All,
c.transform.GetComponent<Player>().Name);

        // Reset the ball
```

```
            Reset();
        }
    }
```

With the ball complete, we are finally ready to move onto the last script that we will need to create, the behavior for the trigger that starts the whole game!

1. Open the **Scripts** folder
2. Create a new C# script named **GameTrigger**
3. Select the **Start Trigger** Game Object from the **Hierarchy** of our previously saved scene
4. Click **Add Component**
5. Add the **GameTrigger** script to the Game Object
6. Select the **Scripts** folder
7. Open the **GameTrigger.cs** script you just created
8. Insert the following code

GameTrigger.cs

```
using BeardedManStudios.Forge.Networking.Unity;
using UnityEngine;

public class GameTrigger : MonoBehaviour
{
    private bool started;

    private void Update()
    {
        // If the game started we will remove this trigger from the scene
        if (FindObjectOfType<GameBall>() != null)
            Destroy(gameObject);
    }

    private void OnTriggerEnter(Collider c)
    {
        // Since we added 2 sphere colliders to the player, we need to
        // make sure to only trigger this 1 time
        if (started)
            return;

        // Only allow the server player to start the game so that the
        // server is the owner of the ball, otherwise if a client is the
        // owner of the ball, if they disconnect, the ball will be
destroyed
        if (!NetworkManager.Instance.IsServer)
            return;

        Player player = c.GetComponent<Player>();

        if (player == null)
            return;

        started = true;

        // We need to create the ball on the network
        GameBall ball =
NetworkManager.Instance.InstantiateGameBallNetworkObject() as GameBall;

        // Reset the ball position and give it a random velocity
        ball.Reset();

        // We no longer need this trigger, the game has started
        Destroy(gameObject);
    }
}
```

Finalization of our game!

Now that we have finished all of the network code, we need to tell Forge what prefabs are suppose to be instantiated when we make those **NetworkManager.Instance.Instantiat...** calls.

1. Open **Bearded Man Studios Inc>Prefabs**
2. Select the prefab named **NetworkManager**
3. Locate the various empty fields in the **NetworkManager** component that is attached to this prefab
4. In the **GameBall Network Object** array field, put the **GameBall** prefab from the **Prefabs** folder as the **Element 0** (0th index)
5. In the **Player Network Object** array field, put the **Player** prefab from the **Prefabs** folder as the **Element 0** (0th index)

Congratulations

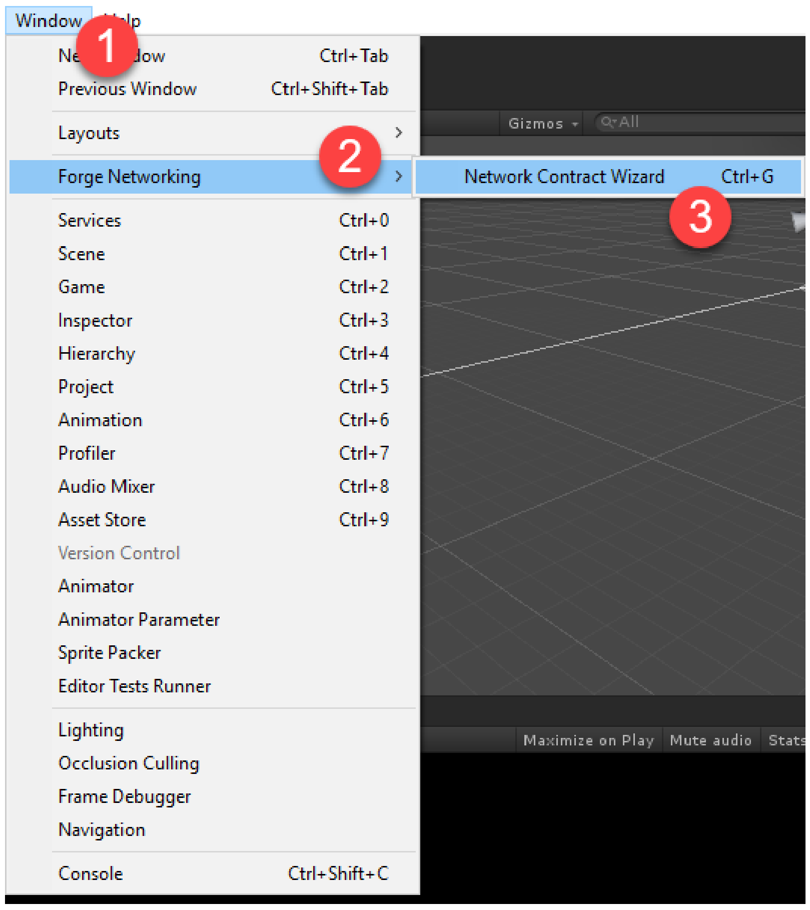
You have completed the steps for this tutorial. All that is left is to build, run and test it out. To do this just build the project as you normally would do within the Unity Editor.

1. Build the project
2. Run 2 instances of the project
3. Select **Host** in one instance
4. Click **Connect** in the second instance
5. You may be prompted to allow access to the application on the firewall, which you will need to accept

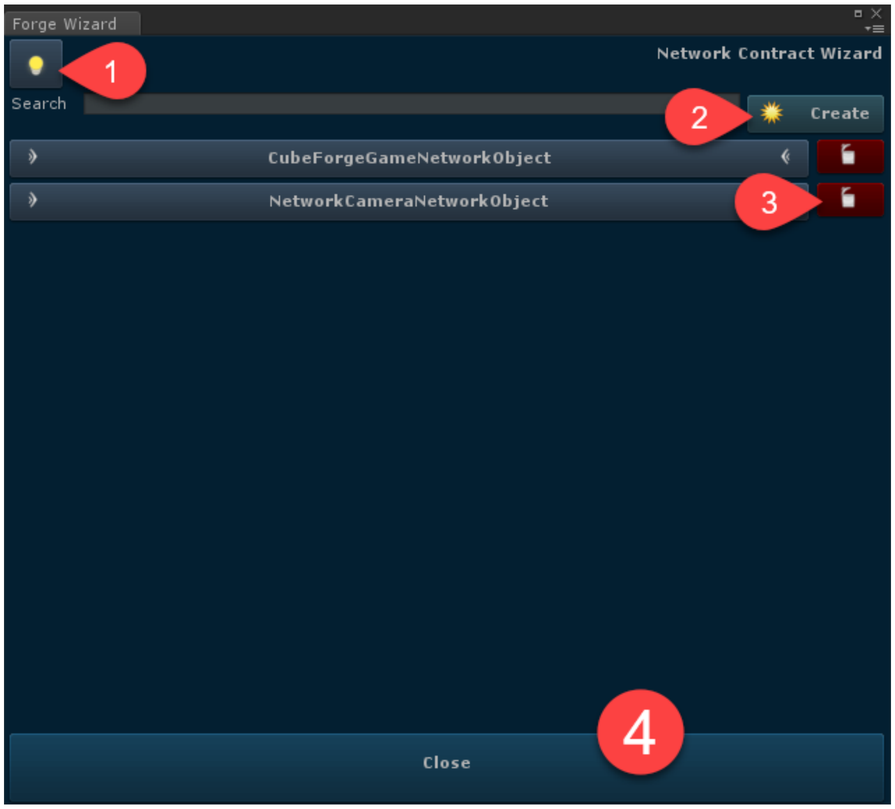
Network Contract Wizard (NCW)

The Network Contract Wizard is the new way for you to implement networking for your game/app.

To do so you will need to go to **Window->Forge Networking->Network Contract Wizard** as shown in the example below.



After opening the window you will be presented with this.



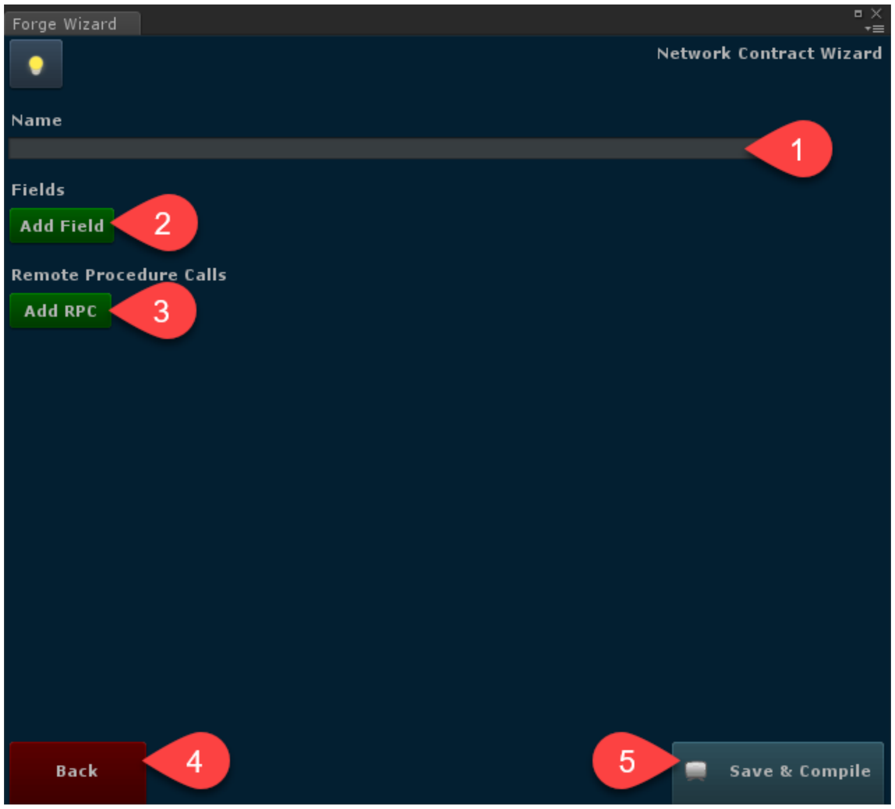
1) Toggle lighting on/off for the editor window only. This will make it easier for your eyes depending on what lighting situation you have.

2) Create, this will make networked objects for you to use for your game/app. (You will spend most of your time hitting this beautifully designed button).

3) Deletion, this will properly delete the networked objects that you have made.

Note: You are able to tap any of the NetwrokObjects to modify them as well.

Creation Menu

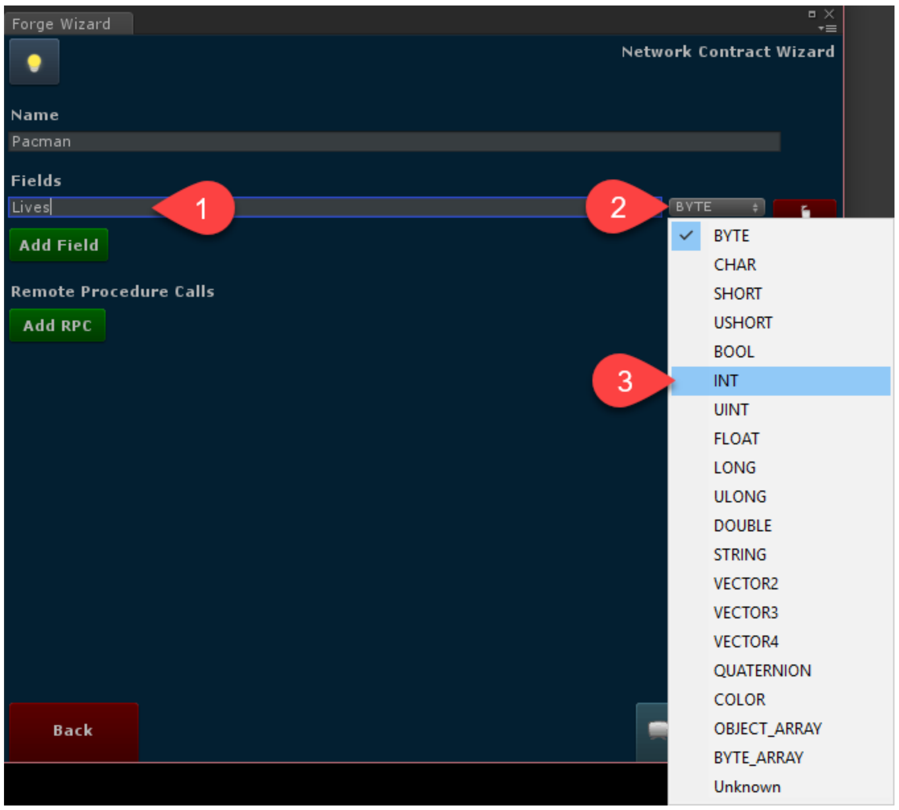


1) Name field for you to name your networked object (Do not use the same name as ones already made)

2) Add fields (This is where you would add variables to sync across the network)

3) Add RPC (This is where you would create callbacks for other clients to messages/data being sent)

Adding a Field



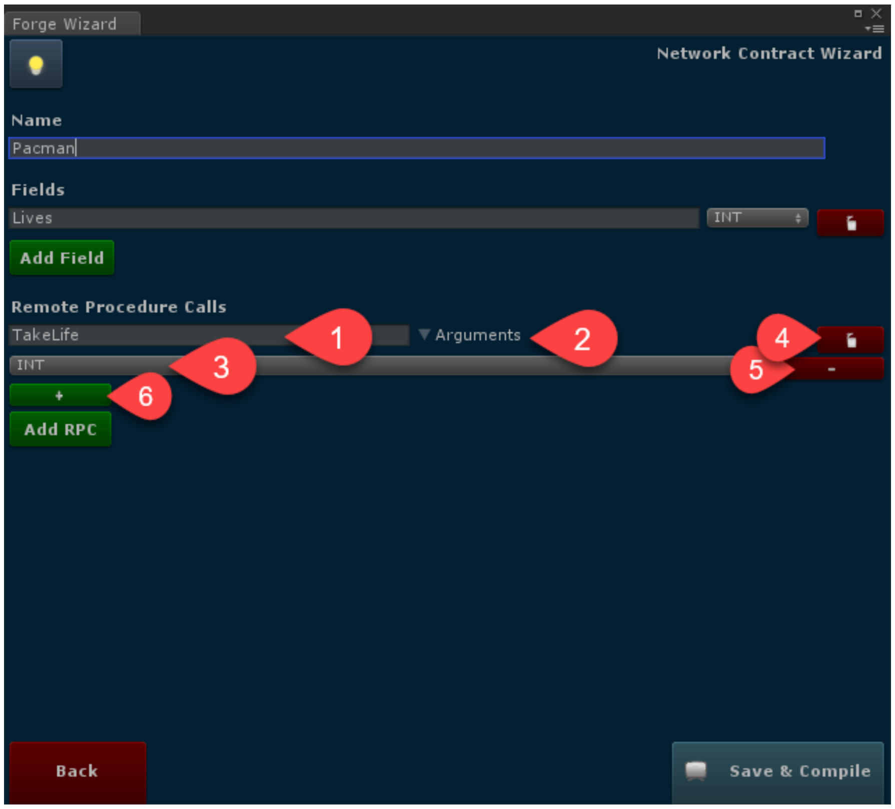
1) The name of the variable

2) The type of variable this will be

3) There are many different types to select from (This is just an example of selecting int for this variable)

Note: The trash bin next to this variable will delete it.

Adding a RPC



1) Name of the RPC

2) Arguments for this RPC that will be sent across the network.

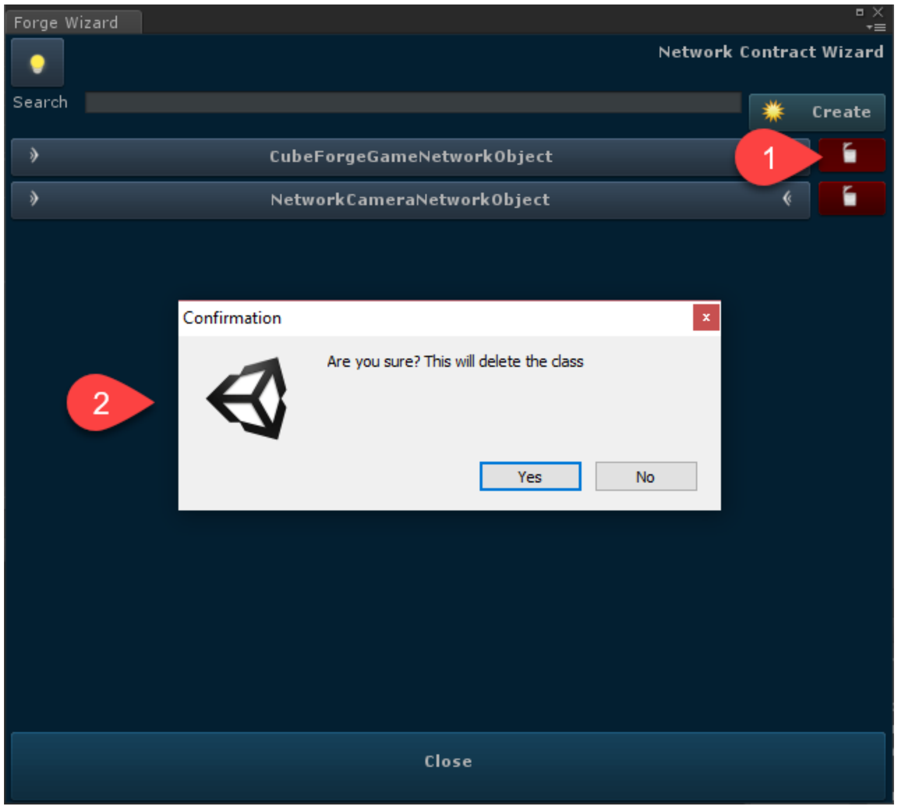
3) Value type that can be passed across the network.

4) Deletion of this RPC

5) Deletion of this Value Type

6) Add more Value Types for the arguments

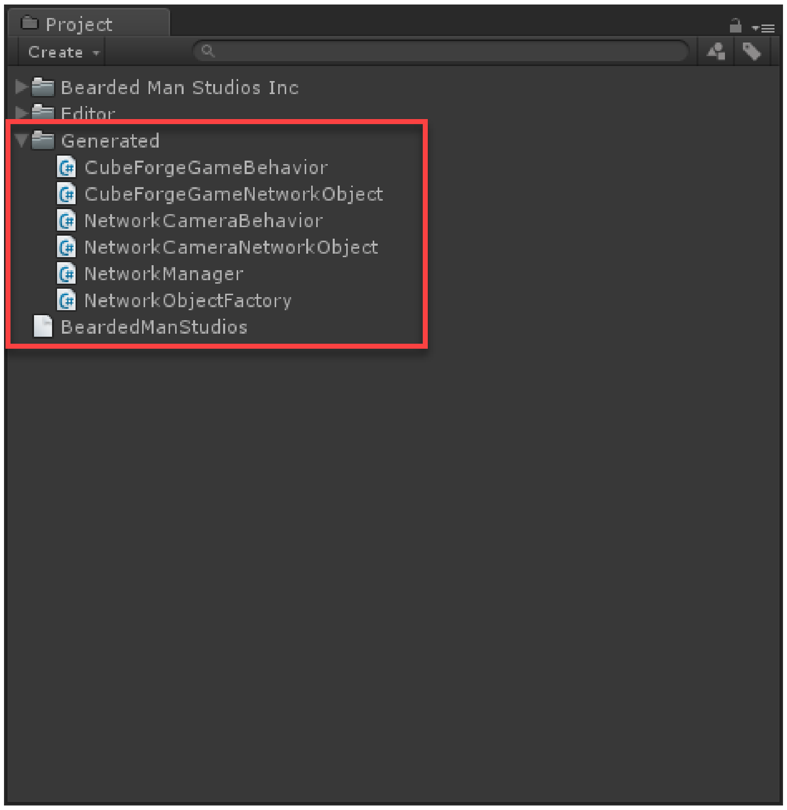
Main Menu Deletion


Forge Wizard
Network Contract Wizard
Search
Create
CubeForgeGameNetworkObject
NetworkCameraNetworkObject
1
2
Confirmation
Are you sure? This will delete the class
Yes
No
Close

1) This will trash this Networked Object correctly.

2) You will be prompted with this window when doing so.

Project Directory



You will notice that all generated code will be located in your project directory under "**Generated**".

Note: This will be changeable from the editor in the future.

Extending Generated Classes

You may have noticed by now that Forge Networking Remastered (FNR) will generate network code through the Network Contract Wizard (NCW). The main purpose of this is to completely remove reflection and make the system easier to test and debug. The generated code will hook into the Forge Networking core framework and setup hooks and connections so that you, as the user of the API, does not have to do all that repetitive work and you can just focus on developing your product.

If you have looked through the generated code you may have seen that the classes that are generated are **partial** classes. If you are unfamiliar with partial classes, we would like to invite you to review the standard MSDN C# documentation for **partial classes**. If you are familiar with the concept of partial classes then you are already a step closer to being able to easily extend the generated code.

In many cases it would be advantageous for us to be able to **add more** code to generated code. This is because we may have multiple different objects inheriting or using the classes that are generated. In this scenario, it would not make sense to write another class and make sure to remember to attach it to every object that uses the class in question. There are scenarios where you would just like to be able to manually add more options to a generated class for further use or for utility sake. To do this, all you need to do is create a new class with the same name as the target class in question. Then make sure to place the keyword **partial** just before you type the **class** keyword.

```
public partial class MyClass…
```

Partial classes are essentially compiled together into one combined class, so it is like editing the source file without having to actually change the source file.

> **WARNING**
> DO NOT make manual edits to any generated class. Any changes that you do to these classes will be overridden on the next **Save & Compile** of the NCW. Please do not place any of your partial class code into the Generated folder either, we can not guarantee that this folder will keep the integrity of it's contents.

Network Object

Changing Ownership

As you may have read in Network Object, the network object has an owner. This owner is responsible for updating the Fields. Since there is one owner of an object, you may want to update who owns an object at runtime. For example, if there is a car on the street, it may be currently owned by the server (because the server spawned the object), however a player gets into the car to drive it, now we would want the player to own the car and update it's variables. To do this we have two methods that are a part of the Network Object and one extra method to override to validate ownership changes on the server.

There are two different methods to use for changing ownership of an object; one for the client and one for the server. The client method is **TakeOwnership** and has no arguments. Any client can call this method on the **networkObject** to request ownership of an object. The method that you would use to force ownership from the server is **AssignOwnership** which has an argument **NetworkingPlayer** (the player who is going to be the new owner for the network object). Only the server can call this method on the **networkObject** due to it's authoritative nature. If a client were to call this method, nothing would happen.

As for verifying an ownership change request, which if you don't do it will just accept the change ownership request, there is one method to override. We live in a world where we just can't trust our clients to send us messages that they should be in the way that we wish them to. Clients may want to cheat the system and try to exploit the network for fun or just so that they can win. Because of this we have created a simple method for you to override in order to check any game critical RPCs that are received from the client and verify them. Before continuing with this part of the documentation, please review the Extending Generated Classes documentation to learn more about **partial** classes.

> **Notice**
> This check is done on a separate thread from Unity's main thread

When we use the Network Contract Wizard (NCW), it will create 2 generated classes; one for our MonoBehaviour and one for our NetworkObject. The one that we want to focus on in this example is the generated NetworkObject. So let's say that we opened up the Network Contract Wizard (NCW) and we created a contract named **Car**; this will generate a NetworkObject class named **CarNetworkObject**. Now in another folder (not in the Generated folder) you will create a new C# script called **CarNetworkObject**. In here you will create your partial class for the **CarNetworkObject** and you will override the **AllowOwnershipChange** method like so:

AllowOwnershipChange

```
protected override bool AllowOwnershipChange(NetworkingPlayer newOwner)
{
    // The newOwner is the NetworkingPlayer that is requesting the
ownership change, you can get the current owner with just "Owner"
}
```

Notice that you need to return a boolean from this function. If you return **true** then the ownership change will be allowed to go through, it will be invoked on the server and on the clients. If you return **false** then the ownership change will not be invoked at all and will be dropped.

Destroying the Network Object

One of the most common actions that you may want to do on the network is to destroy the various network objects that you create. Any object that has a **networkObject** or that derives from one of the generated classes made by the Network Contract Wizard (NCW) can be destroyed on the network. If you have a reference to the object you wish to destroy then it is just a matter of calling one function:

```
networkObject.Destroy();
```

This will not only destroy the Network Object, but it will also call the UnityEngine.GameObject::Destroy method on the gameObject that the networkObject is attached to. If you wish to know when an object is destroyed you can register to the NetworkObject::onDestroy event.

```
networkObject.onDestroy += MyMethod;
```

Fields

The Network Object (most commonly seen as Network Object) is a variable found in the classes that were generated from using the Network Contract Wizard (NCW). When you create a field in the Network Contract Wizard (NCW) for a generated type, then it will be added as a property of that generated type. Below are the currently supported types of fields allowed in a network object field.

Field	Size
byte	8 bits
sbyte	8 bits
short	16 bits
ushort	16 bits
int	32 bits
uint	32 bits
long	64 bits
ulong	64 bits
float	32 bits
double	64 bits
char	8 bits
Vector2	64 bits
Vector3	96 bits
Vector4	128 bits
Quaternion	128 bits
Color	128 bits

Field Usage

When you generate a class it will be generated with a **networkObject** variable. This variable has all of the fields that you described in the Network Contract Wizard (NCW) built into it. For example, if you created a field in the Network Contract Wizard (NCW) that was named position and was selected to be a VECTOR3 then you will have access to it by doing **networkObject.position**. One thing that you will notice when doing this is that this field is actually a C# property, but we will explain this momentarily.

On some fields (such as VECTOR3) you will notice that it has a greyed out button labeled interpolate. If you were to click this, it will turn on interpolation for this field. You can set the interpolation time by assigning the text input field to the right of the button once it is active, default is 0.15. If you are not familiar with interpolation, basically when we send messages across the network we are dealing with millisecond gaps of information, this will cause objects to seem as though they were teleporting or lagging. By using interpolation, you can smooth out these movements to look more natural.

Behind the Scenes

What is going on behind the scenes? Well let's start with where we left off in the last section on how the fields are actually properties. These properties have a getter/setter on each one. The getter will simply return a private field in the generated network object class; however, the setter does a few more actions. When you assign the value of the property, the setter will first set the private field to the value specified. Next a dirty flag will be set to tell the network to syndicate the change for that variable on the next network update for this object. What this dirty flag allows is for FNR to be able to only pick fields that have changed and send those across the network. This reduces the amount of data being sent by a lot depending on how often other variables are updated.

Notes

In the current released version RC1, there is a bug with array types being sent over the network, this is why array types (including strings) are not listed in the type table above. Technically it is considered bad practice to make array types, including strings, as a field for a network object. Array types are better updated and managed using an RPC. Though it is considered bad practice, we will be fixing this bug in future updates to allow full control to be in the developers hands.

Remote Procedure Calls RPC

Clearing Buffered RPCs

When buffering RPC calls you may have noticed that the buffered call exists for the entire life-span of the network object. This is so that critical functions can be performed on an object to clients who connect later (after the RPC was executed). The problem you may run into with this is that you would need to delete the object to reset the level or partly reset the level. This of course presents the problem of managing and instantiating objects remotely. To resolve this issue, you can clear out the buffered RPCs of an object using the **NetworkObject::ClearRpcBuffer.** This will request for the buffered RPCs of a specific object to be cleared on the server if called from the client, otherwise it will immediately clear the buffered RPCs for the network object in question if called from the server.

> **Notice**
> Only the server or the owning player of the given network object is allowed to clear the buffer.

> **Notice**
> Of course, as mentioned above, the RPC buffer of a specific network object will be cleared when that object is destroyed on the network: Destroying the Network Object.

RpcArgs and the RpcInfo "Info" Struct

RpcInfo

If you have done any RPC calls by now, then you will notice that they all are supplied 1 argument being **RpcArgs**. Inside of this RpcArgs you will find an object named **Info** which is of type **RpcInfo**. Basically, what this contains is the various information about the RPC being called. At the time of this writing there are 2 variables provided, the **TimeStep** and the **SendingPlayer**. The TimeStep is of ulong type and is useful for determining at what timestep (millisecond) when the RPC was called on the sender which is often useful in lockstep systems or detecting cheating. The **SendingPlayer** is of type NetworkingPlayer and is most useful on the server. For clients, this variable will always be the server since Forge Networking follows a completely Authoritative Design in how it communicates. For the server the **SendingPlayer** will be the client who sent the RPC.

Example: Let's say that the client wanted to assign its name on the server and so it sends a string via RPC to request the name assignment. Using the **SendingPlayer** from the RpcInfo we can easily get the player object and assign the name.

Using RpcInfo

```
// Client calls this code somewhere
networkObject.SendRpc("AssignName", Receivers.Server, "My New Name");

// The Rpc would look like the following
public override void AssignName(RpcArgs args)
{
    if (!networkObject.IsServer)
        return;

    string playerName = args.GetNext<string>();
    RPCInfo info = args.Info;
    info.SendingPlayer.Name = playerName;
}
```

RpcArgs Arguments

The RpcArgs struct has helper functions to be able to get the variables that were sent across the network as arguments. There are two main functions to achieve this behavior **RpcArgs::GetNext<T>** and **RpcArgs::GetAt<T>**. When you call a RPC the arguments provided are received in the order that they were sent. The **GetAt** call will allow you to provide the index you wish to get. So if you were to do something like **SendRpc(" MyRpcFunc", Receivers.All, "cat", 9, true)** then if you were to call **args.GetAt<int>(1)** you would get the value **9**. The **GetNext** method has an internal counter that is used to quickly go through and get all of the variables. So using the same **MyRpcFunc** example above you could do the following:

GetNext

```
string name = args.GetNext<string>();    // "cat"
int count = args.GetNext<int>();        // 9
bool isBrown = args.GetNext<bool>();    // true
```

RPC Validation by Server

We live in a world where we just can't trust our clients to send us messages that they should be in the way that we wish them to. Clients may want to cheat the system and try to exploit the network for fun or just so that they can win. Because of this we have created a simple method for you to override in order to check any game critical RPCs that are received from the client and verify them. Before continuing with this part of the documentation, please review the Extending Generated Classes documentation to learn more about **partial** classes.

Notice
This check is done on a separate thread from Unity's main thread

When we use the Network Contract Wizard (NCW), it will create 2 generated classes; one for our MonoBehaviour and one for our NetworkObject. The one that we want to focus on in this example is the generated NetworkObject. So let's say that we opened up the Network Contract Wizard (NCW) and we created a contract named **Ball**; this will generate a NetworkObject class named **BallNetworkObject**. Now in another folder (not in the Generated folder) you will create a new C# script called **BallNetworkObject**. In here you will create your partial class for the **BallNetworkObj ect** and you will override the **ServerAllowRpc** method like so:

ServerAllowRpc

```
protected override bool ServerAllowRpc(string methodName, Receivers
receivers, RpcArgs args)
{
    // The methodName is the name of the RPC that is trying to be called
right now
    // The receivers is who the client is trying to send the RPC to
    // The args are the arguments that were sent as part of the RPC message
and what the receivers will receive as arguments to the call
}
```

Notice that you need to return a boolean from this function. If you return **true** then the RPC will be allowed to go through, it will be invoked on the server and on the clients (if specified in Receivers). If you return **false** then the RPC will not be invoked at all and will be dropped.

Unity Integration

Main Threading RPCs

The RPC behaviors in Forge Networking work on the network reading thread for purposes of performance when it comes to reading network data. This means that accessing any Unity specific objects in this thread will cause an error (please see Threading in Unity). Many of our newer users will be unconformable working with function pointers, lambda expressions or threading so we have created an easy way for you to make all RPC calls perform on the main thread. At any point in your application you can assign the Rpc's **MainThreadRunner** object. By doing this you will essentially make all incoming RPC calls run on Unity's main thread. To do this you would do the following in your code.

Assigning Rpc.MainThreadRunner

```
Rpc.MainThreadRunner = MainThreadManager.Instance;
```

For an example of this, please see the **MultiplayerMenu.cs** file and follow the **useMainThreadManagerForRPCs** variable.

If you wish to manually say when to run an RPC on the main thread. You can always use the **MainThreadManage.Run** method to queue actions to perform on the main thread. To understand this further, please review Threading in Unity.

Threading in Unity

If you have dealt with threading before and attempted to use it within Unity, then you probably know by now that you are not able to access the native Unity features from any thread other than the **main thread**. This does raise a bit of a complication when it comes to a completely multithreaded system such as Forge Networking. We've decided that running only on the main thread restricts our users and ourselves from achieving the full potential in performance that we would like. For this reason, we allow the user to manage how code is offloaded to the main thread in their applications.

What is threading?

If you are familiar with threading then you can skip this section.

This is not going to be a complicated explanation of what the processor is doing at a hardware level but a simple abstract explanation so that it is not such an unfamiliar topic. I'm sure by now we are all familiar with machines that say things like **dual-core** or **quad-core** or **octa-core** etc. Imagine if you will that the machine has 2 cpu chips when we say **dual core** if this were the case, you could also imagine that you can run 2 things at the same exact time. This works much like if you hired someone to fold papers, if you hired a second person to fold papers at the same time, then you could get twice the work done in the same amount of time at the same exact time. Now lets imagine that the papers were a variable such as an int, what if both employees finished folding a paper at the same exact time and they both reached for the next paper in the stack? Well only one person can alter one paper at a time, you could wind up with the paper being folded twice!

Now that you have that basic understanding of a simple issue that comes up with parallel computing (threading) you can say, "what if unity messes with the game object, but I do at the same time?". Of course no good will come out of this, this is precisely why Unity does not allow you to manipulate Unity objects from an external thread. There are probably a large number of reasons that Unity has chosen not to simply allow you to lock a mutex, but I digress. For this reason, we have created a helper class named **MainThreadManager** which we will talk more in depth in a later section.

NOTE: Many modern CPUs can have multiple threads per core, for example the i7-6700K has 2 threads per core making a 4 core processor with 8 threads.

What runs on a separate thread in Forge?

In forge we have 2 critical threads for both the client and the server, and 1 extra critical thread for the server (in TCP mode). When in TCP mode the special thread that the server runs is the connection thread. This thread listens for new client connections and will begin the acceptance process from here. The other 2 threads that are shared on client and server, UDP and TCP, is the write and read threads. There is one long lived thread that is used for the reading of network messages. This reading thread will also execute RPC methods, read message events, and so forth. Almost everything that is processed on the network can be traced back to the read thread. The second thread, the write thread, is shared for all players. This thread is shared so that it can support thousands of connections/players at a time on lower end CPUs. This thread is an on-demand thread. It starts up when messages are being sent, and it shuts down when there are no more messages being sent. The client also uses the same write thread logic, however it is not as active as the server write thread for obvious reasons (it only communicates with the server and nobody else).

Currently the only network communication that you need to worry about as an end user that get's called on a separate thread is an RPC.

Running Unity specific code on the main thread

We have created a helper class for you to be able to offload any logic to the main thread from a separate thread. This helper class is called **MainThreadManager** and there are 2 main ways that you can use this class. The entry point for both methods of use is the static method **Run**; see the code snippets below for practical uses. The examples below emulate a RPC method's contents.

Note
Be sure to add this using statement to the top of your code: **using BeardedManStudios.Forge.Networking.Unity;**

MainThreadManager.Run Function Pointer

```
public override void MyCustomRPC(object[] args)
{
    // Register the private function within this class to be called on the
main thread
    MainThreadManager.Run(OtherFunction);
}

private void OtherFunction()
{
    Debug.Log("Hello World!");
}
```

You can see that the above example requires another accessible method in order to pass it into the main thread manager's Run method.

MainThreadManager.Run Lambda Expression

```
public override void MyCustomRPC(object[] args)
{
    // Setup a temporary method call (lambda expression) to be executed on
the main thread
    MainThreadManager.Run(() => { Debug.Log("Hello World!"); });
}
```

The above is the preferred **method**

The lambda expression is a native C# feature that allows you to essentially create an inline function at runtime. Please see this website or the official documentation for more information on lambda expressions.

What does the Main Thread Manager do?

The Main Thread Manager is actually a pretty small and simple singleton class. When you send a method pointer or inline expression into the Run method it will be added to a queue. The Main Thread Manager is a Unity Game Object and will automatically create itself if one is not created already. Every **FixedUpdate** for this object, it will check to see if there are any pending methods in the queue, if so it will run them and remove them from the queue. By running these methods in the **FixedUpdate** they are automatically ran on the main thread.

Debugging

Network Logging

Forge Networking has it's own logging system built in for the purposes of debugging and tracing any errors that may happen during network communication. When an exception is thrown on the networking layer, the logging system will pick that up and process the log. There are currently 2 main forms of logging that is built into the system, the in-game logging and the file system logging. We do not recommend using the in-game logging for any production environment, however we do suggest that you enable file logging for production environments.

The Forge Networking logger will only log network exceptions so it will not have any impact on performance, except if there are many exceptions being thrown (which shouldn't happen unless there is a critical process error in the game). This log does a little more than just log network exceptions however; it also logs exceptions that are thrown by your code during the network transport time. Things such as an invalid parameter to an RPC or an invalid cast being performed by your code. Because of this, we recommend always having this log available when testing your code in order to reduce support tickets on our end and in order to keep you as productive as possible towards your awesome product!

How To Enable Logging

1. Navigate to Bearded Man Studios IncScripts->Logging->ResourcesBMSLogger
2. Having the BMSLogger selected, you can then check 'Log To File'

This will then allow you to log all exceptions and common networking debug logs into the logging folder (including logs that you write).

How to Enable Visible Logging

Note: This will allow you to see the logs in the build itself on any device.

1. Navigate to Bearded Man Studios IncScripts->Logging->ResourcesBMSLogger
2. Having the BMSLogger selected, you can then check 'Logger Visible'

The final location of this log file will be dependent on whether it is in the editor or a build.

Editor location: Root Unity DirectoryAsset Logsbmslog.txt

Windows Build Location: ExecutableDirectoryExecutableName_DataLogsbmslog.txt

Using the Logger yourself for development purposes

Example Logging

```
BeardedManStudios.Forge.Logging.BMSLog.Log("ANYTHING YOU WANT TO LOG
HERE!");
BeardedManStudios.Forge.Logging.BMSLog.LogFormat("FOLLOWING A FORMAT
[{0}]", "ANYTHING YOU WANT TO LOG HERE!");
BeardedManStudios.Forge.Logging.BMSLog.LogWarning("ANYTHING YOU WANT TO LOG
HERE!");
BeardedManStudios.Forge.Logging.BMSLog.LogWarningFormat("FOLLOWING A FORMAT
[{0}]", "ANYTHING YOU WANT TO LOG HERE!");
BeardedManStudios.Forge.Logging.BMSLog.LogException([System.Exception]);
BeardedManStudios.Forge.Logging.BMSLog.LogException("ANYTHING YOU WANT TO
LOG HERE!");
BeardedManStudios.Forge.Logging.BMSLog.LogExceptionFormat("FOLLOWING A
FORMAT [{0}]", "ANYTHING YOU WANT TO LOG HERE!");
```

Above is code examples of how to call the logger to use it for logging purposes.

Note: All exceptions will automatically be logged, so put them in places that should never be called frequently, useful for testing on builds and figuring out what went wrong on the network.

NetWorker

Thread Safe Player Iteration

Sometimes it is important for our server to go through every player on the network and perform some action for them. Being that the players can connect asynchronously to the server, the **Players** list can be altered at any point, even while we are currently iterating through the players. There are two ways to be able to go through the players in a thread-safe manor **WHICH YOU MUST ALWAYS DO**. You either will be required to lock the players before performing any actions on them, or you can use the provided helper method **NetWorker::IteratePlayers**.

If you were to use a simple lock, then let's say we had a reference to a **NetWorker** with the variable name myNetWorker. Now you have direct access to the players via the myNetWorker.Players getter, however it is not thread safe to just simply access them in this way; you instead will lock them as seen in the following:

Locking

```
lock (myNetWorker.Players)
{
    // Do your player iteration logic here
}
```

The second way would be to use the **NetWorker::IteratePlayers** method as described above. There are two ways that you can do this, the first being to provide a lambda expression [INSERT LINK] for quick inline actions and the other is to provide a function pointer.

Lambda Expression

```
// This is an example of using a lambda expression
myNetWorker.IteratePlayers((player) =>
{
    // Do your player iteration logic here
});
```

Function Pointer

```
// This is an example of using a function pointer
myNetWorker.IteratePlayers(GoThroughPlayers);

// ...

private void GoThroughPlayers(NetworkingPlayer player)
{
    // Do your player iteration logic here
}
```

NAT Traversal Server

NAT Hole Punching

Nat hole punching is a way for users behind a firewall to be able to connect to each other without requireing them to open their firewall ports on their router. To do this, it requires a special server that is dedicated to registering hosts and keeping an open communication with that game host. When a client requests to join a host that is behind a NAT they will communicate with the NAT service host. This NAT service host will then act as a delegate to request that the game host begin communications with the requesting client.

NOTICE

To enable NAT hole punching you are required to host an accessible NAT hole punching server.

When you look at the **Connect** method of the **ServerUDP** and **ClientUDP** classes you will notice that there are 2 optional parameters related to nat traversal. One is the NAT host address and the other is the NAT server port. When you provide a NAT host address to the **Connect** method on a **ServerUDP** object, you will be telling the server where it needs to register itself and you will be opening communication with the NAT server for it to forward client connection requests. When you provide a NAT host address to the **Connect** method on a **ClientUDP** object, you will be telling the client to communicate with that specific NAT host to open communications with the desired server.

Setting the correct NAT host in the **Connect** method of the **ServerUDP** and **ClientUDP** is all you will need to do in order to enable NAT hole punching in your application.

NOTICE
The NAT host address and port must be the same on both the server and the client.