

1. User Manual	3
1.1 Asset Store Invoice Registration	3
1.2 Developer Portal	3
1.2.1 Logging in for the First Time	3
1.3 Community	4
1.4 Getting Started With Forge Networking	4
1.4.1 Connecting to a Server	5
1.4.2 Custom Networked MonoBehavior	7
1.4.3 Hosting a server	10
1.4.4 Proper Disconnection Of Sockets	11
1.4.5 Step by step example	11
1.5 Simple Networked Mono Behavior	13
1.5.1 Is Owner	13
1.5.2 BRPC (Remote Procedure Calls)	14
1.5.3 Owner Id	15
1.5.4 IsServerOwner	16
1.5.5 Current RPC Sender	17
1.5.6 Is Setup	18
1.5.7 Networked Id	19
1.5.8 Owning Net Worker	20
1.5.9 Owning Player	21
1.5.10 Don't Destroy On Load	22
1.5.11 Locate	23
1.5.12 Network Destroy	23
1.5.13 Get RPC	24
1.5.14 Queue RPC For Instantiate	24
1.5.15 Cleanup	24
1.5.16 Reflect	24
1.5.17 Owner Update, Non-owner Update, Owner Fixed Update, Non-owner Fixed Update	24
1.5.18 InvokeRPC	25
1.5.19 AuthoritativeRPC, AuthoritativeURPC, RPC, URPC	25
1.5.20 Serialize and Deserialize	28
1.5.21 NetworkedBehaviors	28
1.5.22 Destroy On Disconnect	28
1.5.23 Allow Ownership Change, Change Owner	29
1.5.24 GenerateUniqueID	30
1.5.25 Reset For Scene	30
1.5.26 Reset All	31
1.5.27 ResetBufferClear	31
1.5.28 Initialize	31
1.5.29 Delay Initialize	31
1.5.30 NetworkStart	31
1.5.31 Setup Objects	32
1.5.32 Setup	32
1.5.33 Disconnect	32
1.5.34 OnDestroy	33
1.5.35 OnApplicationQuit	34
1.5.36 NetworkDisconnect	34
1.6 Networked Monobehavior	35
1.6.1 Deserialize	35
1.6.2 Input Requests	35
1.6.3 Manual NetSync	38
1.6.4 NetSyncToServer	38
1.6.5 Network Controls	39
1.6.6 Serialized	40
1.7 Master Class Beginner	40
1.7.1 Initializing the Firewall Request	40
1.7.2 Remote Procedure Calls	41
1.7.3 Control Access With IsOwner	42
1.7.4 SimpleNetworkedMonoBehavior vs NetworkedMonoBehavior	42
1.7.5 Network Instantiation	43
1.7.6 Buffered Networking Instantiate	44
1.7.7 Networking Destroy	45
1.7.8 Capturing Exceptions	46
1.7.9 Serializing Fields and Properties	51
1.7.10 Spawn Remote Object	54
1.7.11 Owning Updates	54
1.7.12 Stop Interpolate on Independent NetSync	55
1.7.13 HTTP Library	56
1.7.14 Binary	58
1.8 Master Class Intermediate	58
1.8.1 Using The Cache System	58

1.8.2 Working With Authoritative Server Option	59
1.8.2.1 Authoritative Sync Thresholds	63
1.8.2.2 Working with Floating Point Inputs	64
1.8.3 Packet Drop and Network Latency Simulations	66
1.8.4 Auto Launcher	67
1.8.5 Forge Transport Object	68
1.8.6 Proximity Based Updates and Events	69
1.9 Walkthroughs	71
1.9.1 Tic Tac Toe	71
1.10 Write Custom (Custom Data Serialization)	75
1.10.1 Write Custom - Sending Classes Across the Network	75
1.10.2 Write Custom - Unique Identifier	79
1.11 Self and Cloud Hosting	79
1.11.1 Headless Linux Server	79
1.12 Forge Web Server (Addon)	83
1.12.1 Overview of Forge Web Server	83
1.12.2 See Server Statistics and Kick a Player	84
1.12.3 Webpages Overview and Editing	84
1.12.4 MVC Controllers and Actions	84
1.12.5 How to use Replacement Variables	85
1.12.6 Writing Custom Web Server Commands	85
1.12.7 Creating a Custom Controller and Action	86
1.13 Master Server	87
1.13.1 Building a Master Server from Unity	87
1.13.2 Registering with the Master Server	87
1.13.3 Getting Host List from Master Server	88
1.13.3.1 Getting Host List from Master Server - Pagination	88
1.13.4 Removing Host From Master Server	89
1.13.5 Updating Player Count on Master Server	89
1.13.6 Re-Registering Info on Master Server	89
1.14 Forge Utilities	89
1.14.1 Chat System	90

User Manual

Welcome to Forge Networking, 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.

Asset Store Invoice Registration

The Unity Asset Store is a great place to get addons for the Unity Game Engine and many of our users have purchased Forge Networking from this location. The only flaw is that each asset submission requires the approval of Unity which usually takes 3-5 business days. Since we submit new versions at a high velocity the Asset Store can easily have an out of date version of Forge Networking which will be missing features that we have listed in our documentation (since we actually update our docs soon after making new features). In order to get the latest version and know that you are up to date will all of the latest bug fixes and features we recommend that you register your Asset Store invoice on our site to unlock downloading of the latest version.

Note: We require either a purchase through our actual website OR for you to register your Unity Asset Store Invoice Number in your profile before you are able to use our website forums.

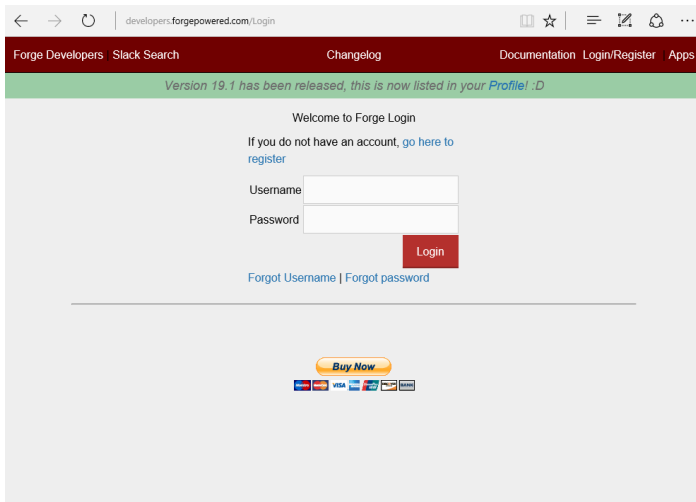
1. Register if you have not already on our developer portal ([here](#))
2. Log into the Forge Networking developer portal ([here](#))
3. Go to your [Developer Profile](#)
4. Locate the section labeled "Register Your Copy Of Forge Networking"
5. Locate the input field next to "Unity Asset Store Invoice #"
6. Input your Unity Asset Store Invoice Number into this input box
7. Click the "Register" button

Developer Portal

Logging in for the First Time

First go to <http://developers.forgepowered.com> that is our homepage for Forge Networking.

Next you will want to go to Login/Register in the top right.

The screenshot shows a web browser window with the URL 'developers.forgepowered.com/Login'. The page has a dark red header with navigation links: 'Forge Developers', 'Slack Search', 'Changelog', 'Documentation', 'Login/Register', and 'Apps'. Below the header is a green banner with the text 'Version 19.1 has been released, this is now listed in your Profile! :D'. The main content area is light gray and contains a 'Welcome to Forge Login' message. It includes a link 'If you do not have an account, go here to register'. There are two input fields for 'Username' and 'Password', followed by a red 'Login' button. Below the login fields are links for 'Forgot Username' and 'Forgot password'. At the bottom of the page, there is a 'Buy Now' button with a yellow background and a row of payment logos including Visa, Mastercard, and others.

From here you will be asked to enter your login credentials, if you haven't made an account with us before, then tap the link 'go here to register'.

Registering is super simple, just fill out the boxes below :D

Username	<input type="text"/>
Email	<input type="text"/>
Verify Email	<input type="text"/>
Password	<input type="text"/>
Verify Password	<input type="text"/>
Agree	<input type="checkbox"/> I agree to the terms

[Sign Me Up!](#)

You will then need to fill out all the information for you to make an account. After that, you will need to register your invoice on the [Forge Developer Profile Page](#).

There you will see 'Register Your Copy Of Forge Networking', In the input field next to the 'Unity Asset Store Invoice#', you will input your Asset Store Invoice here.

Next, you will click register and you have successfully registered Forge Networking to this account.

Please note that we do not allow multiple uses with the same invoice #, as Forge is unlimited use on as many projects as you like, it is still per-developer basis.

Youtube video for demonstration - <http://youtube.com/watch?v=QhFxhw2EroA>

Community

The Forge Networking community is one of the best and largest communities for an Asset Store addon and we are proud to be able to be a part of it. The community has grown well beyond just a place to talk about Forge Networking and has become a place where people work together, laugh, share, and socialize. Don't get us wrong though, it is a place of serious business with some hard hitting questions, bug reports, answers and everything else for networking! One of our goals from the beginning was not to only create a community of network programmers, but to create a community of developers in general. In this sense we hope that you too will join the community!

How to login to the Wiki

Video - <https://youtu.be/qoErMS1DdrE>

You are not able to register directly on the Wiki. All you need to do is register an account on this developer portal and then register your version of Forge Networking (unless you purchased through the site). Once that is complete, you can then login to the wiki by using the username/email and password you signed up with.

How to login to Slack

Slack is a tool that we here at Forge Networking use in order to communicate in real time with you, the developers. Slack is a feature of those who are registered owners of Forge Networking. When you purchase Forge Networking through the website, you are automatically sent an email invite to the Slack team within 24 hours (though it is usually instant). If you purchased Forge Networking through the Asset Store then you will need to register on the developer website using your Asset Store invoice number. See [this part of the documentation](#) to learn how to register your invoice number on the developer site.

Note: You can also head over to <https://slack.com/signin/find> and input the email you used to register on the Forge developer site to start the login/registration process for Slack and the Forge Networking team.

Getting Started With Forge Networking

Most server architectures are either Peer-to-Peer or Server-Client based.

Forge networking is mainly a Server-Client based networking solution where you have one dedicated host and other would connect to him. Now there is a Peer-to-Peer option by enabling Nat punch-through and having a dedicated hosting platform to match clients together rather than act as a host for clients.

Most data is being sent from their payload of bytes from the client to the server and from the server to the client.

Packets are then interpreted by the server for validation and then sent to the remaining clients if chosen to do so for communication.

Highly recommend that Forge Network be used for a Server-Client based solution rather than a Peer-to-Peer, because we don't like cheaters for them to send invalid information to both clients and Server validation is the best way to prevent cheating.

Youtube video for demonstration: <https://www.youtube.com/watch?v=3rhLnBZQngw>

Connecting to a Server

Main Method Call

Connecting to a server/host is as simple as calling one function `Networking.Connect` at any time. Below is a sample of the method and its arguments.

Forge-Connecting-to-Server.cs

```
// Arguments:
//  string <IP_ADDRESS>
//  ushort <PORT_NUMBER>
//  Networking.TransportationProtocolType <PROTOCOL_TYPE>
//  optional bool <IS_WIN_RT> [default false]

Networking.Connect("127.0.0.1", 15937,
Networking.TransportationProtocolType.UDP, true);
```

Connection Events

When you attempt to host a server you can register connection events, register connection failure events, or check if the connection is currently established.

To check if a connection is established, you can use the following:

Forge-Check-Connected.cs

```
if (Networking.Connected)
{
    // TODO:  Fancy stuff
}
```

To register a delegate to the connected event (which fires as soon as the port is bound and listening) you just need to assign the event directly to the socket.

Forge-Register-Connected.cs

```
// Note: 15937 is the port number of the socket you started the connection
on
Networking.Sockets[15937].connected += MyMethod;

void MyMethod()
{
    Debug.Log("Connection Established");
}

// Or you can in-line if you wish

Networking.Socket[15937].connected += delegate
{
    Debug.Log("Connection Established");
};
```

Disconnection Events

I think we all need to know when the network has died so that we can not have our game simply crash. So, in order to find out when we are disconnected, we will need to register the `Networking.Socket[15937].disconnected` event.

Forge-Register-Disconnected.cs

```
// Note: 15937 is the port number of the socket you started the connection
on
Networking.Sockets[15937].disconnected += MyMethod;

void MyMethod()
{
    Debug.Log("Connection Closed or Lost");
}

// Or you can in-line if you wish

Networking.Sockets[15937].disconnected += delegate
{
    Debug.Log("Connection Closed or Lost");
};
```

You can also use the `Networking.Sockets[15937].serverDisconnected` event to execute logic for when the server disconnects the player for any given reason.

Forge-Register-Client-Disconnect.cs

```
// Note: 15937 is the port number of the socket you started the connection
on
Networking.Sockets[15937].serverDisconnected += ByeBye;

void ByeBye(string reason)
{
    Debug.Log("The server has disconnected you because of: " + reason);
}

// Or you can in-line if you wish
Networking.Sockets[15937].serverDisconnected += delegate(string reason)
{
    Debug.Log("The server has disconnected you because of: " + reason);
};
```

Custom Networked MonoBehavior

Serializing Custom Instance Variables

With our system you are able to serialize any variables in your class (public, protected or private). This allows you to update your custom variables across the network. To do this you just need to send your variables getters and setters through the AddNetworkVariable method. Below is an example of a simple class that will update a variable across the network the "long" way.

Forge-SendMyVarsMethods.cs

```
using UnityEngine
using BeardedManStudios.Network;

public class SendMyVarsMethods : NetworkedMonoBehavior
{
    public int number = 53;
    private Vector3 direction = new Vector3(15.3f, 3.74f, 9.5f);

    protected override void Awake()
    {
        base.Awake();

        AddNetworkVariable(GetNum, SetNum);
        AddNetworkVariable(GetDirection, SetDirection);
    }

    private int GetNum()
    {
        return number;
    }

    private void SetNum(int newVal)
    {
        number = newVal;
    }

    private Vector3 GetDirection()
    {
        return direction;
    }

    private void SetDirection(Vector3 newDir)
    {
        direction = newDir;
    }
}
```

Now that you have seen the "long" way of serializing your variables across the network, we can explore the "short" way which will achieve the same exact results through [lambda expressions](#).

Forge-SendMyVarsLambdas.cs

```
using UnityEngine
using BeardedManStudios.Network;

public class SendMyVarsLambdas : NetworkedMonoBehavior
{
    public int number = 53;
    private Vector3 direction = new Vector3(15.3f, 3.74f, 9.5f);

    protected override void Awake()
    {
        base.Awake();

        AddNetworkVariable(() => number, x => number = (int)x);
        AddNetworkVariable(() => direction, x => direction = (Vector3)x);
    }
}
```

Serializing Variables from Other Components

Sometimes we do not have (or want to modify) other components (classes) that are on an object, yet, we still want to be able to send variables from those classes across the network. In this example I will be using the "short" (lambda) version of serialization, though the "long" form will work just as well here.

Forge-SendOtherComponentVars.cs

```
using UnityEngine
using BeardedManStudios.Network;

public class SendOtherComponentVars : NetworkedMonoBehavior
{
    protected override void Awake()
    {
        base.Awake();

        // Let's serialize the light intensity of this object across the network
        AddNetworkVariable(() => light.intensity, x => light.intensity = (int)x);

        // Let's serialize the position of this transform
        AddNetworkVariable(() => transform.position, x => transform.position =
(Vector3)x);
    }
}
```

Notes About Serializing Variables

One of the main things about our network system is that the variables that you are serializing across the network are **ONLY** sent when they have been **CHANGED**. This means you save a lot on bandwidth! It is also important to note that all of the serialization that you do in this derivative of the `NetworkedMonoBehavior` is controlled by the throttle that you use for this class (the `networkTimeDelay`).

Hosting a server

Main Method Call

Creating a server/host is as simple as calling one function `Networking.Host` at any time. Below is a sample of the method and its arguments.

Forge-Starting-Server.cs

```
// Arguments:
//  ushort <PORT_NUMBER>
//  Networking.TransportationProtocolType <PROTOCOL_TYPE>
//  int <MAX_PLAYERS>

Networking.Host((ushort)15937, Networking.TransportationProtocolType.UDP,
31);
```

Connection Events

When you attempt to host a server you can register connection events, register connection failure events, or check if the connection is currently established.

To check if a connection is established, you can use the following:

Forge-Check-Connected.cs

```
if (Networking.Connected)
{
    // TODO:  Fancy stuff
}
```

To register a delegate to the connected event (which fires as soon as the port is bound and listening) you just need to assign the event directly to the socket.

Forge-Register-Connected.cs

```
// Note: 15937 is the port number of the socket you started the connection
on
Networking.Sockets[15937].connected += MyMethod;

void MyMethod()
{
    Debug.Log("Connection Established");
}

// Or you can in-line if you wish

Networking.Socket[15937].connected += delegate
{
    Debug.Log("Connection Established");
};
```

Disconnection Events

I think we all need to know when the network has died so that we can not have our game simply crash. So, in order to find out when we are disconnected, we will need to register the `Networking.Socket[15937].disconnected` event.

Forge-Register-Disconnected.cs

```
// Note: 15937 is the port number of the socket you started the connection
on
Networking.Sockets[15937].disconnected += MyMethod;

void MyMethod()
{
    Debug.Log("Connection Closed or Lost");
}

// Or you can in-line if you wish

Networking.Sockets[15937].disconnected += delegate
{
    Debug.Log("Connection Closed or Lost");
};
```

Proper Disconnection Of Sockets

When you host a server or connect to a server, this is often done through the `Networking.Host` or the `Networking.Connect` static methods. These methods will return a `NetWorker` object that has a `Disconnect` method built in, however this is often not the method you want to call.

The disconnect method described above is used to disconnect a socket that you created yourself without the use of `Networking.Host` or `Networking.Connect`. So you may be wondering how you can go about disconnecting your `NetWorker` when you want to without causing any issues. We have created a `Networking.Disconnect` method to make this easy. Basically all that is needed is for you to call `Networking.Disconnect` and pass in the port number or the `NetWorker` reference to disconnect it.

That's it!

Step by step example

Importing the Package

To import our package into unity just do one of the following:

1. Drag it from your file system into the unity window and drop it onto the "Assets" folder
2. With Unity open, double click on the package and then click on import when prompted in Unity
3. Locate your purchase in the Unity Asset Store and click the "Import" link. If you have not yet downloaded it, you will need to click the "Download" link before the "Import" link becomes available

Create a Scene

To create a new scene, just go to File->New Scene or use the appropriate hot-keys. Once this is complete you should save your scene to the file system. Click File->Save Scene and when prompted, save it to your Unity project named "Game".

Making a Networked Cube

In your new scene click on GameObject->3D Object->Cube. You will see a new cube in your scene. Select the cube and then set its position to (0, 0, 0) so that it is visible by the camera. Now click on the "Add Component" button in the "Inspector" of Unity while the cube is still selected. Type in "NetworkedMonoBehavior" and you will see a C# script that is named that. Select that script to add it to the object.

Making the Cube Move Across the Network

To see that the cube is being networked, we will add a script to it that makes it move. Right click on the "Assets" folder and then go to Create->C# Script. Name this script "DemoMove" and then open the file to be edited. Make the script look like the following and then save it:

Forge-DemoMove.cs

```
using UnityEngine;

public class DemoMove : MonoBehaviour
{
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.UpArrow))
            transform.position += Vector3.up;

        if (Input.GetKeyDown(KeyCode.DownArrow))
            transform.position += Vector3.down;

        if (Input.GetKeyDown(KeyCode.RightArrow))
            transform.Rotate(Vector3.up, 45.0f);

        if (Input.GetKeyDown(KeyCode.LeftArrow))
            transform.Rotate(Vector3.right, 45.0f);
    }
}
```

Preparing Your Build

Now that you have altered the scene, make sure to save it with File->Save Scene. Next, you need to open up the Build Settings by clicking File->Build Settings

First, we are going to make sure that we have "Run In Background" turned on for our build so that the network communications continue while the game is not focused on our desktop. Select the "Player Settings" button located in the lower left corner of the build settings window. Open the settings section labeled "Resolution and Presentation" in the inspector window. Now just click the checkbox labeled "Run In Background*" so that it is turned on.

Next, while the Build Settings window is still open, navigate to the Bearded Man Studios, Inc.->Networking->Scenes folder and drag the "ForgeQuickStart" scene into the build settings area labeled "Scenes in Build". Next, navigate to your "Game" scene and drag it into the same area but below the "ForgeQuickStart" scene.

Under the "Platform" area of the Build Settings, click on "PC, Mac & Linux Standalone" then click the "Switch Platform" button on the lower left.

Running the Build

Now click "Build and Run" and then save the game onto your hard drive. Once the game opens (accept any firewall permissions you are prompted with), click on "Start Server", then go to the Unity Editor and open the "ForgeQuickStart" by double clicking it. Press play in the editor and then click "Start Client". Now you can click on your running PC build and press any of the arrow keys.

Congratulations

You now have glorious networking!

Simple Networked Mono Behavior

Simple Networked Monobehavior is the bread and butter of Forge Networking, it will be your most used class in building the networking infrastructure of your game/app. You would want to always use the Simple Networked Monobehavior when you are sending data back and forth that doesn't necessarily need to be realtime updates such as the NetworkedMonobehavior. For basic [Remote Procedure Calls](#) calls and small things, I highly recommend using this class over the NetworkedMonobehavior as it is less clutter and can solve 80% of what you need to do.

Is Owner

The purpose of Is Owner is for you to determine if you are the owner of that object over the network. It is a public Boolean so that means that the NetworkedMonoBehavior will also be able to use it as well as any classes you make to check upon it.

As an example below, say we want to make a door that only opens for the owner of that door. Below we do just that by checking the IsOwner boolean.

IsOwner Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedDoor : MonoBehaviour
{
    private void OnCollisionEnter(Collision col)
    {
        if (col.gameObject != null)
        {
            SimpleNetworkedMonoBehavior snmb =
col.gameObject.GetComponent<SimpleNetworkedMonoBehavior>();
            if (snmb != null)
            {
                if (snmb.IsOwner)
                {
                    //TODO: Open the door for I am the owner of it!
                }
                else
                {
                    //TODO: Display message to the user that he doesn't have permission to
enter!
                }
            }
        }
    }
}
```

BRPC (Remote Procedure Calls)

The BRPC attribute is for you to label a method/function to be used over the network. To do so is very easy and painless, you simply just have to put the tag of [BRPC] on the top of a method/function and then it is ready to be called across the network.

BRPC Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class SimpleRPCTest : SimpleNetworkedMonobehavior
{
    public int Health = 0;

    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            RPC("IncreaseHealth", NetworkReceivers.All, 1);
        }
    }

    [BRPC]
    public void IncreaseHealth(int amount)
    {
        //This will increase the health across all clients locally.
        //If we want this health to be always synced across for new players
        joining in thereafter then we need to call this as a BufferedRPC
        // Example: RPC("IncreaseHealth", NetworkReceivers.AllBuffered, 1); //I
        do not recommend doing this and would prefer using the [NetSync] attribute
        from NetworkedMonobehavior
        Health += amount;
    }
}
```

Owner Id

Owner ID is a way for you to tell which networked object is owned by which player. You will always have a list of players available on the server to do the comparison checks upon, as well as being able to request the players whenever needed. You would use the OwnerID to get further information about objects that 'IsOwner' is not able to provide. For Example: Being able to tell which players are on Team 0, and which are on Team 1.

Owner ID Example

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BeardedManStudios.Network;

public class BasketballPlayer : SimpleNetworkedMonoBehavior
{
    public int TeamID = -1;
    public static System.Collections.Generic.Dictionary<ulong, int>
    GamePlayers = new System.Collections.Generic.Dictionary<ulong,int>();
    protected override void NetworkStart()
    {
        base.NetworkStart();
        //Assign a random team id!
        if (IsOwner)
            RPC("SetPlayerTeam", NetworkReceivers.AllBuffered,
Random.Range(0, 1));
    }
    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            int nextTeamID = 0;
            //Change the team!
            if (TeamID == 0)
                nextTeamID = 1;

            //I don't recommend to calling this every time as it would be costly on
the network as the game would progress further. If you plan on calling an
BufferedRPC a lot
            //Please consider using a NetworkedMonobehavior instead and putting the
[NetSync] attribute above it.
            RPC("SetPlayerTeam", NetworkReceivers.AllBuffered, nextTeamID);

        }
    }
    [BRPC]
    public void SetPlayerTeam(int teamID)
    {
        TeamID = teamID;
        ulong myPlayerID = Networking.PrimarySocket.Me.NetworkId;
        if (GamePlayers.ContainsKey(myPlayerID))
            GamePlayers[myPlayerID] = TeamID;
        else
            GamePlayers.Add(myPlayerID, TeamID);
    }
}
```


IsServerOwner

The IsServerOwner boolean is another way to check if the networked object is owned by the server. Instead of doing just a simple 'IsOwner' check and getting clarification if that is owned by that player or not, we can get further information to tell if it is owned specifically by the Server as well.

IsServerOwner Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class ServerOwnedObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();
        if (IsServerOwner)
        {
            Debug.Log("This object is owned by the server! Not the
clients!");
        }
    }
}
```

Current RPC Sender

CurrentRPCSender is a way for you to determine whom called the RPC to your method/function. This will be the player that sent the RPC to you.

CurrentRPCSender Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class ChatMessageMonitor : SimpleNetworkedMonoBehavior
{
    string LastMessageReceived = "";
    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            RPC("ReceiveMessage", NetworkReceivers.All, "Hello World!");
        }
    }
    [BRPC]
    public void ReceiveMessage(string msg)
    {
        LastMessageReceived = msg;
        ulong senderID = CurrentRPCSender.NetworkId;
        string ip = CurrentRPCSender.Ip;
        Debug.Log("Received message from " + senderID + " on " + ip + ": "
+ msg);
    }
}
```

Is Setup

This is a way for you to determine whether an object is setup on the network to be used. What that means is if the object is not setup, it simply cannot be used to send RPC's commands and other network related things until it has been properly setup.

IsSetup Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class BasicSetup : SimpleNetworkedMonoBehavior
{
    private void Awake()
    {
        Debug.Log("[Awake] Am I setup? " + IsSetup); //This will always
return false
    }
    protected override void NetworkStart()
    {
        base.NetworkStart();
        //I am ready to do networking!!!
        Debug.Log("[Network Start] Am I setup? " + IsSetup);
    }
}
```

Networked Id

This is the unique ID of the SimpleNetworkedMonobehavior and derivatives of it.

Note: This ID is not to be confused with NetworkingPlayer.NetworkId, this ID is a unique ID for every SimpleNetworkedMonobehavior even if owned by the same player.

When a SimpleNetworkedMonobehavior is successfully instantiated or connected to the network, it will assign this unique ID to be used and it is required for calling any RPC's or Syncing of variables.

You can use this ID to determine unique objects in your game as well.

NetworkedID Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    public bool IsHeroObject = false;
    protected override void NetworkStart()
    {
        base.NetworkStart();
        if (IsOwner && !Networking.PrimarySocket.IsServer)
        {
            //If we are the owner of the object and we are not the server,
            // then let the server know we are a Hero object!
            RPC("SetAsHeroObject", NetworkReceivers.AllBuffered);
        }
    }
    [BRPC]
    public void SetAsHeroObject()
    {
        IsHeroObject = true;
    }
}
```

In the example above, I set all clients as hero objects and ignore the server doing the same.

Owning Net Worker

The `OwningNetworking` is the socket connection that this object is using. As you can have multiple socket connections in your game/application, you are free to check which socket connection a particular object is using by simply checking the `OwningNetworker`.

From there you can do all sorts of cool things in finding out more information about the current server.

As an example below I check to see if I am connected to a primary server, I then use the internal `NetworkingManager` to poll the player list as (being a client, I don't know how many players there are in the server until I request it).

OwningNetworker Example

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    public bool IsSocketConnectionPrimaryServer
    {
        get
        {
            if (!NetworkingManager.IsOnline) //Offline Play
                return true;
            if (OwningNetWorker != null)
                return OwningNetWorker.Host == "127.0.0.1"; //Set to the IP
of the primary server
            return false;
        }
    }
    protected override void NetworkStart()
    {
        base.NetworkStart();
        if (OwningNetWorker.IsServer)
            return;
        Debug.Log("Are we connected to the primary server? " +
IsSocketConnectionPrimaryServer);
        NetworkingManager.Instance.PollPlayerList((players) =>
        {
            Debug.Log("How many players? " + players.Count);
        });
    }
}
```

Owning Player

OwningPlayer is the way to get the player who owns that particular object.

OwningPlayer Example

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();
        Debug.Log("My Network ID is: " + OwningPlayer.NetworkId);
        if (OwningPlayer.NetworkId == 0) //I am the server!
            Debug.Log("The server owns this object!");
        Debug.Log("My ip is: " + OwningPlayer.Ip);
        if (OwningNetWorker.IsServer)
            NetworkingManager.Instance.SetName("Billy");
        else
            NetworkingManager.Instance.SetName("Client#" +
OwningPlayer.NetworkId);
        NetworkingManager.Instance.PollPlayerList((players) =>
        {
            foreach (NetworkingPlayer p in players)
            {
                Debug.Log("Connected player: " + p.Name);
            }
        });

        //AuthoritativeRPC("PrivateMessage", OwningNetWorker, OwningPlayer, true,
"Hello World! - PM");
    }

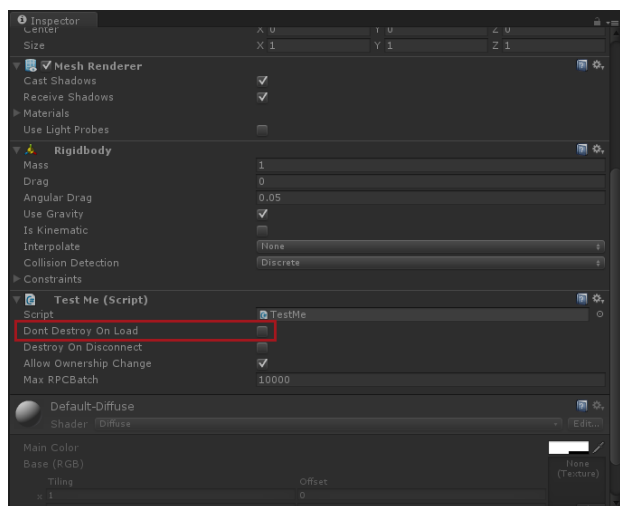
    [BRPC]
    private void PrivateMessage(string msg)
    {
        Debug.Log("Received a private message: " + msg + "\nFrom: " +
CurrentRPCSender.Name);
    }
}
```

Using the example above, I get the information about an object as it is created on the network. I then set all connected clients names to "Client#" + Player.NetworkId. Only setting the servers name to "Billy". When I poll the users from the server, I then get a response with all the connected players names and who is connected.

You can use the OwningPlayer to also send a direct RPC over to them by doing a AuthoritativeRPC. In the example above if you uncomment out AuthoritativeRPC it would then just send a direct message to itself of "Hello World! - PM". You can easily use this to send private RPC/Messages from one client to the other.

Don't Destroy On Load

Don't Destroy On Load is a toggle that is able to be set on the a SimpleNetworkedMonobehavior derivative controls from the inspector window.



Marking an object as 'Don't Destroy On Load' will make it so that whenever you transition to another scene, it's networked variables will remain intact moving forward.

Locate

Locate allows you to be able to grab any networked object in the game by passing in the NetworkedID (Objects, not the same as Player.NetworkID).

Locate Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        SimpleNetworkedMonoBehavior myself = Locate(NetworkedId);
        if (myself != null)
            Debug.Log("Yippie!");
    }
}
```

Using the example above, I grab my own SimpleNetworkedMonoBehavior by passing in my own NetworkedID

Network Destroy

NetworkDestroy is similar to Networking.Destroy, although NetworkDestroy is an internal way to destroy a networked object. You can also call this outside in a MonoBehaviour as it is a static method.

NetworkDestroy Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        if (Networking.PrimarySocket.IsServer)
            return;

        //Kill this object, KILL IT WITH FIRE!
        bool success = NetworkDestroy(NetworkedId);

        //Alternative ways to destroy an object as well
        //SimpleNetworkedMonoBehavior.NetworkDestroy(NetworkedId);
        //NetworkingManager.NetworkDestroy(NetworkedId);

        Debug.Log("Did I succeed in destroying my networked object? " +
            success);
    }
}
```

This will handle all the cleanup on the server so that all clients would destroy that object.

Get RPC

After using Reflect, GetRPC takes in an ID parameter of the function/method that needs to be used. This is all used internally by ForgeNetworking.

Queue RPC For Instantiate

This is an internal call for ForgeNetworking to hold onto all data until the object has been created successfully on the network and is hooked up properly. It will then execute all QueuedRPCs on that particular object when it has done so. You will never call this method unless you are wanting to change the way RPC's are being queued internally for when they are not instantiated on the networking just yet.

Cleanup

This is an internal call by ForgeNetworking to remove our hooks into the MainThreadManager for when an object is about to be destroyed.

Reflect

Reflect is used internally by ForgeNetworking to gather all the functions with a [BRPC] attribute on them to be used across the network.

Owner Update, Non-owner Update, Owner Fixed Update, Non-owner Fixed Update

These are helper functions that get called for you so that you don't have to stick everything in just a regular 'Update' unity function and do if checks on 'IsOwner' boolean.

Below I give a list of comments that showcase exactly what each update function can do, this will help you in the long run when you need things to only be run on one client and not the other. Such as Input!

Examples

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    public void Update()
    {
        //TODO: I am just a regular update that runs the same on all
clients!
    }
    protected override void OwnerUpdate()
    {
        base.OwnerUpdate();
        //TODO: I am like a unity update! Except I only run on the OWNER of
this object!
    }
    protected override void NonOwnerUpdate()
    {
        base.NonOwnerUpdate();
        //TODO: I am like a unity update! Except I only run on the
NON-OWNER of this object!
    }
    protected override void OwnerFixedUpdate()
    {
        base.OwnerFixedUpdate();
        //TODO: I am like a unity fixed update! Except I only run on the
OWNER of this object!
    }
    protected override void NonOwnerFixedUpdate()
    {
        base.NonOwnerFixedUpdate();
        //TODO: I am like a unity fixed update! Except I only run on the
NON-OWNER of this object!
    }
}
```

InvokeRPC

InvokeRPC is primarily used internally for ForgeNetworking, it validates the RPC being called to make sure that the RPC exists and the parameters given are compatible. If the RPC is validated the method returns true, then the RPC is added to the stack which executes the RPC on the next frame. The NetworkingStreamRPC is the stream created that can be passed over the network to represent the request, parameters and all other meta information needed by the network.

AuthoritativeRPC, AuthoritativeURPC, RPC, URPC

AuthoritativeRPC is a way for you to call a specific RPC for a particular user/player. It allows for you to communicate specifically to a given player.

AuthoritativeURPC is just an unreliable rpc to that particular user/player, does the same as AuthoritativeRPC but it might not make it to the other end. (Useful for FPS type of games where data is constantly sent and lost).

RPC is for the networked object to call a particular function on the other end.

URPC is the same as RPC but it is unreliable and it's data can be lost on the way to the other client(s) (usefull for FPS type of games).

In the example below I use all the RPC calls in order to show results of each of the different types.

RPC Examples

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();
        if (OwningPlayer.NetworkId == 1)
        {
            if (Networking.PrimarySocket.IsServer)
            {
                //I am the first player to join and I(the server calling
                this function) want to send a message to this client!
                AuthoritativeRPC("PrivateMessage", OwningNetWorker,
                OwningPlayer, true, "Hello there player1");
                AuthoritativeURPC("UnreliablePrivateMessage",
                OwningNetWorker, OwningPlayer, true, "Did you receive this?!");
            }
            RPC("ServerMessage", NetworkReceivers.Server, "Hello there!");
            URPC("UnreliableServerMessage", NetworkReceivers.Server, "Did you
receive this?!");
        }
        [BRPC]
        private void PrivateMessage(string MessageFromServer)
        {
            Debug.Log("(RPC)Server messaged me: " + MessageFromServer);
        }
        [BRPC]
        private void UnreliablePrivateMessage(string MessageFromServer)
        {
            Debug.Log("(URPC)Server messaged me: " + MessageFromServer);
        }
        [BRPC]
        private void ServerMessage(string MessageFromClient)
        {
            Debug.Log("(RPC)Client Messaged me: " + MessageFromClient + " -
from player " + CurrentRPCSender.NetworkId);
        }
        [BRPC]
        private void UnreliableServerMessage(string MessageFromClient)
        {
            Debug.Log("(URPC)Client Messaged me: " + MessageFromClient + " -
from player " + CurrentRPCSender.NetworkId);
        }
    }
}
```

Serialize and Deserialize

These are both used in NetworkedMonoBehavior and not used in SNMB. It is meant as way to serialize a class across the network and deserialize it on the other end.

[Serialized](#) - Click here for the NetworkedMonoBehavior documentation of Serialize

[Deserialize](#) - Click here for the NetworkedMonoBehavior documentation of Deserialize

NetworkedBehaviors

This is a public static dictionary that you can use to reference all the networked objects in your game/project. This list is based on the NetworkedId of those objects which can be used to validate or grab key information from any object you may have. This list is quite the best thing to reference whenever you have all the information in place and want to act upon it.

Below is an example of how I could validate a powerup by checking if the object exists and if it is owned by the server.

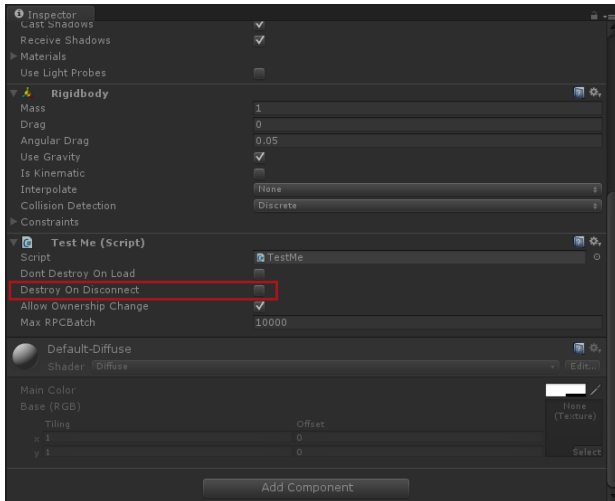
NetworkedBehaviors Example Usage

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BeardedManStudios.Network;

public class PickupPowerManager : MonoBehaviour
{
    public bool ValidatePickup(ulong PowerID)
    {
        SimpleNetworkedMonoBehavior behavior = null;
        if
        (SimpleNetworkedMonoBehavior.networkedBehaviors.ContainsKey(PowerID))
            behavior =
            SimpleNetworkedMonoBehavior.networkedBehaviors[PowerID];
        if (behavior == null || !behavior.IsServerOwner)
            return false;
        return true;
    }
}
```

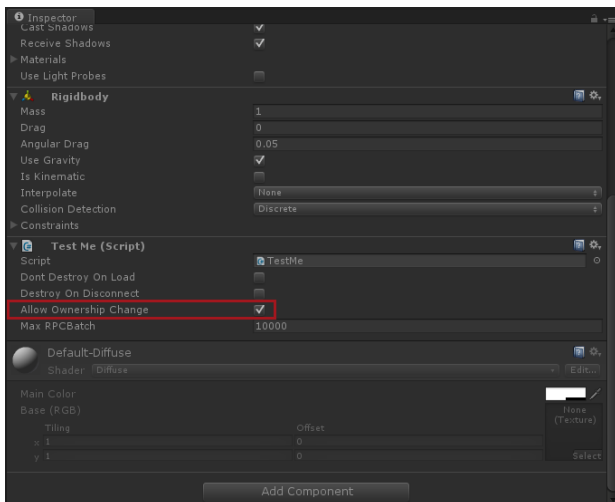
Destroy On Disconnect

The Destroy On Disconnect means that whenever you lose connection to the host it would destroy that object from the host and client. This is accessible from the inspector window of any object that is a derivative of SimpleNetworkedMonoBehavior.



Allow Ownership Change, Change Owner

The allow ownership change means that object will allow permission to change ownership from one object to the next. By enabling the toggle from the inspector window of a SimpleNetworkedMonobehavior means that the object can change ownership.



After doing so you are then able to call 'ChangeOwner' on any class that derives from SimpleNetworkedMonobehavior with the AllowOwnershipChange toggled on/true. You then would pass in the new ID of the player so that he may become the owner.

ChangeOwner Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        if (!Networking.PrimarySocket.IsServer)
            ChangeOwner(0); //Change my ownership to Player 0 (which is the
server)
    }
}
```

Only the Server or OwningPlayer can change the ownership of it's own object.

GenerateUniqueID

This function is internally used by ForgeNetworking to give a unique ID to SimpleNetworkedMonoBehavior and it's derivatives a unique ID when successfully instantiated on the network.

Reset For Scene

This is called automatically when you are doing a scene transition from on to the next using the Networking.ChangeClientScene. You will never call ResetForScene as it is internally called by ForgeNetworking for the example below. Reset for scene also internally only skips objects that are marked with 'dontDestroyOnLoad' = true and any other ones marked to skip.

ChangeClientScene Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        string nextLevel = "YOUR_LEVEL_GOES_HERE";
        if (Networking.PrimarySocket.IsServer)
        {
            Application.LoadLevel(nextLevel);
            Networking.ChangeClientScene(OwningNetWorker, nextLevel);
        }
    }
}
```

With the example above, you can also target a specific port/socket/player as well.

Reset All

Reset all is similar to ResetForScene, but it will destroy everything even if it is marked as 'dontDestroyOnLoad' = true. This is an internal call used for when you are resetting the networking by calling Networking.NetworkingReset().

NetworkingReset Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        if (OwningNetWorker.IsServer)
            return;

        //Kill connect
        Networking.NetworkingReset();
    }
}
```

The example above will make it so that any client who connects will be disconnected automatically. This can be useful if you want to make yourself a whitelist or limit the amount of connections at runtime for casual users vs VIP.

If you want to Ban a specific player, you can do so by calling BanPlayer on the Socket (NetWorker), passing in the playerId and how long they should be banned for in minutes. Feel free to change this code to do more than just that as well.

ResetBufferClear

ResetBufferClear is an internal call for ForgeNetworking when an object has been destroyed and re-initialized later. When an object gets destroyed, its RPCs and calls to it are then discarded for any of the buffer it had, if it is then created again this function will call internally so that it may receive its buffer again.

Initialize

This is an internal call in the ForgeNetworking, this will be called automatically when an object has been created on the network.

Delay Initialize

This is an internal call in ForgeNetworking that will setup the object once the socket connection has been successfully made to the server.

NetworkStart

This is an overridable function to which you are able to send all your networking related calls after on. Once this is called you are connected to the network successfully and can from then on send RPC calls and all networking dealing with the connected socket.

NetworkStart Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : SimpleNetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart(); //ALWAYS CALL BASE BEFORE DOING ANY NETWORKING
        RELATED TASKS
        //TODO: Send all RPC's here since we know we are connected
        successfully.
        RPC("ServerMessage", NetworkReceivers.All, "I am connected to the
server!");
    }
    [BRPC]
    public void ServerMessage(string messageReceived)
    {
        Debug.Log("Recieved message: " + messageReceived + " - from player:
" + CurrentRPCSender.NetworkId);
    }
}
```

Setup Objects

This is an internal call for ForgeNetworking to setup all networked objects on the client/server. It takes in a list of networked objects and sets each one up in their correct order.

Setup

This is an internal call for ForgeNetworking to setup this particular networked object with a provided socket connection. With the socket connection provided, the object can then receive network related params.

Disconnect

Disconnect is a function to which you would override if you want to do any additional logic when an object is disconnected. This is only called when the user does an Application.Quit() or scene resets (transition to another scene). This can be useful if you want to make sure no additional logic is being run on objects as you are transitioning from one scene to the next for static objects.

Disconnect Override Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayer : SimpleNetworkedMonoBehavior
{
    bool isAvailable = false;

    protected override void NetworkStart()
    {
        base.NetworkStart();

        isAvailable = true;
        //I am available and good to go!
    }

    public override void Disconnect()
    {
        base.Disconnect();

        isAvailable = false;
        //I am no longer available as this object just disconnected from
        it's connection.
    }
}
```

OnDestroy

OnDestroy must be overridden as a function whenever you are deriving from [Simple Networked Mono Behavior](#) or [Networked Monobehavior](#). Calling the base first, then executing your code logic below.

Please follow the below example as a ways of following the standards of networked programming with ForgeNetworking.

OnDestroy Override Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayer : SimpleNetworkedMonoBehavior
{
    string ServerIP = "1234567";

    //WE MUST ALWAYS OVERRIDE IF WE USE OnDestroy
    protected override void OnDestroy()
    {
        //MUST CALL BASE FIRST! THEN ADDITIONAL LOGIC BELOW
        base.OnDestroy();

        //My object has been destroyed, cleanup any additional logic
        ServerIP = string.Empty;
    }
}
```

OnApplicationQuit

OnApplicationQuit is similar to [OnDestroy](#) in that it must be overridden whenever you want to use it. Below is a proper example of how to override OnApplicationQuit.

OnApplicationQuit Override Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayer : SimpleNetworkedMonoBehavior
{
    public System.Action FinalCleanup = null; //A public action/event to
    hook into from outside this class when we quit

    //MUST CALL OVERRIDE IF WE WANT TO USE OnApplicationQuit
    protected override void OnApplicationQuit()
    {
        //MUST ALWAYS CALL BASE FIRST BEFORE YOUR LOGIC!
        base.OnApplicationQuit(); //This will behave differently depending
        if it is a SimpleNetworkedMonobehavior or NetworkedMonoBehavior.

        //The program has quit, lets do any additional logic here to make
        sure we do a clean exit
        if (FinalCleanup != null)
            FinalCleanup();
    }
}
```

NetworkDisconnect

NetworkDisconnect is similar to [OnApplicationQuit](#) and [OnDestroy](#) in that we must override the function if we want to do logic following it's call. NetworkDisconnect will always be called whenever the client forcible gets kicked out; either by himself or the server (per object basis).

Below is an example of how to override NetworkDisconnect.

NetworkDisconnect Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayer : SimpleNetworkedMonoBehavior
{
    public System.Action KickedCleanup = null; //Action to hook into for
    when we kick ourselves or the server does

    //MUST ALWAYS CALL OVERRIDE ON THIS IF WE WANT TO USE NetworkDisconnect
    protected override void NetworkDisconnect()
    {
        //MUST ALWAYS CALL THIS FIRST BEFORE DOING YOUR LOGIC
        base.NetworkDisconnect(); //This will disconnect the socket and any
        information from this object

        //We either kicked ourselves out of the connection or the server
        did
        if (KickedCleanup != null)
            KickedCleanup();
    }
}
```

Networked Monobehavior

This is the meat and bones of Forge Networking. We highly recommend for proper efficiency to make sure to utilize Networked Monobehavior on objects that need their values synchronized in realtime and to always use SimpleNetworkedMonobehavior if you are doing trivial tasks such as only sending [Remote Procedure Calls](#) (i.e. chat).

You would always want to use Networked Monobehavior when you want to handle the interpolation of an object and synchronizing the position, scale, and rotation.

Deserialize

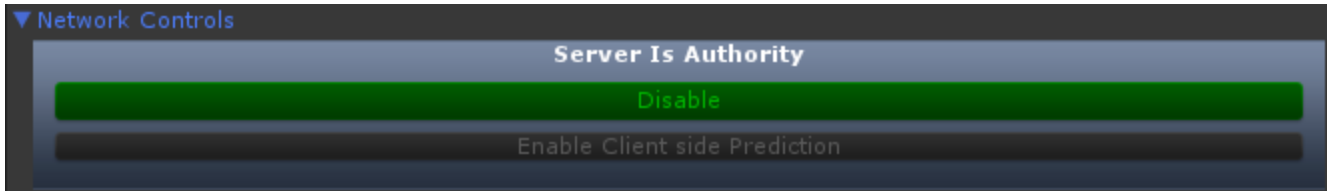
This can only be called by the owner of the object and dependent on if it is marked as 'Is Server Authority enabled'. This process is called internally by Forge Networking for deserializing a networking stream, it must be deserialized in the proper order for it to properly grab the variables on both ends. It will then map the NetSync variables accordingly as well as Lerp those values if marked to.

Input Requests

The best way for sending authoritative input to the server is by doing Input Requests on Server Authoritative networked objects. To do so we need to make sure the following things are set.

We need to make sure that under the 'Network Controls->Server is Authority' that it has been Enabled under our Networked Monobehavior derivative object. Without this, the objects are just controlled by the players that created them and thus there is no server authority to control requests.

The best use of this is for games that need validation from the server before interpreting on it's own client.



Now that has been enabled, it is time to show some example code that we can send to the server so that it can move the object on it's end because we no longer control it (whoever created it owns it, but now that it is set to authoritative, the server will interpret every call and correct it on the client).

In the example code below, I show how you can utilize this for your game/app by listening to the input requests only on the server, and allowing the clients to send their own inputchecks.

Note: I commented out mouse requests as I wasn't using it in my code but rather wanted to show what it would look like for setting it up.

Input Requests Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : NetworkedMonoBehavior
{
    [NetSync]
    public int MyValue = 0;
    protected override void NetworkStart()
    {
        base.NetworkStart();
        //If we are not the server, ignore input requests as only the
server can listen
        //Socket == null would be a safecheck for Offline play
        if (Networking.PrimarySocket == null ||
!Networking.PrimarySocket.IsServer)
            return;
        inputRequest += ServerReceivedInputHeld;
        inputUpRequest += ServerReceivedInputUpRequest;
        inputDownRequest += ServerReceivedInputDownRequest;
        //Mouse requests
        //mouseRequest += ServerRecievedMouseHeldRequest;
        //mouseUpRequest += ServerRecievedMouseUpRequest;
        //mouseDownRequest += ServerRecievedMouseDownRequest;
    }
    #region Keyboard Input
    private void ServerReceivedInputUpRequest(KeyCode keyCode, int frame)
    {
    }
    private void ServerReceivedInputDownRequest(KeyCode keyCode, int frame)
    {
        switch (keyCode)
        {
            case KeyCode.UpArrow:
                MyValue++;
                break;
            case KeyCode.DownArrow:
```

```

        MyValue--;
        break;
    }
}
private void ServerReceivedInputHeld(KeyCode keyCode)
{
    switch (keyCode)
    {
        case KeyCode.RightArrow:
            transform.position += Vector3.right * 5.0f *
Time.deltaTime;
            break;
        case KeyCode.LeftArrow:
            transform.position += Vector3.left * 5.0f * Time.deltaTime;
            break;
    }
}
#endregion
#region Mouse Input
//private void ServerRecievedMouseUpRequest(int id, int frame)
//{
//}
//private void ServerRecievedMouseDownRequest(int id, int frame)
//{
//}
//private void ServerRecievedMouseHeldRequest(int id)
//{
//}
#endregion
private void OnGUI()
{
    if (!IsOwner)
        return;
    GUILayout.Label("My value: " + MyValue);
}
protected override void OnDestroy()
{
    //DESTROY HOOKED EVENTS BEFORE BASE
    inputRequest -= ServerReceivedInputHeld;
    inputUpRequest -= ServerReceivedInputUpRequest;
    inputDownRequest -= ServerReceivedInputDownRequest;
    base.OnDestroy();
}
protected override void OwnerUpdate()
{
    //We own this object, so we can send input requests here!
    InputCheck(KeyCode.RightArrow);
    InputCheck(KeyCode.LeftArrow);
    InputCheck(KeyCode.UpArrow);
    InputCheck(KeyCode.DownArrow);
    //If we want to do the same authoritative input for mouse
    //MouseCheck(0);
}

```

```

        //MouseCheck(1);
    }
}

```

Manual NetSync

This will allow you to manually sync variables across the network whenever you want to send them. This is ideal for situations where you would want an event to happen in your game but only have the server send it over when it has been told to do so.

In the example below, I made it so hitting Right/Left arrow would increment/decrease my Mana variable, and Spacebar would Sync that mana across the network to everyone.

Manual Netsync Example

```

using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : NetworkedMonoBehavior
{
    [ManualNetSync]
    public int Mana = 100;

    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            Mana++;
        }
        else if (Input.GetKeyDown(KeyCode.LeftArrow))
        {
            Mana--;
        }
        else if (Input.GetKeyDown(KeyCode.Space))
        {
            bool reliable = true;
            NetworkReceivers recievers = NetworkReceivers.All;
            SerializeManualProperties(reliable, recievers);
        }
    }
}

```

NetSyncToServer

This is a way to synchronize variables only to the server. It is essentially doing the same as setting the receivers to Server from a NetSync, but just a quick attribute to put on top of your variable so you don't have to describe the receivers.

NetSyncToServer Example

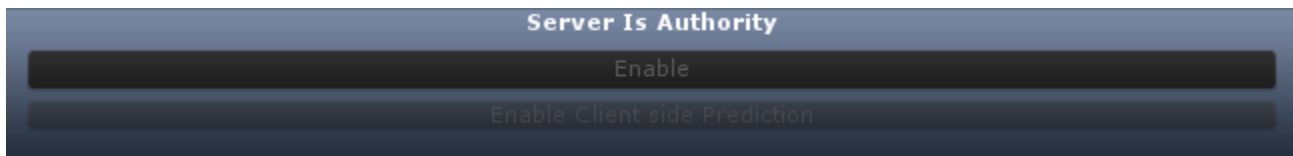
```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : NetworkedMonoBehavior
{
    [NetSyncToServer]
    public int Mana = 100; //ONLY THE CLIENT AND SERVER WILL HAVE THE
    CORRECT VALUES, JOINING CLIENTS WILL NOT
    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            Mana++;
        }
        else if (Input.GetKeyDown(KeyCode.LeftArrow))
        {
            Mana--;
        }
    }
}
```

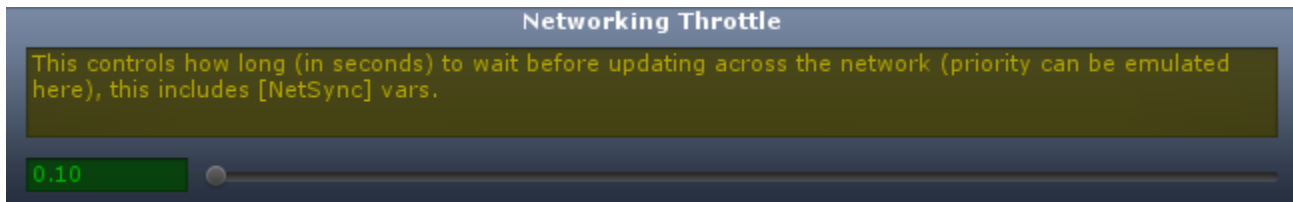
Network Controls

Server Is Authority means that the object will be sending it's input requests to the server and for the server to authorize them.

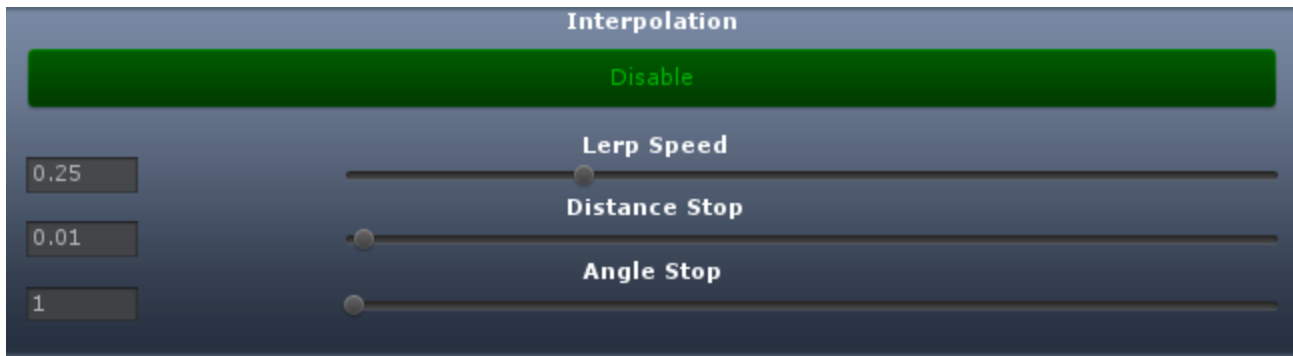
Please see [Input Requests](#) for how you can utilize this feature. Turning on client side prediction will allow you to manually object the object and for it to still process it's information on the server but if the server notices a difference it will automatically correct it on the client.



The Networking Throttle is a way for you to determine how often you would like this particular networked object to be updated on the network. For efficiency on slow paced games, it is highly recommend that you keep this value higher than 0.10, but for fast paced games, it is recommend to keep it low but also minimize how many networked objects your game will have. This will be Key to making your game stable and unstable.



Ever get jagged movement from the client but not on the server? That is because Interpolation needs to be turned on and modified so that it can fine tune for your clients connected. This will make sure the movements are smooth. (Default settings are recommended, but feel free to experiment on your own with these values depending on your game/app).



The Easy Controls tab is for you to determine what x, y, z variables will be synced across the network as well as which values will have interpolation affect them. Highly recommend to leave this disabled and not used if you are not concerned about syncing the positions/rotations/scale of a particular object. Best usage is to put this enabled as for Position/Rotation, but leaving scale to be determined on the server. Also make sure to not have multiple objects in the scene with these values always enabled just means more data is being synced across the network for no reason, so only enable these on networked objects that need their movements/rotations synced.



This Miscellaneous (misspelled in this photo) tab is for you to get some extra controls over your networked object. From here you can determine if the sending of the data is reliable at all times, if it is a player object so that that NetworkingPlayer.Player will be set properly, Destroy On Disconnect for objects that need to be cleaned up whenever a disconnect happens, and Teleport To Initial Position will instantly set the object to the appropriate position when it has been instantiated over the network.



Serialized

Is a way for you to grab the latest NetworkedMonobehavior derivative serialization (including all netsync variables) into a BMSByte. This is automatically done internally by Forge Networking as it is sending data back and forth across the network but if you want to see how we are serializing it you can take a look at the code in this class.

Master Class Beginner

Initializing the Firewall Request

Forge-InitializingFirewallRequest.cs

```
using UnityEngine;
using BeardedManStudios.Network;

public class StartGame : MonoBehaviour
{
    public void Start()
    {
        ushort portNumber = 15937;
        Networking.InitializeFirewallCheck(portNumber);
    }
}
```

So usually when we make a game we want the Windows firewall to pop up at the start, not during, simply because it would make the player have to leave the game in order to accept their firewall settings when they try to join a server.

In order to achieve this we can use the `Networking.InitializeFirewallCheck(<PORT>);` method. This method can be placed anywhere at any time. It returns void and its entire purpose is to be a quick helper on kicking off the Windows Firewall.

The way that this method works behind the scenes is that it will attempt to create a TCP connection which will trigger Windows to open up the firewall window. This connection is immediately closed once it has successfully bound so it doesn't stick around during your game.

HINT: To prevent this firewall check from happening all of the time you could use `PlayerPrefs` in order to save that you have already done the check. If you were to call this method every time you start the game, it will still run normally and NOT open the Firewall every time you start the game. This is due to how Windows operates, if it has already been authorized by the player, then it will not request authorization again.

Youtube video for demonstration - <https://youtu.be/eihUZb8axPk>

Remote Procedure Calls

Forge-RemoteProcedureCall.cs

```
using UnityEngine;

using BeardedManStudios.Network;

public class MoveBallUp : SimpleNetworkedMonoBehavior
{
    [BRPC]
    private void MoveBall(Vector3 moveBy)
    {
        transform.position += moveBy;
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
            RPC("MoveBall", new Vector3(0.35f, 1.2f, -2.32f));
    }
}
```

Remote Procedure Calls (RPCs) are a way of calling a method from within a script across the network. In our system you can mark a function in a

class that derives from `SimpleNetworkedMonoBehavior` or any of its derivatives (such as `NetworkedMonoBehavior`) with the attribute "[B RPC]" (as seen in the script above on line 7). This will allow this particular method to be called from across the network via its string name.

When you call an RPC you can do so by calling the method `RPC<T>("T")` that is a part of the class that has the RPC function in question (as seen on line 18). The first argument of this method is the string name of the method you wish to call, in the case of the example, this would be "MoveBall". The second argument of the RPC are the arguments that are to be passed into the method in the order that the function signature has them in.

Note: There is an overload to the RPC function in which case the second argument is a `NetworkReceivers` enum to stat who should be executing this method. If this is the second argument, then the consecutive arguments after this should be the arguments that are to be passed into the method in the order that the function signature has them in.

If you chose to use the `NetworkReceivers` argument, then you can select between options that are buffered and ones that are not buffered. The buffered arguments mean that every time someone new connects to the game, they will call this RPC method immediately. Of course you can set delays inside of the method for players who just connected to lower the load of what happens upon connection.

Youtube video for demonstration - https://youtu.be/P-tY_lwN9Z8

Control Access With IsOwner

[Owner Update](#), [Non-owner Update](#), [Owner Fixed Update](#), [Non-owner Fixed Update](#) has more examples on how this can work as well, so please check that out if the example below doesn't explain it well enough.

Forge-ControlAccessWithOwner.cs

```
using UnityEngine;

using BeardedManStudios.Network;

public class MoveCube : NetworkedMonoBehavior
{
    public float moveSpeed = 5.0f;

    protected override void OwnerUpdate()
    {
        base.OwnerUpdate();

        if (Input.GetKey(KeyCode.UpArrow))
            transform.position += Vector3.up * moveSpeed * Time.deltaTime;

        if (Input.GetKey(KeyCode.DownArrow))
            transform.position += Vector3.down * moveSpeed * Time.deltaTime;

        if (Input.GetKey(KeyCode.RightArrow))
            transform.Rotate(Vector3.up, 5.0f);

        if (Input.GetKey(KeyCode.LeftArrow))
            transform.Rotate(Vector3.right, 5.0f);
    }
}
```

Youtube video for demonstration - <https://youtu.be/LQNHf1NyWh8>

SimpleNetworkedMonoBehavior vs NetworkedMonoBehavior

SimpleNetworkedMonoBehavior is best utilized for doing [Remote Procedure Calls](#) and is best for doing basic network functionality for most optimization.

Networked MonoBehavior is best for doing interpolation, syncing position/rotation/scale, and for utilizing NetSync. It is always recommended to use SimpleNetworkedMonoBehavior whenever possible for the lowest overhead on networking, but for any objects that need to use the Networked MonoBehavior features to do so.

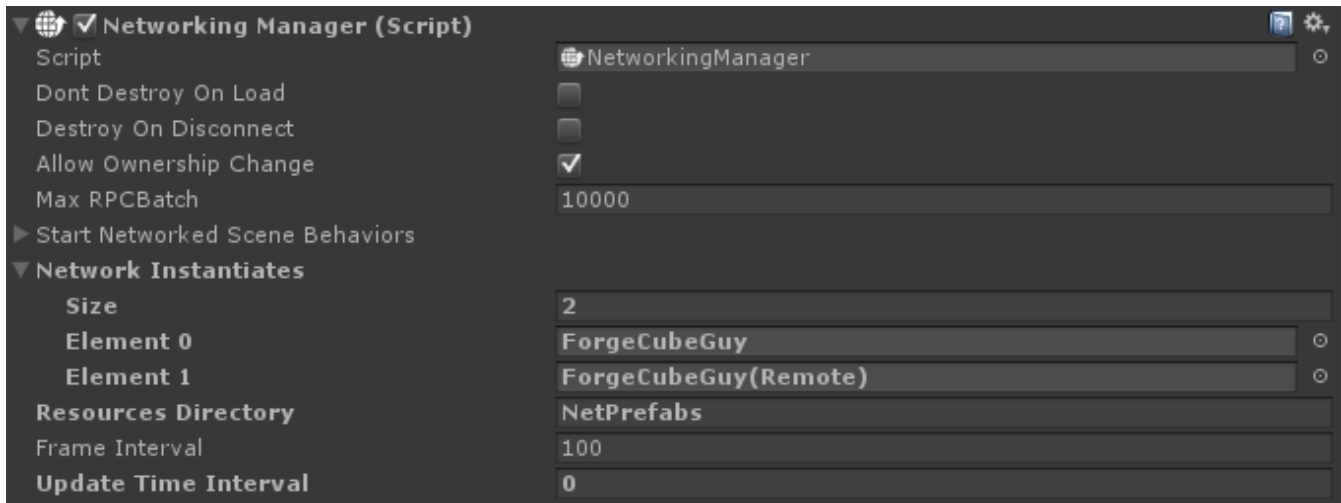
Youtube video for demonstration - https://youtu.be/U-_99-80usY

Network Instantiation

The basic call for instantiating an object over the network, similar to `GameObject.Instantiate`, but it isn't local, as it will instantiate the object over the network as well.

The proper way to instantiate an object is to make sure that you are properly connected to a socket as well as making sure that there is a Networking Manager in the scene that you are in.

On the Networking Manager, you want to make sure that you have your networked object you want to instantiate on the socket connection in the Network Instantiates array.



In the picture above, I have my 'ForgeCubeGuy' and 'ForgeCubeGuy(Remote)' specified as objects I want to instantiate across the network.

Now that is done, it just takes a bit of code to spawn them!

Network Instantiate Example

```
public class PlayerSpawnerExample : MonoBehaviour
{
    public GameObject objectToSpawn = null;
    private void Start()
    {
        if (objectToSpawn == null)
        {
            Debug.LogError("Cannot have a null object to spawn!");
            return;
        }
        if (NetworkingManager.Socket == null ||
NetworkingManager.Socket.Connected)
            Networking.Instantiate(objectToSpawn,
NetworkingReceivers.AllBuffered, PlayerSpawned);
        else
        {
            NetworkingManager.Instance.OwningNetWorker.connected +=
delegate()
            {
                Networking.Instantiate(objectToSpawn,
NetworkingReceivers.AllBuffered, PlayerSpawned);
            };
        }
    }
    private void PlayerSpawned(SimpleNetworkedMonoBehavior playerObject)
    {
        Debug.Log("The player object " + playerObject.name + " has spawned
at " +
            "X: " + playerObject.transform.position.x +
            "Y: " + playerObject.transform.position.y +
            "Z: " + playerObject.transform.position.z);
    }
}
```

In this code example, I make sure to only spawn the object over the network if it has successfully connected, otherwise I queue it up to be instantiated when it has successfully connected.

Youtube video for demonstration - <https://youtu.be/p9TLa8zY504>

Buffered Networking Instantiate

As shown in the example below, we simply wait until we are fully connected to the socket before trying to instantiate an object over the network.

Forge-BufferedNetworkInstantiation.cs

```
using UnityEngine;

using BeardedManStudios.Network;

public class MakePlayer : MonoBehaviour
{
    private void Start()
    {
        if (Networking.PrimarySocket.Connected)
            Networking.Instantiate("Guy", NetworkReceivers.AllBuffered);
        else
        {
            Networking.PrimarySocket.connected += delegate()
            {
                Networking.Instantiate("Guy", NetworkReceivers.AllBuffered);
            };
        }
    }
}
```

Youtube video for demonstration - <https://youtu.be/-L5KE57RJSY>

Networking Destroy

This is for you to be able to destroy an object over the network.

In the example below, I destroy the enemy whenever his health is less than or equal to 0.

Forge-NetworkingDestroy.cs

```
using UnityEngine;

using BeardedManStudios.Network;

public class NetworkedEnemy : NetworkedMonoBehavior
{
    [NetSync]
    public int Health = 100;
    protected override void OwnerUpdate()
    {
        base.OwnerUpdate();

        if (Health <= 0)
        {
            Networking.Destroy(this);
        }
    }
}
```

Capturing Exceptions

At any point in a program something could go wrong with the network, or invalid values or data structures could be used which will cause an exception. In Forge we use our event driven structure in order to capture these exceptions and enable you as a developer to execute logic based on these specific exceptions. The following shows how you can filter these various exceptions.

Forge-CapturingExceptions.cs

```
// You can register a function to listen to a NetWorkers error event
NetWorker socket = Networking.Host((ushort)port, protocolType, playerCount,
isWinRT);
socket.error += SocketError;

//...

private void SocketError(System.Exception exception)
{
    if (exception is System.Net.Sockets.SocketException)
        Debug.Log("This is somekind of socket exception, could be that the
port is already in use?");
    else if (exception is NetworkException)
        Debug.Log("This is a Forge Networking specific exception");
    else if (exception is System.Exception)
        Debug.Log("It is a system exception");
    else
        Debug.Log("What is this exception?");

    Debug.Log("We are now going to log the exception with Unity
Debug.LogException");
    Debug.LogException(exception);
}
```

Here is some sample code on how to effectively use the event to check to see if a host is currently already running on the specified port. If so, then it will attempt to connect to the next port (port++). This will happen a maximum of ten times before stopping.

Forge-AttemptHostOnNextPort.cs

```
using UnityEngine;
using System.Collections;
using BeardedManStudios.Network;

public class AttemptHostOnNextPort : MonoBehaviour
{
    public bool hosting = false;
    public string ip = "127.0.0.1";
    public int port = 15937;
    public Networking.TransportationProtocolType protocolType =
Networking.TransportationProtocolType.UDP;

    #if UNITY_WINRT && !UNITY_EDITOR
    private bool isWinRT = true;
    #else
    private bool isWinRT = false;
    #endif
}
```

```

private NetWorker socket = null;

public bool proximityBasedUpdates = false;           // Only send
other player updates if they are within range
public float proximityDistance = 5.0f;              // The range for
the players to be updated within

public bool ready = false;

public int playerCount = 32;
public int maxHostTries = 10;
private int currentHostTries = 0;

private void Update()
{
    if (Input.GetKeyDown(KeyCode.C))
        StartClient();
    else if (Input.GetKeyDown(KeyCode.S))
        AttemptHosting();
}

void StartClient()
{
    hosting = false;
    Debug.Log("client port " + port);

    socket = Networking.Connect(ip, (ushort)port, protocolType, isWinRT);

    Debug.Log("client connected " + socket.Connected);

    if (proximityBasedUpdates)
        socket.MakeProximityBased(proximityDistance);

    socket.serverDisconnected += delegate(string reason)
    {
        NetworkingManager.CallOnMainThread(delegate(object[] args)
        {
            Debug.Log("The server kicked you for reason: " + reason);

        }, null);
    };

    if (socket.Connected)
    {
        Debug.Log("1");
        ready = true;
    }
    else
    {
        Debug.Log("3");
        socket.connected += delegate() { ready = true; Debug.Log("2"); };
    }
}

```



```

        Networking.SetPrimarySocket(socket);
    }

    void AttemptHosting()
    {
        if (currentHostTries++ >= maxHostTries)
            throw new System.Exception("Unable to establish host connection, max
port attempts reached");

        int tries = 10;
        hosting = true;

        Debug.Log("players " + playerCount);
        Debug.Log("protocol " + protocolType);
        Debug.Log("winrt " + isWinRT);

        socket = Networking.Host((ushort)port, protocolType, playerCount,
isWinRT);
        socket.error += PrimarySocket_error;
        socket.connected += socket_connected;
    }

    void socket_connected()
    {
        Debug.Log("Connection established on port: " + port);
        Networking.SetPrimarySocket(socket);

        if (proximityBasedUpdates)
            socket.MakeProximityBased(proximityDistance);

        socket.serverDisconnected += delegate(string reason)
        {
            NetworkingManager.CallOnMainThread(delegate(object[] args)
            {
                Debug.Log("The server kicked you for reason: " + reason);
                Application.Quit();
            }, null);
        };

        Debug.Log("h1");
        ready = true;
    }

    private void PrimarySocket_error(System.Exception exception)
    {
        if (exception is System.Net.Sockets.SocketException)
        {
            // Trying to connect to same port error code is 10048, we only want to
retry on that one
            if (((System.Net.Sockets.SocketException)exception).ErrorCode == 10048)
            {
                Debug.Log("Port " + port + " is in use, trying next port");
            }
        }
    }

```

```
    port++;  
    AttemptHosting();  
}  
else  
    Debug.LogException(exception);  
}  
else  
{  
    Debug.LogException(exception);  
}
```

```
}  
}  
}
```

Serializing Fields and Properties

Available serialization types:

- byte
- char
- short
- ushort
- bool
- int
- uint
- float
- long
- ulong
- double
- string
- Vector2
- Vector3
- Vector4
- Color
- Quaternion

Source Code:

ForgeExample_Move.cs

```
using UnityEngine;  
using System.Collections;  
using System.Collections.Generic;  
using BeardedManStudios.Network;  
using BeardedManStudios.Network.Unity;  
  
public class ForgeExample_Move : NetworkedMonoBehavior  
{  
    [NetSync("Print", NetworkCallers.Everyone)]  
    public int number = 0;  
    public TextMesh nameMesh = null;  
    private BMSByte rawMessageCache = new BMSByte();  
    public void Print()  
    {  
        Debug.Log("The number has been updated to " + number);  
    }  
    protected override void NetworkStart()  
    {  
        // Always call the base Start method
```

```

        base.NetworkStart();
        if (!IsOwner)
            return;
        // Read raw data from the network
        OwningNetWorker.rawDataRead += OwningNetWorker_rawDataRead;
    }
    private void OwningNetWorker_rawDataRead(NetworkingPlayer sender,
BMSByte data)
    {
        // In this test we are just writing a string across the network
        string message = System.Text.Encoding.UTF8.GetString(data.byteArr,
data.StartIndex(), data.Size);
        Debug.Log("Hello " + message);
    }
    protected override void OwnerUpdate()
    {
        base.OwnerUpdate();
        if (Input.GetKeyDown(KeyCode.B))
            AssignName("Brent");
        else if (Input.GetKeyDown(KeyCode.F))
            AssignName("Farris");
        if (Input.GetKey(KeyCode.UpArrow))
            transform.position += Vector3.up * 5.0f * Time.deltaTime;
        if (Input.GetKey(KeyCode.DownArrow))
            transform.position += Vector3.down * 5.0f * Time.deltaTime;
        if (Input.GetKey(KeyCode.LeftArrow))
            transform.position += Vector3.left * 5.0f * Time.deltaTime;
        if (Input.GetKey(KeyCode.RightArrow))
            transform.position += Vector3.right * 5.0f * Time.deltaTime;
        // Send a raw message across the network (total of 13 bytes)
        if (Input.GetKeyDown(KeyCode.E))
        {
            rawMessageCache.Clone(System.Text.Encoding.UTF8.GetBytes("Brent
Farris"));
            Networking.WriteRaw(OwningNetWorker, rawMessageCache,
OwnerId.ToString(), false);
        }
        if (Input.GetKeyDown(KeyCode.Space))
            number++;
    }
    private void AssignName(string newName)
    {
        NetworkingManager.Instance.SetName(newName);
        RPC("UpdatePlayers");
    }
    [BRPC]
    private void UpdatePlayers()
    {
        NetworkingManager.Instance.PollPlayerList(GetPlayers);
    }
    private void GetPlayers(List<NetworkingPlayer> players)
    {
        MainThreadManager.Run(() =>

```

```

    {
        // Note: This is not optimal, it is just for the idea
        foreach (KeyValuePair<ulong, SimpleNetworkedMonoBehavior> kv in
networkedBehaviors)
        {
            if (kv.Value is NetworkedMonoBehavior)
            {
                foreach (NetworkingPlayer player in players)
                {
                    if (kv.Value.OwnerId == player.NetworkId)
                    {
                        //TODO: Do something to the
networkedMonobehavior for this player because
                        string ownerPlayerName = player.Name;
                        string objectName = kv.Value.name;
                        NetworkedMonoBehavior nBehavior =
(NetworkedMonoBehavior)kv.Value;
                        break;
                    }
                }
            }
        }
    });
}
private void OnGUI()
{
    if (!IsOwner)
        return;
    GUILayout.Space(25);
    // The server NetworkingManager object controls how fast the
client's times are updated
    GUILayout.Label("The current server time is: " +
NetworkingManager.Instance.ServerTime);
    GUILayout.Label("Press B or F to assign your name across the
network.");
    GUILayout.Label("This will also updated get the latest list of

```

```

    players" );
    }
}

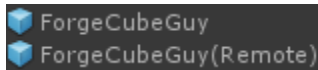
```

Youtube video for demonstration - <https://youtu.be/6tb1NVxlaOA>

Spawn Remote Object

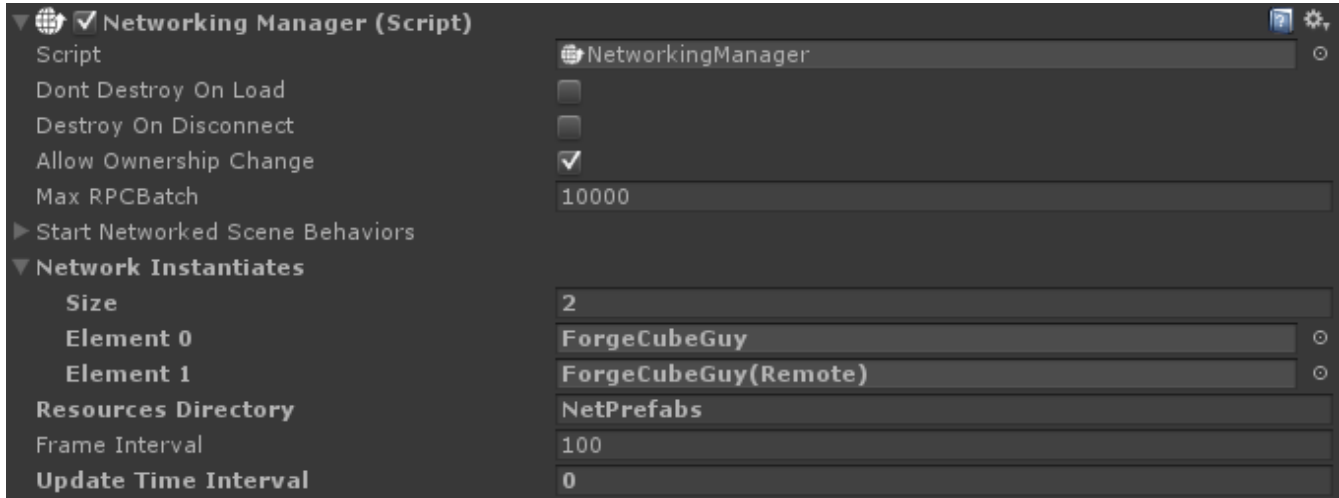
These are basically objects that are created instead of the original for the other players.

In order to set this up to work properly you will need to have your first prefab's name unique. Then the object you want to make an object a remote of it, (i.e. sphere/cube), you then would paste the exact same name, but add '(Remote)' to the end of it.



This will then tell Forge Networking that the object marked as '(Remote)' will instead be instantiated instead of the original for the other clients (including the server). So utilize this, one final step is needed.

You will need to make sure that the networking manager has both prefabs inside it as well.



There you have it! You will now spawn the remote object for the other clients while you yourself will spawn the 'hero' object as you will.

Youtube video for demonstration - <https://youtu.be/kfsHQZMiy2A>

Owning Updates

Video - <https://youtu.be/M1pQ1eaJW00>

ForgeExample_DemoOwningUpdate.cs

```
using UnityEngine;
using System.Collections;

using BeardedManStudios.Network;

public class DemoMove : NetworkedMonoBehavior
{
    //protected override void Update()
    //{
    //    base.Update();

    //    if (Input.GetKeyDown(KeyCode.Space))
    //    {
    //        if (IsOwner)
    //            transform.position += Vector3.up;
    //        else
    //            Debug.Log("You do not own this object!");
    //    }
    //}

    protected override void OwnerUpdate()
    {
        base.OwnerUpdate();

        if (Input.GetKeyDown(KeyCode.Space))
        {
            transform.position += Vector3.up;
        }
    }

    protected override void NonOwnerUpdate()
    {
        base.NonOwnerUpdate();

        if (Input.GetKeyDown(KeyCode.Space))
        {
            Debug.Log("You do not own this object!");
        }
    }
}
```

To see a full list of owning updates please check [Owner Update](#), [Non-owner Update](#), [Owner Fixed Update](#), [Non-owner Fixed Update](#) for a complete list.

Stop Interpolate on Independent NetSync

Video - <https://youtu.be/VDEZpq8iiqo>

Stop Interpolation Example

```
using UnityEngine;
using BeardedManStudios.Network;

public class NetworkedPlayerObject : NetworkedMonoBehavior
{
    //Now this will instantly change to the value rather than lerp to it.
    [NetSync(NetworkedMonoBehavior.NetSync.Interpolate.False)]
    public float Mana = 100;
    protected override void OwnerUpdate()
    {
        if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            Mana += 15;
        }
        else if (Input.GetKeyDown(KeyCode.LeftArrow))
        {
            Mana -= 15;
        }
    }
}
```

In the code example above, I simply make it so that it doesn't interpolate that NetSync variable by setting the flag to false. This will make it so that the value will instantly jump to +/- 15 depending on which key I pressed, whereas before it would lerp to +/- 15 of a difference. Go ahead and try setting it to true and false and see the differences!

HTTP Library

With the example below we go over all the functionality that you are able to do with the HTTP library. With Get/Post/Put/Delete/GetImage able to send requests on a multithreaded addition to Forge Networking.

Http Example

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using BeardedManStudios.Network;

public class GetRequesterExample : MonoBehaviour
{
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {

```



```

//Get
string url = "https://www.google.com/";
HTTP httpRequest = new HTTP(url);
httpRequest.Get((response) =>
{
    Debug.Log("Response:\n" + response);
}));

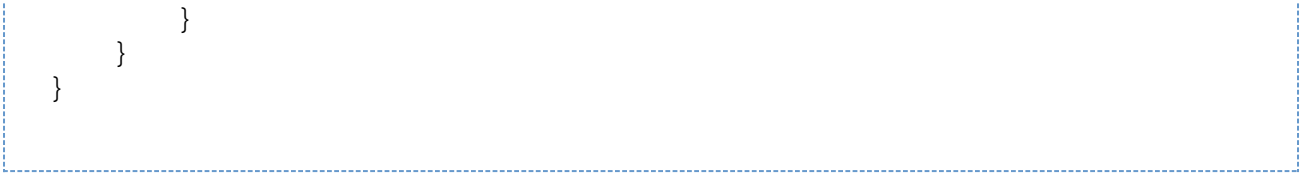
//Post
string[] postHeaders = new string[] {};
string postMessage = "testMessage";
HTTP httpPost = new HTTP(url, postHeaders);
httpRequest.Post((response) =>
{
    Debug.Log("Post Response:\n" + response);
}, postMessage);

//Put
HTTP httpPut = new HTTP(url);
httpPut.Put((response) =>
{
    Debug.Log("Put Response:\n" + response);
}));

//Delete
HTTP httpDelete = new HTTP(url);
httpPut.Delete((response) =>
{
    Debug.Log("Delete Response:\n" + response);
}));

//Delete
HTTP httpImage = new HTTP(url);
httpPut.GetImage((response) =>
{
    if (response == null)
        return; //Failed to get a proper image response
    Texture2D imageResponse = new Texture2D(2, 2);
    byte[] imageBytes = response as byte[];
    if (imageBytes == null || imageBytes.Length == 0)
        return; //There are no bytes in this response!
    imageResponse.LoadImage(imageBytes);
    Debug.Log("Image Loaded successfully!");
}));

```



Video - <https://youtu.be/8daZme5uD2I>

Binary

Understanding Binary is a necessary for understanding how data is being send across the network. Since there is a lot of information on the internet already to cover what binary is and how it is used, I am just going to show a basic brief example on it.

In our basic numbering system, we go from 0-9. Our system is a Base 10 system. Whereas binary is a Base 2.

As an example of this, counting 1010 is counted as 'One thousand and ten' in our base 10 system, in binary it is counted as 'ten'.

1000	100	10	1
1	0	1	0

In binary, it looks like this

8	4	2	1
1	0	1	0

So in our base 10 system, we can see that we do $0 \times 1 + 1 \times 10 + 0 \times 100 + 1 \times 1000$ (final answer is 1010). In binary we do $0 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8$ (final answer is 10).

Now that we understand binary, we now know how much data is being send back and forth across the network.

Things to keep in mind,

1bit = 1binary digit (off/on, 0/1, no/yes, false/true, and so on)

1 byte = 8bits (8 binary digits, as seen in the example above for our conversion)

Please do explore the internet for further understanding of how this can impact your app/game for networking.

Youtube video for demonstration: <https://www.youtube.com/watch?v=KU2gCVtMqw4>

Master Class Intermediate

Using The Cache System

The server Cache system (key-value storage) is a way to store transient data on a server session. An example of how the Cache system can be utilized is to have the server keep a running count of kills or player connections. Rather then finding a place in some class to store this information and then make special functions for sending and updating it, you can use the cache system. Then a client at any point can request that data from the server and easily process the response.

Note: In a future release the server will be able to persist that data to disk for loading later.

The Cache system is found in the BeardedManStudios.Network namespace in a class named "Cache". This is a static class, so you are able to execute methods without an instance.

The following are code snippets on how to use the Cache class on both the client and the server:

CacheSnippet.cs

```
// Server code
Cache.Set<int>("test", 9); // Saves the key "test" to be the value 9 which
the client can retrieve

// Client code
Cache.Get<int>("test", (object x) => { Debug.Log(x); }); // Prints out 9 to
the Unity Console
```

Notice how in the server code you are only required to provide a data type, a key and a value. This will create a variable in memory with the key "test" and the value of 9 (in the above example).

Something to be aware of is that each key can only have 1 instance. If you were to assign the key "test" to something else at another point in time, it will be replaced with that new value. So any client that requests the data will get the latest version of it.

Also, it is worth noting that only the server can assign key-value pairs. However both server and clients can perform Get commands.

Cache Example Script

```
using UnityEngine;
using System.Collections;
using BeardedManStudios.Network;

public class CacheExample : MonoBehaviour
{
    private int _storedInt = 0;

    private void Update()
    {
        //These keys can be in whatever keycode you want
        if (Input.GetKeyDown(KeyCode.S))
        {
            _storedInt++;
            Cache.Set<int>("test", _storedInt);
        }
        else if (Input.GetKeyDown(KeyCode.G))
        {
            Cache.Request<int>("test", (object x) =>
            {
                Debug.Log("Cache contains: " + x);
                _storedInt = (int)x;
            });
        }
    }
}
```

Youtube video for demonstration - <https://youtu.be/0RovmT1HR9w>

Working With Authoritative Server Option

Video - <https://youtu.be/cy-64YPF8Jg>

If you have been looking through the various options on the NetworkedMonoBehavior you may have found something labeled "Server is Authority" and "Client Side Prediction" when you select this option. You quite possibly were wondering what this means.

First of all lets talk about what the "Server is Authority" option does. Basically when you select this toggle box the client doesn't do anything but send its input requests to the server. These inputs are then processed on the server rather than blindly accepting variable changes and alterations from the client. Because of this the client no longer has control over the values of the objects that it "owns" (though it still is seen as the owner of the object). This means that the server is completely responsible for simulating everything including position, rotation, physics, variables marked with NetSync or added through the AddNetworkedVariable function, etc.

Okay, so if the server is calculating everything, that would mean that the client would have to wait for the server to respond with the updated positions and variables. This makes the gameplay much more secure but now we have to deal with the idea of lag. Lets say that I press a key and it takes 100 milliseconds to go to the server and 100 milliseconds to come back. That would mean that there is a 200 millisecond delay between when the client pressed the key and when their object reacted. This obviously produces a very bad experience for the client. So how do we solve it?

Introducing client side prediction! It is actually a really simple concept. Programs you see are highly predictable, if you have all of the variables it will simulate the nearly the same everywhere (unless somewhere else the variables are different). So, knowing this, you can actually simulate the same thing that the server will simulate except on the client before the server responds. All that is left to do when you do this is to make sure that if the client gets too far out of sync with the server, it is updated. This means the server is ultimately the authority.

Great! I get the concept, now show me the code!

First, derive your class from the NetworkedMonoBehavior. Now you are able to register your input events for the server and for client side prediction. The following is a code example of how to use the events and methods for getting your authoritative server and predictive client working! :)

ForgeExample_AuthoritativeController.cs

```
using UnityEngine;
using System.Collections;

using BeardedManStudios.Network;

public class ForgeExample_AuthoritativeController : NetworkedMonoBehavior
{
    /// <summary>
    /// The speed of our character
    /// </summary>
    public float speed = 5.0f;

    private void Start()
    {
        // An event that fires whenever a key down request has happened
        inputDownRequest += Input;

        // An event that fires whenever a key has been pressed down and has not
        // been released yet
        inputRequest += InputDown;

        // An event that fires whenever a key releasedrequest has happened
        inputUpRequest += InputUp;

        // An event that fires whenever a mouse button down request has happened
```

```

        mouseDownRequest += MouseDown;

        // An event that fires whenever a mouse button has been pressed down and
        has not been released yet
        mouseRequest += Mouse;

        // An event that fires whenever a mouse button up request has happened
        mouseUpRequest += MouseUp;
    }

    /// <summary>
    /// Called when there was an mouse button release request
    /// </summary>
    /// <param name="keyCode">The index of the button that was
released</param>
    private void MouseUp(int buttonIndex, int frame)
    {
        Debug.Log("The mouse button with index " + buttonIndex + " was
released!");
    }

    /// <summary>
    /// Called while mouse button was pressed down and has not been released
yet
    /// </summary>
    /// <param name="keyCode">The index of the button that was pressed
down</param>
    private void Mouse(int buttonIndex)
    {
        Debug.Log("The mouse button with index " + buttonIndex + " is currently
being pressed down!");
    }

    /// <summary>
    /// Called when there was an mouse button down request
    /// </summary>
    /// <param name="buttonIndex">The index of the button that was pressed
down</param>
    private void MouseDown(int buttonIndex, int frame)
    {
        Debug.Log("The mouse button with index " + buttonIndex + " was pressed
down!");

        if (buttonIndex == 0)
            Debug.Log("Requested frame was on " + frame + " current frame is " +
NetworkingManager.Instance.CurrentFrame);
    }

    /// <summary>
    /// Called when there was an input key release request
    /// </summary>
    /// <param name="keyCode">The key that was released</param>
    private void InputUp(KeyCode keyCode, int frame)

```

```

{
    Debug.Log(keyCode.ToString() + " was released and this is " +
(!OwningNetWorker.IsServer ? "not" : "") + " the server");
}

/// <summary>
/// Called while an input key was pressed down and has not been released
yet
/// </summary>
/// <param name="keyCode">The key that is currentlty pressed down</param>
private void Input(KeyCode keyCode, int frame)
{
    Debug.Log(keyCode.ToString() + " was pressed and this is " +
(!OwningNetWorker.IsServer ? "not" : "") + " the server");
}

/// <summary>
/// Called when there was an input key down request
/// </summary>
/// <param name="keyCode">The key that was pressed down</param>
private void InputDown(KeyCode keyCode)
{
    switch (keyCode)
    {
        case KeyCode.LeftArrow:
            transform.position += Vector3.left * speed * Time.deltaTime;
            break;
        case KeyCode.RightArrow:
            transform.position += Vector3.right * speed * Time.deltaTime;
            break;
    }
}

protected override void OwnerUpdate()
{
    base.OwnerUpdate();

    // Check for a right arrow down or up and request on server if changed
    InputCheck(KeyCode.RightArrow);

    // Check for a left arrow down or up and request on server if changed
    InputCheck(KeyCode.LeftArrow);

    // Check for a left mouse button down or up and request on server if
    changed

```

```

    MouseCheck ( 0 ) ;
  }
}

```

If you want some more examples please go check out [Input Requests!](#)

Authoritative Sync Thresholds

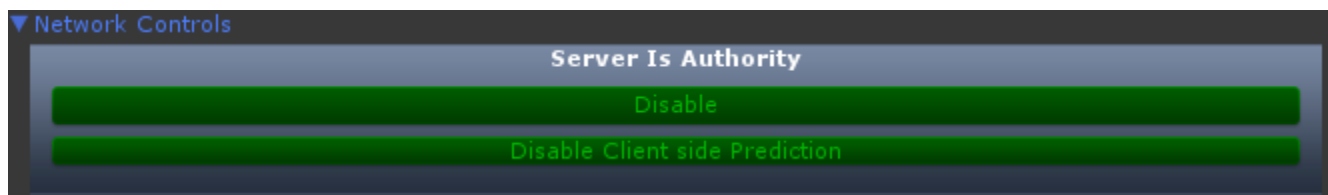
Authoritative Sync Thresholds is a way for you to properly allow the client side prediction to have a longer range of a sync threshold. This means that if the distance has been reached beyond that on the client compared to the server, then the server will automatically correct it's positioning.

Great! How do I get this setup?

Well first you will want to head over to our example scene '**AuthoritativeAndClientPrediction**' so that we can use this as a template for showing how it works. After heading there you will notice that we only spawn one type of prefab, '**ForgeCubeGuy**' under the example scene prefabs folder.

We must have **Server is Authority Enabled** you'll know when it is enabled when it turns Green and the button's text switches to 'Disable' for when you want to disable it.

Next we want to make sure we set '**Enable Client side Prediction**' doing so will make it so the client will predict it's own positioning/scale/rotation, while it will only sync to the server when the server tells it has gone too far out of sync.

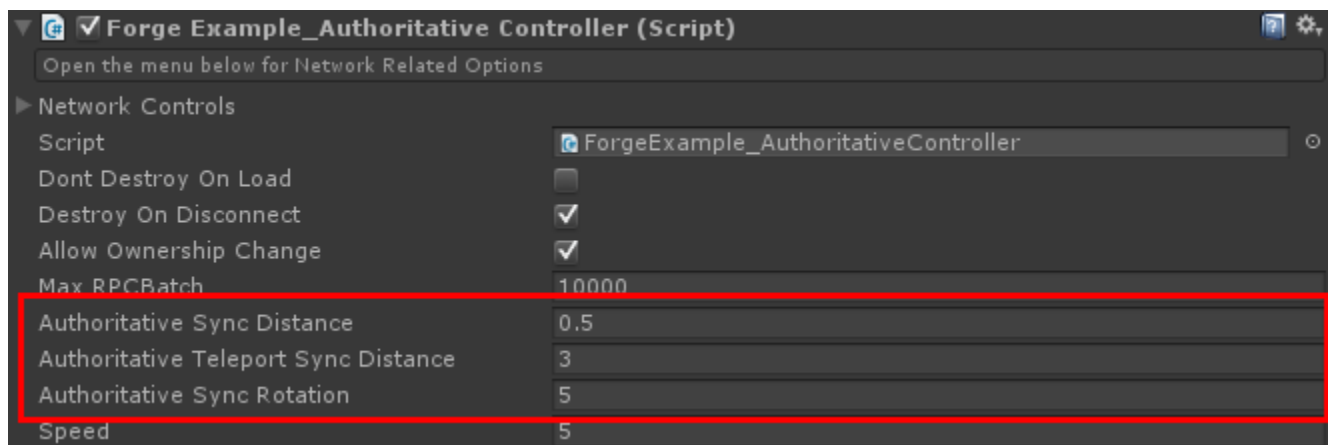


After selecting that prefab, you'll notice this portion at the bottom of any Networked Monobehavior derivative script.

Authoritative Sync Distance - This is the distance set for how far away the client side prediction can be against the server's position.

Authoritative Teleport Sync Distance - This means that after it goes past this distance, the client will automatically Teleport to it's correct positioning on the server.

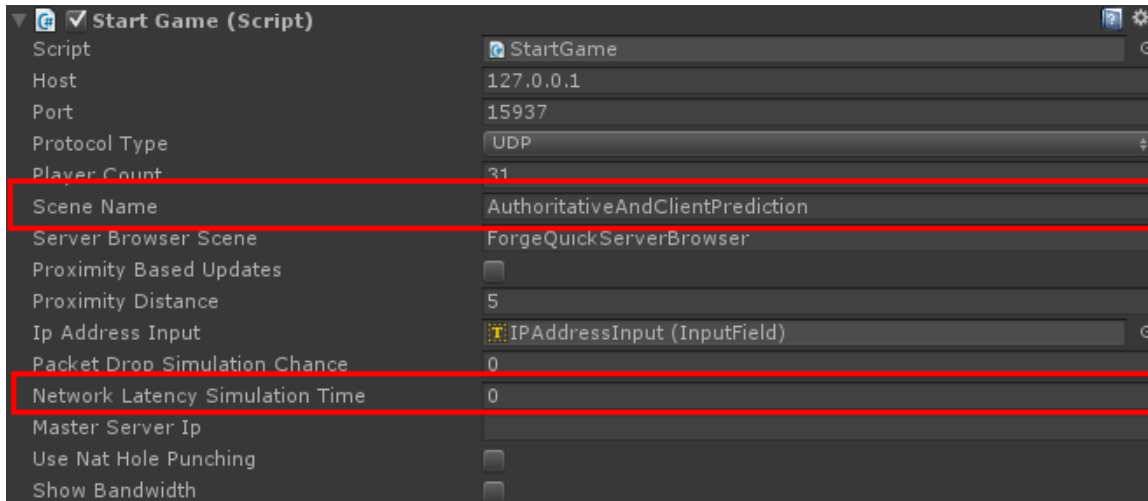
Authoritative Sync Rotation - This is the difference allowed for the client side's rotation compared to the server before automatically syncing to it's correct rotation.



Currently we have our **Authoritative Sync Distance** set to 0.5, if we change it to 5 then that will allow more breathing room so that there is no rubber banding.

From here you can experiment with changing this value to different amounts and building out the **AuthoritativeAndClientPrediction** with the **For geQuickStartMenu** set to run that particular scene.

Just make sure to set the '**Scene Name**' variable to **AuthoritativeAndClientPrediction**.



You can also mess with the **Network Latency Simulation Time** to be a value of 250-500 milliseconds so that you can see it in action of how it syncs to what the server has.

Youtube video demonstration - <https://youtu.be/PJBi8E9tyY4>

Working with Floating Point Inputs

This tutorial works best by looking at the example scene '**AuthoritativeAndClientPrediction**'.

In there you would want to click on the '**Networking Manager**' and replace the '**ForgeCubeGuy**' in both the Networking Manager's script for Network Instantiates, and Forge Example_Make Player script to use '**ForgeCubeGuyAltInput**'.

Below is the following code that was used to send this data across.

This is the code that is used for serializing the floating point valued inputs across the network which will not be authoritative. These 2 floats will be sent from the client to the server and the server will use them when calculating the core authoritative logic.

ForgeExample_AuthoritativeControllerFloats.cs

```
using UnityEngine;
using System.Collections;

using BeardedManStudios.Network;

public class ForgeExample_AuthoritativeControllerFloats :
NetworkedMonoBehavior
{
    [NetSync]
    public float horizontal = 0.0f;

    [NetSync]
    public float vertical = 0.0f;

    protected override void OwnerUpdate()
    {
        horizontal = Input.GetAxis("Horizontal");
        vertical = Input.GetAxis("Vertical");
    }
}
```

This is the core server logic for taking the input values provided by the client and doing authoritative calculations based on them. The client will also use this code for the client side prediction as well.

ForgeExample_AuthoritativeControllerBody.cs

```
using UnityEngine;
using System.Collections;

using BeardedManStudios.Network;

public class ForgeExample_AuthoritativeControllerBody :
NetworkedMonoBehavior
{
    public float speed = 5;
    public float horizontal = 0.0f;
    public float vertical = 0.0f;

    public ForgeExample_AuthoritativeControllerFloats inputController = null;

    //We leave it as a regular update because on the server we may not own
    this object
    private void Update()
    {
        horizontal = inputController.horizontal * speed * Time.deltaTime;
        vertical = inputController.vertical * speed * Time.deltaTime;

        transform.position += new Vector3(horizontal, vertical, 0);
    }
}
```

Youtube video for demonstration - <https://youtu.be/pr94vPIDtvw>

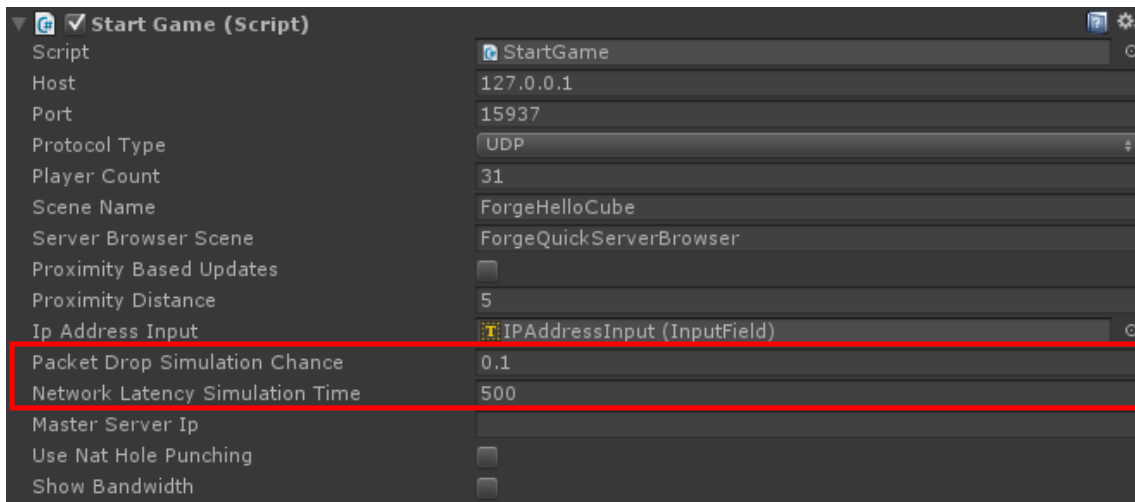
Packet Drop and Network Latency Simulations

Packet Drop and Network Latency Simulations are a great way to fully test out your game when a client has a bad connection. It is a way for you to internally test latency issues within your game by utilizing these features.

In our Start Game script you will notice that we have 'Packet Drop Simulation Chance' and 'Network Latency Simulation Time' that are exposed as variables to modify.

Packet Drop Simulation Time is a 0 - 1 % of how many packets would be lost. (i.e. 0.15 will mean that 15% of the packets will be lost)

Network Latency Simulation Time is the milliseconds it takes to send the message across one client to the other. In the below image, I set it to a 500 milliseconds delay.



After these are set, feel free to test out your game and see how it handles the latency!

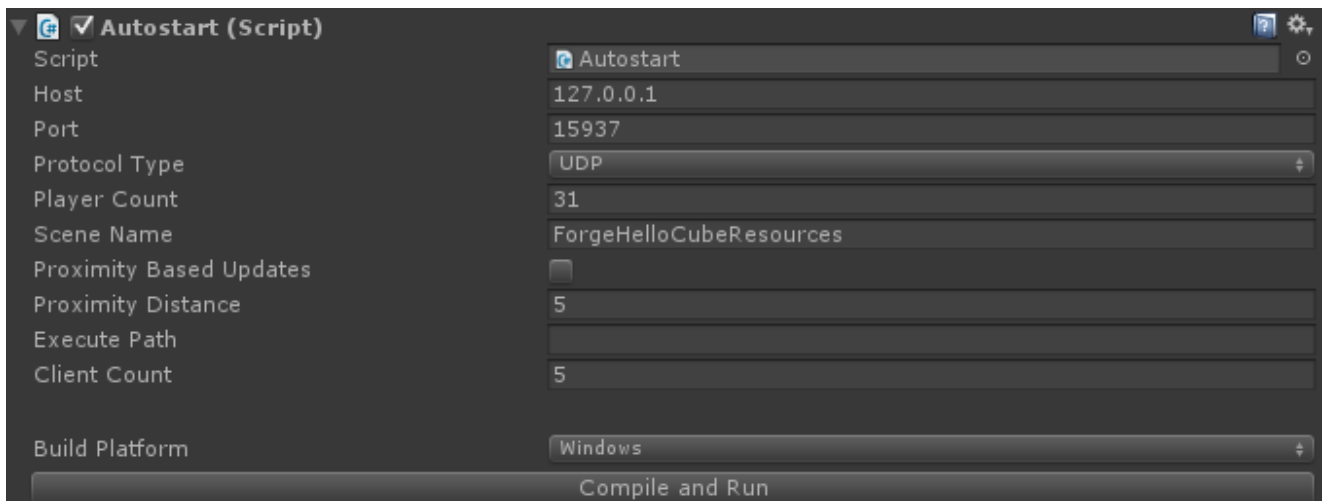
Youtube video for demonstration - <https://youtu.be/zm7wSx-eG0g>

Auto Launcher

The Autolauncher is a way for you to launch multiple clients at once rather than opening a bunch of them to do the same thing. This is just a convenient way to do so compared to the constant shift+click opening.

First go to our AutoLauncher scene included with Forge Networking.

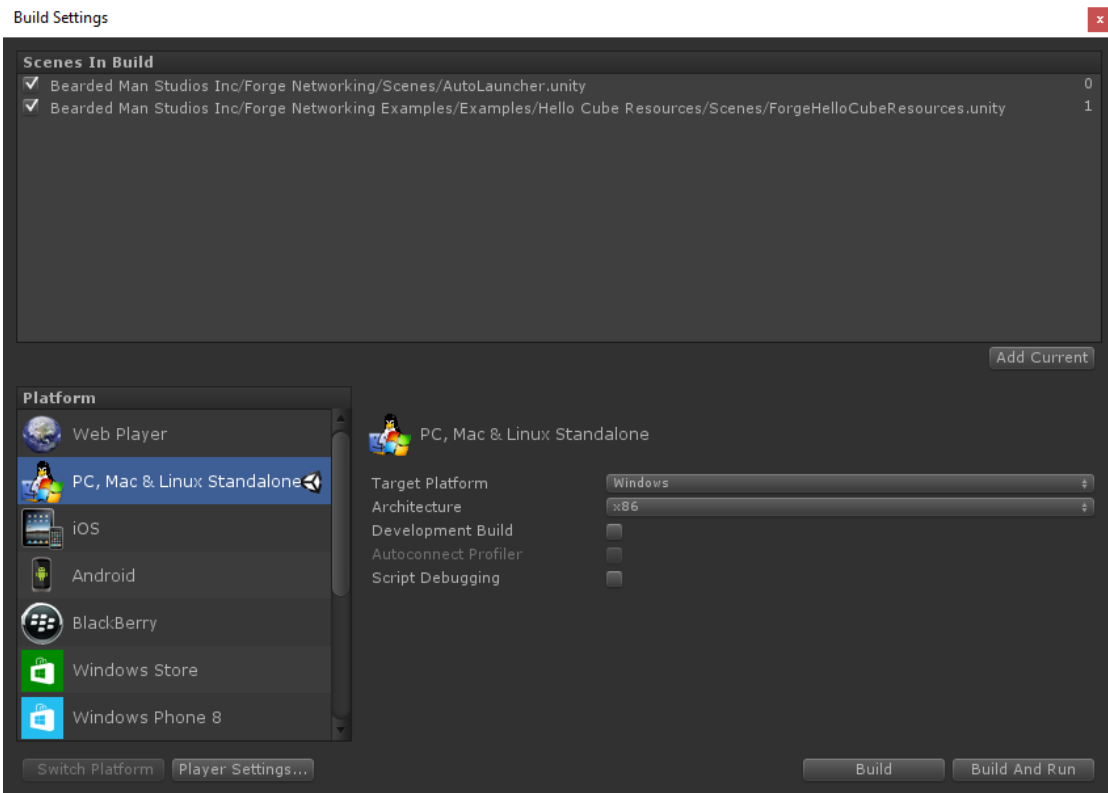
You will then see an object called 'AutoStart'. Clicking on that will give you the same properties as our ForgeQuickStartMenu, StartGame script but with a few differences.



Here you want to setup your '**Scene Name**' to your particular scene you would want to launch. You would then set how many clients to open the game under '**Client Count**'.

After doing that you would want to set your '**Build Platform**' to the platform you are building for. (Notice there isn't one for mobile as the autostart wasn't made to launch multiple clients for mobile as that would presume you have that many devices to launch on).

The next thing would be to set your build settings with the proper scenes like so.



Then hit the big '**Compile and Run**' button on the **Autostart** script when you are good to go.

Youtube video for demonstration - <https://youtu.be/X4IYC78pTeE>

Forge Transport Object

Forge Transport objects are similar to [Write Custom - Sending Classes Across the Network](#) but are instead class objects that can be derived from to do the same thing.

By deriving from `ForgeTransportObject`, you are able to make your own class with variables you want to serialize across the network. Just make sure these variables are on the approved list in [Serializing Fields and Properties](#).

In the example below, I hook into the event for when the transport has been completed, as well as print out what it's values are.

Forge Transport Object Example

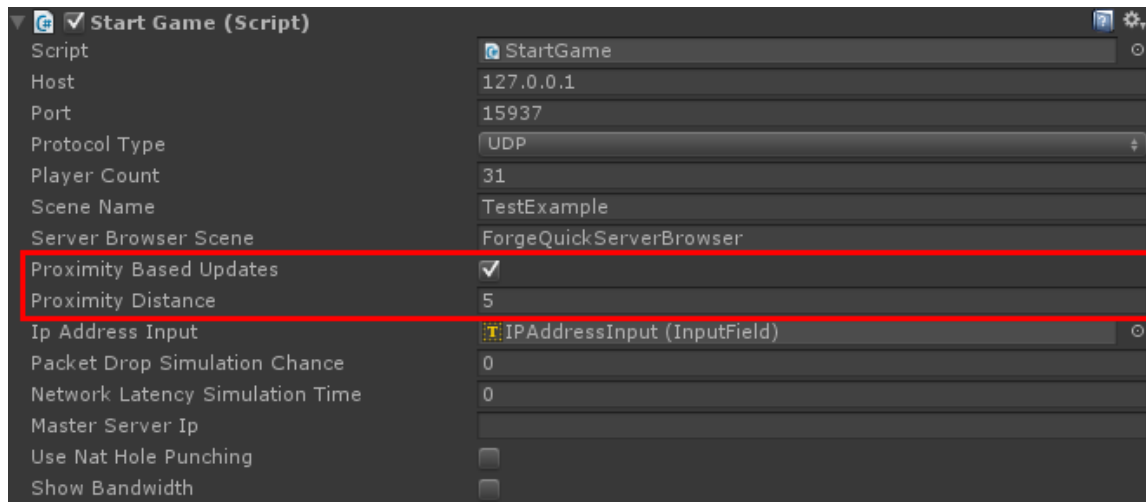
```
using UnityEngine;
using System.Collections;
using BeardedManStudios.Network;
public class TransportExample : MonoBehaviour
{
    public ForgeExample_ObjectToTransport transportObject = new
ForgeExample_ObjectToTransport();
    private void Awake()
    {
        transportObject.transportFinished += TransportComplete;
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Debug.Log("Before");
            Debug.Log(transportObject.ToString());
            transportObject.apple += 5;
            transportObject.brent += 3.14f;
            transportObject.cat = transportObject.cat == "hat" ? "bat" :
"hat";
            transportObject.dog = !transportObject.dog;
            transportObject.Send(NetworkReceivers.All);
        }
    }
    private void TransportComplete(ForgeTransportObject target)
    {
        Debug.Log("After");
        Debug.Log(target.ToString());
    }
}
```

Youtube video for further demonstration - <https://youtu.be/q59X3sfFUKs>

Proximity Based Updates and Events

Proximity based updates are a great thing! They help limit the amount of messages being sent across the network from clients to other clients. Of course the server will still receive all the calls and updates but that definitely reduces the load from the clients.

To do so, we first need to make sure proximity based updates are enabled.



Here we have 'Proximity Based Updates' enabled and the 'Proximity Distance' set to 5, so if the distance of two players is greater than 5, then we will no longer render them based on our example code below. Not only will we not render them, but we will not send constant data from clients to other clients that are not within range. The data will still be transmitted to the server, so the server will always be update to date, but other clients will not be bombarded with that object's information.

The next thing important thing to set is our player, we need to mark it as a player object otherwise proximity updates don't matter to it. As proximity based updates only work for objects that are marked as 'IsPlayer'.



To do so, you will need to open the Network Controls on your [Networked Monobehavior](#) derivative and make sure that 'Is Player' is highlighted.

Now we are good to go and the proximity based updates will work properly from one player to another.

In the example below, we simply turn off the renderer if they have left the proximity of another networked object based on the proximity updates.

ForgeExample_ProximityEvents.cs

```
public class ForgeExample_ProximityEvents : NetworkedMonoBehavior
{
    protected override void NetworkStart()
    {
        base.NetworkStart();

        // This is an event that will fire whenever the other object enters my
        proximity
        enteredProximity += (mine, other) => { Debug.Log(other.name + " entered
my (" + mine.name + ") proximity"); };

        // This is an event that will fire whenever the other object leaves my
        proximity
        exitedProximity += (mine, other) => { Debug.Log(other.name + " left my ("
+ mine.name + ") proximity"); };
    }

    // This is a function that will fire whenever the other object (this
    object) enters "my" player proximity
    protected override void EnteredProximity()
    {
        base.EnteredProximity();
        this.GetComponent<Renderer>().enabled = true;
    }

    // This is a function that will fire whenever the other object (this
    object) leaves "my" player proximity
    protected override void ExitedProximity()
    {
        base.ExitedProximity();
        this.GetComponent<Renderer>().enabled = false;
    }
}
```

Walkthroughs

Tic Tac Toe

Video - <https://youtu.be/4D4cPjszvgk>

Get the Tic Tac Toe assets/package from [this forum post!](#)

Source Code:

BoardManager.cs

```
using UnityEngine;
```

```

using System.Collections;
using System.Collections.Generic;

using BeardedManStudios.Network;

public class BoardManager : SimpleNetworkedMonoBehavior
{
    public int playerOneWins = 0;
    public int playerTwoWins = 0;
    public int catWins = 0;

    public GameObject x = null;
    public GameObject o = null;

    private List<GameObject> spawns = new List<GameObject>();
    private byte[] board = new byte[9];

    public Transform[] spaces = null;
    public bool playerOneTurn = true;

    private bool gameOver = false;

    protected override void Start()
    {
        base.Start();
        Setup();
    }

    private void Setup()
    {
        for (int i = 0; i < spawns.Count; i++)
            Destroy(spawns[i]);

        spawns.Clear();

        for (int i = 0; i < board.Length; i++)
            board[i] = 0;

        gameOver = false;
    }

    protected override void Update()
    {
        base.Update();

        if (gameOver)
        {
            if (Input.GetKeyDown(KeyCode.Space))
                RPC("ResetGame");

            return;
        }
    }
}

```



```

    if (playerOneTurn != OwningNetWorker.IsServer)
        return;

    if (Input.GetMouseButtonDown(0))
    {
        RaycastHit hit;
        if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out hit))
        {
            for (int i = 0; i < spaces.Length; i++)
            {
                if (hit.transform == spaces[i])
                {
                    RPC("PlacePiece", i);
                    break;
                }
            }
        }
    }
}

[BRPC]
private void PlacePiece(int location)
{
    if (gameOver)
        return;

    GameObject spawn = null;
    if (playerOneTurn)
    {
        spawn = Instantiate(x);
        board[location] = 1;
    }
    else
    {
        spawn = Instantiate(o);
        board[location] = 2;
    }

    spawn.transform.position = spaces[location].position;
    spawns.Add(spawn);

    playerOneTurn = !playerOneTurn;

    // Only the server will check the game rules
    if (OwningNetWorker.IsServer)
        EvaluateBoard();
}

[BRPC]
private void Winner(byte winner)
{
    gameOver = true;

```

```

switch (winner)
{
    case 1:
        Debug.Log("Player One wins!");
        playerOneWins++;
        break;
    case 2:
        Debug.Log("Player Two wins!");
        playerTwoWins++;
        break;
    default:
        Debug.Log("CAT wins!");
        catWins++;
        break;
}

Debug.Log("Scoreboard\nPlayer One: " + playerOneWins + "\nPlayer Two: " +
playerTwoWins + "\nCAT: " + catWins);
}

[BRPC]
private void ResetGame()
{
    if (!gameObject)
        return;

    Setup();

    Debug.Log("Game Reset!");
}

private void EvaluateBoard()
{
    if (board[4] != 0)
    {
        if (Equals(board[0], board[4], board[8]) ||
            Equals(board[2], board[4], board[6]) ||
            Equals(board[1], board[4], board[7]) ||
            Equals(board[3], board[4], board[5]))
        {
            RPC("Winner", board[4]);
            return;
        }
    }

    if (board[0] != 0)
    {
        if (Equals(board[0], board[1], board[2]) ||
            Equals(board[0], board[3], board[6]))
        {
            RPC("Winner", board[0]);
            return;
        }
    }
}

```

```

    }
}

if (board[8] != 0)
{
    if (Equals(board[6], board[7], board[8]) ||
        Equals(board[2], board[5], board[8]))
    {
        RPC("Winner", board[8]);
        return;
    }
}

bool cat = true;
for (int i = 0; i < board.Length; i++)
{
    if (board[i] == 0)
    {
        cat = false;
        break;
    }
}

if (cat)
    RPC("Winner", (byte)0);
}

private bool Equals(params byte[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[0] != args[i])
            return false;
    }

    return true;
}
}

```

Write Custom (Custom Data Serialization)

This is all extremely useful information for sending specific data over the network.

Write Custom - Sending Classes Across the Network

Sending classes across the network can be extremely useful if you just want to not create a lot of [Networked Monobehavior](#) or [Simple Networked Mono Behavior](#) objects in the process. This will reduce memory load and make it a lot cleaner for you to do so.

I have included an example of how you would send a class across the network with it's data serialized. Please use it below as an example to follow when you are trying to do the same.

Note: The 'CustomReadWriteID' is unique to this class, so please make different unique id's you plan on making multiple different write custom events!

Forge-SendClassAcrossNetwork.cs

```
using UnityEngine;
using System.Collections;
using BeardedManStudios.Network;
public class SendClassExample : MonoBehaviour
{
    #region Serializing Variables
    public uint CustomReadWriteID = 55000;
    public int Ammo = 0;
    public bool IsDead = false;
    public float Currency = 5.1f;
    public string PlayerName = "Brett";
    public double Experience = 0.00000001;
    public Vector2 PlayerLocalPos = new Vector2(5, 15);
    public Vector3 PlayerWorldPos = new Vector3(90, 0, 10);
    public Vector4 PlayerRotation = new Vector4(90, 0, 0);
    #endregion
    #region Not Serialized
    private bool _connected = false;
    #endregion
    #region Unity API
    private void Awake()
    {
        if (Networking.PrimarySocket == null)
            Networking.connected += Setup;
        else
            Setup(null); //Since we don't care about the socket passed
back, we are just going to pass null here
    }
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            if (!_connected)
            {
                Debug.LogError("We are not connected yet! Cannot send data
until we are");
                return;
            }
            Ammo += 1;
            IsDead = !IsDead;
            Currency += 0.5f;
            PlayerName = PlayerName == "Brett" ? "Bread" : "Brett";
            Experience += 0.00000013;
            PlayerLocalPos += Vector2.up;
            PlayerWorldPos += Vector3.down;
            PlayerRotation += Vector4.one;
            BMSByte data = Serialize(); //Serialize our properties!
            //Send it across the network! :)
            Networking.WriteCustom(CustomReadWriteID,
```

```

Networking.PrimarySocket, data, true);
    }
}
#endregion
#region Private API
/// <summary>
/// Called when we are connected successfully
/// </summary>
/// <param name="socket"></param>
private void Setup(NetWorker socket)
{
    Networking.connected -= Setup;
    _connected = true;
    Networking.PrimarySocket.AddCustomDataReadEvent(CustomReadWriteID,
ReadFromNetwork);
}
/// <summary>
/// Serializes the variables we want to
/// </summary>
/// <returns></returns>
private BMSByte Serialize()
{
    //This order is very important on the other end to deserialize!
    Make sure they match!
    object[] sendingParams = new object[] {
        Ammo,
        IsDead,
        Currency,
        PlayerName,
        Experience,
        PlayerLocalPos,
        PlayerWorldPos,
        PlayerRotation
    };
    return ObjectMapper.MapBytes(new BMSByte(), sendingParams);
}
private void ReadFromNetwork(NetworkingPlayer sender, NetworkingStream
stream)
{
    Deserialize(stream);
}
/// <summary>
/// Will deserialize all the data read from the network
/// </summary>
/// <param name="stream"></param>
private void Deserialize(NetworkingStream stream)
{
    //Must Deserialize in the order that we serialized it!
    Ammo = ObjectMapper.Map<int>(stream);
    IsDead = ObjectMapper.Map<bool>(stream);
    Currency = ObjectMapper.Map<float>(stream);
    PlayerName = ObjectMapper.Map<string>(stream);
    Experience = ObjectMapper.Map<double>(stream);
}

```

```
PlayerLocalPos = ObjectMapper.Map<Vector2>(stream);
PlayerWorldPos = ObjectMapper.Map<Vector3>(stream);
PlayerRotation = ObjectMapper.Map<Vector4>(stream);
#region Debug Information
System.Text.StringBuilder sBuilder = new
System.Text.StringBuilder();
sBuilder.Append("Update Report:");
sBuilder.Append("\nAmmo:");
sBuilder.Append(Ammo);
sBuilder.Append("\nIsDead:");
sBuilder.Append(IsDead);
sBuilder.Append("\nCurrency:");
sBuilder.Append(Currency);
sBuilder.Append("\nPlayerName:");
sBuilder.Append(PlayerName);
sBuilder.Append("\nExperience:");
sBuilder.Append(Experience);
sBuilder.Append("\nPlayerLocalPos:");
sBuilder.Append(PlayerLocalPos.ToString());
sBuilder.Append("\nPlayerWorldPos:");
sBuilder.Append(PlayerWorldPos.ToString());
sBuilder.Append("\nPlayerRotation:");
sBuilder.Append(PlayerRotation.ToString());
sBuilder.Append("\nEnd Report");
Debug.Log(sBuilder.ToString());
sBuilder = null;
#endregion
```


Bearded Man Studios, Inc.

This source code, project files, and associated files are
copyrighted by Bearded Man Studios, Inc. (2012-2015) and
may not be redistributed without written permission.

\-----+-----*/

```
using UnityEngine;
using BeardedManStudios.Network;

#if UNITY_EDITOR && !UNITY_WEBPLAYER
using System.Collections;
#endif

namespace BeardedManStudios.Network.Unity
{
    public class StartHeadlessLinux : MonoBehaviour
    {
        public string host = "127.0.0.1"; // IP address
        public int port = 15937; // Port number
        public Networking.TransportationProtocolType protocolType =
Networking.TransportationProtocolType.UDP; // Communication protocol
        public int playerCount = 31; // Maximum player count --
excluding this server
        public string sceneName = "Game"; // Scene to load
        public bool proximityBasedUpdates = false; // Only send
other player updates if they are within range
        public float proximityDistance = 5.0f; // The range for
the players to be updated within

        private NetWorker socket = null; // The initial
connection socket

        private void Awake()
        {
            DontDestroyOnLoad(gameObject);
        }

        public void Start()
        {
            StartServer();
        }

        /// <summary>
        /// This method is called when the host server button is clicked
        /// </summary>
        public void StartServer()
        {
            // Create a host connection
            socket = Networking.Host((ushort)port, protocolType, playerCount,
false);
            Go();
        }
    }
}
```



```

}

private void Go()
{
    if (proximityBasedUpdates)
        socket.MakeProximityBased(proximityDistance);

    socket.serverDisconnected += delegate(string reason)
    {
        MainThreadManager.Run(() =>
        {
            Debug.Log("The server kicked you for reason: " + reason);
            Application.Quit();
        });
    };

    #if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
    #endif

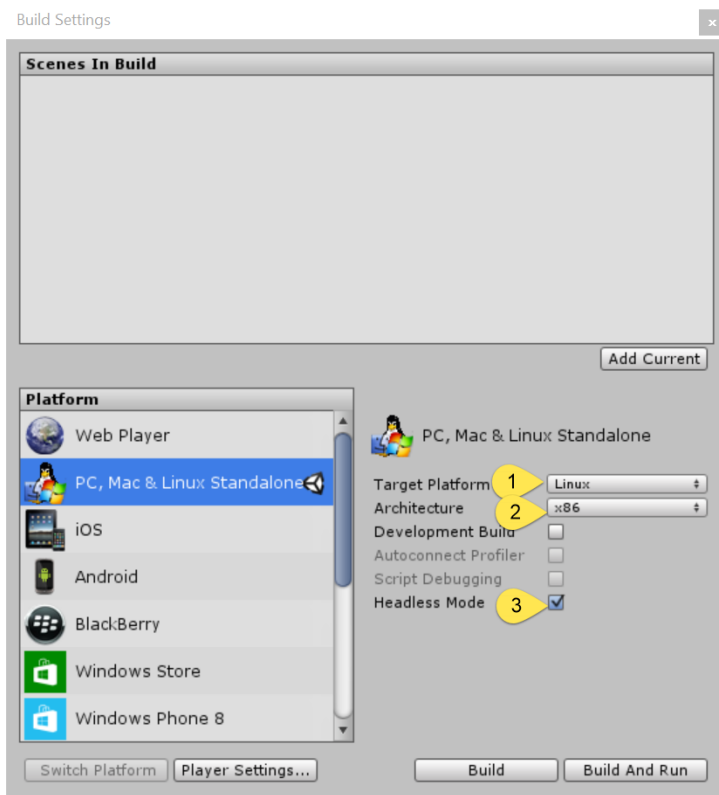
    if (socket.Connected)
        MainThreadManager.Run(LoadScene);
    else
        socket.connected += LoadScene;
}

private void LoadScene()
{
    Networking.SetPrimarySocket(socket);
    Application.LoadLevel(sceneName);
}

```

```
}  
}  
}
```

The very first thing that needs to be done is to create a headless linux build of your server.



1. Select Linux
2. Select Build Version
3. Turn on Headless Mode

Upload the Build

Take all of the contents that were output by the build and upload them to your server via FTP or SFTP.

Make File Executable

First you will need to make the file executable before you are able to run it. You can do this with the following:

```
sudo chmod +x FILE_NAME
```

Remember to replace "FILE_NAME" with your exported file name. If you exported with the name "Pickles" then the file will probably be named "Pickles.x86" in which case your command would look like: `sudo chmod +x Pickles.x86`

Run the Server

You can run the server by typing the following:

`./FILE_NAME`

Remember to replace "FILE_NAME" with the name of your build (the file in the previous example).

Possible Errors

If this is a new Ubuntu Server setup then you are probably missing a few of the required libraries in order to run the program. The following are possible errors and their solutions.

No such file or directory

This means you are not able to run the i386 (intel 32-bit) applications yet. TO get the required libraries to run these programs you will need to run the following command.

```
apt-get install libc6-i386
```

error while loading shared libraries: libstdc++.so.6: cannot open shared object file: No such file or directory

This means you are missing some of the required libs for running 32-bit applications. To be able to run these libraries you will need to run the following command.

```
apt-get install lib32stdc++6
```

It Runs but I can't Connect

This is probably because you firewall is blocking the specified port

```
sudo iptables -I INPUT 1 -p udp --dport PORT_NUMBER -j ACCEPT
sudo iptables -I INPUT 1 -p tcp --dport PORT_NUMBER -j ACCEPT
```

You will need to replace "PORT_NUMBER" with the port number you have specified in your server build. Also note that you only need to run one of these commands (probably the "udp" one). If you did not change the port number that is default for Forge Networking, then the "PORT_NUMBER" can probably be replaced with the number 15937

Forge Web Server (Addon)

Overview of Forge Web Server

The webserver allows you to interact with a Forge Networking server remotely through a web page.

Once you have downloaded and installed the Forge Web Server Package, go to Tools > Forge Networking > Web Server. An object "AutoStartWebServer" will be added. There will be a script component on the GameObject, you can set the port that the web server will be hosted on here.

You can then run the server or build a unity client and run it, the AutoStartWebServer will start the web server. You can then open your favorite web browser and navigate to the web server by typing the following into the address bar. The ip of the server running the instance, followed by a ":" and then the port specified on your AutoStartWebServer script (15940 is our default), i.e 127.0.0.1:15940 if you are hosting locally on the default (if you are having trouble connecting to a remote ip check your port forwarding and firewall).

Once you have navigated to the web server, you will be presented with some basic information about the server. Including run time, start time, and bandwidth usage.

If you navigate to the command tab, you will be presented with an interface for inputting commands into the console. Because the server runs on top of the game server, you have access to everything in the game's runtime. You can type in 'help' as a command to print out some information on the commands available. You can also write your own commands to be executed on the server.

Further security options (password protection) needs to and will be added to the Forge Web Server soon.

Youtube video for demonstration - <http://youtube.com/watch?v=H4GiVo1Kdos>

See Server Statistics and Kick a Player

Once your web server is running, clients should be able to connect as normal to the server. You should see the Player Count update when the browser page is refreshed. You'll also notice the networking usage read outs don't increase, that's because you'll need to set "Show Bandwidth" to true which is a part of the Start Game script, once this is done restart the server and you'll now see the bandwidth values increase once a player joins (and there is someone to send data to) and you refresh the page.

Several commands can be used as described in the previous tutorial, you can use the "help" command to print out all available commands. Some of the default commands can be used to do things such as: Kick players, ban players, spawn a networked object, destroy objects with a given script type.

Examples are given if you type help, but we'll give you a few here:

- Kick a player: to kick a player you can type the following format kick *playerid message*. for example kick 1 "no kick reason" to kick the player with network id 1 and supply the message "no kick reason". Note the inverted commas will be removed but must be there. There is no default display of the message but you can script this message to be displayed in your own system.
- Spawn an object: to spawn a network object you first add the prefab to the network manager, then you can type in the web console a command in the following format spawn *prefab_name*, for example spawn Zombie.
- Destroy all objects with a given script type: to spawn an object type a command with the following format destroyAll *script_type*, for example destroyAll ForgeZombie to destroy all objects with a script of type ForgeZombie attached.

Youtube video for demonstration - <http://youtube.com/watch?v=oDyF603jPBY>

Webpages Overview and Editing

You will likely want to adjust the default html of the web pages, in your unity project you can navigate to Forge Networking Web Server > Resources > www > html. In this sub directory you will find the default html pages used, the 404 page is the page the user is directed to if the url is going to a page that doesn't exist, the index and command files are the pages shown in the previous tutorials.

If you open the index page you will see the files are standard html which can be edited as needed. You will see unusual notation between "<% %>" braces, this is how data is input to the web pages, this will be explained more in a later tutorial. For a more complicated example of what you can do see the commands page, you will see javascript/JQuery can be used like a standard html page.

Youtube video for demonstration - <http://youtube.com/watch?v=B48ulYmqzsk>

MVC Controllers and Actions

Controllers and actions allow you setup pages and sub pages with data being dynamically added. As discussed in the previous article if you look in the html files under Forge Networking Web Server > Resources > www > html you will see several tags between "<% %>" braces. Controllers allow you to fill this space with data generated by our server app. For this tutorial we'll take Forge Networking Web Server > Scripts > Controllers > Index.cs as an example.

You can see in the constructor of the class we add values to the inherited variables dictionary. The key given for each variable corresponds to the key between the "<% %>" braces, if they match the value of the a key the html tag will be replaced by the value returned by the delegate/lambda/method specified by the variables dictionary. More on this in the next tutorial.

The Render method will be called when the page is open and will render the page. However, you can also use actions; any public method that returns a string counts as an action. The action can be accessed as a sub-page of the url. For example www.domain.com/Index/Players, will access the Players Action of the Index controller. While accessing the action the Players method replaces the Render method, and allows you to set different variables or load completely different html pages.

Example Action

```
public string Players(params object[] args)
{
    return PresentHTML("index");
}
```

Using JQuery can allow you to execute Actions without leaving the current page, see the Cmd controller for an example.

Youtube video for demonstration - <http://youtube.com/watch?v=AM5ZWrdHcBM>

How to use Replacement Variables

Replacement Variables are found in the html pages, the syntax involves a "<%" followed by a variable name followed by a closing "%>". These variables represent the way dynamic content/information can be added into the html pages of your forge web server. You can find all the html files at Forge Networking Web Server > Resources > www > html.

For example we could write

example replacement variable

```
<% testValue %>
```

in the index.html page. If we then go to Forge Networking Web Server > Scripts > Controllers > Index.cs we can see the Controller handling the index page (see the previous tutorial). You should see several examples in the Controller already for specifying a replacement variable.

The syntax is as follows,

```
variables.Add(variable_name, method)
```

, so following our example we could write:

```
variables.Add("testValue", GetTestValue);

private string GetTestValue(){
    return "this is a test variable";
}
```

The value would then be written out.

Youtube video for demonstration - <http://youtube.com/watch?v=x285Q9Tzetg>

Writing Custom Web Server Commands

Forge Web Server has a number of commands already available which you can see by typing "help" into the command window as discussed in previous tutorials. You can write your own custom commands for the web server easily. First navigate to Forge Networking Web Server > Scripts > Controllers > Console, in this class you are able to write your own methods that act as a custom command.

For a method to act as a custom command, it must be public, return string and take params string[] as an argument. After this you can write a method as normal to execute code when the command is called. If you wish to return a string to the server you can do so by first using the AddData method. And then returning the Render method. This is actually creating a JSON object which will send the string to the web page.

```
AddData("Echo " + args[0] + " from web server");
return Render();
```

Finally now that you have a working command, all you need to do is give some information about the command to be displayed when the help command is input. This can be done by adding the HelpInfo attribute to the method. The first value is a description of the command. The second parameter is an array of strings, each being a description for each argument. The third parameter is a complete example of the command. Now if you type help you should see your command appear in the results. Your command will also be able to be executed with the name of the method being the command's name.

Example custom command

```
[HelpInfo("This will print a generic string to the command window and to  
unity console", new string[] { "String that will be echoed out" }, "echo  
\"Hello World!\"")]  
public string Echo(params string[] args)  
{  
    if (args.Length != 1)  
        return Error("There needs to be exactly 1 argument for this command");  
    UnityEngine.Debug.Log("Echo " + args[0] + " from web server");  
    AddData("Echo " + args[0] + " from web server");  
    return Render();  
}
```

Youtube video for demonstration - <http://youtube.com/watch?v=TaLy9pC1sMI>

Creating a Custom Controller and Action

You can create custom pages for the web server in the form of Controllers and Actions. First you'll want to create a new html page under the Forge Networking Web Server > Resources > www > html directory. The format is standard html apart from the replacement variables. The name of the html file **does not** correspond to the url of the page.

Next you'll want to create a new C# Controller class under the Networking Web Server > Scripts > Controllers directory. The class must inherit from the PageController class. There needs to be a base constructor and an override of the Render method. The name of the Controller class **will be the url of the page**. For example if you create an AboutUs Controller class the url to access the page will be domainname.com/aboutus (note: not case sensitive).

The Render method that is overridden needs to return a string, this string is the page's html. Typically you will want to get the html from the html file you made, use the PresentHTML method to access it, the argument will be the filename of the html file in the html folder.

Example Controller

```
using UnityEngine;
using System.Collections;

namespace BeardedManStudios.Network.WebServer.Controller{
    public class AboutUs : PageController{
        //required base constructor
        public AboutUs() : base(){

        }

        //default method called when getting data
        public override string Render(){
            //there would need to be an html file called about, this would simply
            send that html file's content.
            return PresentHTML("about");
        }

        //an example action
        public string Me(params string[] args){
            return PresentHTML("about");
        }
    }
}
```

See the MVC Controllers and Actions method for more information on what you can do with controllers and actions.

Youtube video for demonstration - <http://youtube.com/watch?v=s4SsOJ-MIPg>

Master Server

Building a Master Server from Unity

Building your master server Unity instance is simple in Forge Networking, we have done all the hard work for you!

1. Open your build settings
2. Make the "MasterServer" scene the first scene
3. Choose your supported build target platform
4. Build the Unity instance and run it on the desired target machine

Note: You do not need to build an entire new project to build the master server, you can build it from within the same project that you are currently working in.

Registering with the Master Server

To register your a host with the master server you must first make sure that the host has successfully connected. You can do this by using the "connected" event on the NetWorker. Once the host has successfully established a connection you can then use the static register host method found in the ForgeMasterServer class.

RegisterOnMasterServer.cs

```
// string masterServerIp    = The host address of the master server
// ushort port              = The port number of this server (NOT THE
MASTER SERVER PORT)
// int playerCount          = The maximum player count allowed on the
server
// string gameName          = The name of this server
// string gameType          = The current game type (like "Deathmatch")
// string comment            = The comment for the game (Allow users to post
their comment for something like loading screen)
// string password          = The password that is required to join this
server
// string sceneName         = The name of the scene that is currently
running on the server
ForgeMasterServer.RegisterServer(masterServerIp, (ushort)port, playerCount,
gameName, gameType, comment, password, sceneName);
```

Getting Host List from Master Server

One of the main things that need to be done is to get the list of hosts from the Master Server. This can be done by calling the static "GetHosts" method found in the ForgeMasterServer class. This takes in an action (method, delegate or lambda expression) which will then be used with the returned list of hosts.

GetHostsFromMasterServer.cs

```
// string masterServerIp    = The host address of the master server
// int pageNumber           = This is used for server pagination
(ForgeMasterServer.COUNT_PER_PAGE has the count returned per page)
ForgeMasterServer.GetHosts(masterServerIp, pageNumber, (HostInfo[] hosts)
=>
{
    // This is the function that will be called once a host list is
received
    BeardedManStudios.Network.Unity.MainThreadManager.Run(delegate()
    {
        // If you are manipulating things like Unity UI then you will need to
run your logic on the main thread
        foreach (HostInfo host in hosts)
        {
            Debug.Log(host.name + " at " + host.IpAddress + ":" + host.port);
        }
    });
});
```

Getting Host List from Master Server - Pagination

Pagination is a way of limiting the amount of information sent at a single given time. Often you will only need a few results at once and then ping the Master Server for the next set of hosts. By default the amount of hosts returned from the static "GetHosts" method of the ForgeMasterServer class will be 15.

Changing the Returned Server Count

You can change this by going into the ForgeMasterServer.cs file and changing the value of the "COUNT_PER_PAGE" constant variable.

How Pagination Works

The second argument to the static "ForgeMasterServer.GetHosts" method is the "pageNumber". This is the current "page" that you are on for hosts. This is a zero based number and must start from 0 if you want the first items in the list.

When you pass a number into this method it will get that page number. So if the default return count is 15 and I pass page number 0 then I will get hosts 0 - 14 (zero based array). If I were to pass in the value 1 then I will get hosts 15 - 29. If I pass a page number of 2 then I will get hosts 30 - 44 and so on.

Removing Host From Master Server

There are no required steps to the removal process of the Master Server. When the server that is registered disconnects the master server will recognize this within ten (10) seconds and remove the host. If you would like to shorten this time (which uses network more bandwidth) then you can reduce the time inside the inspector of the "MasterServer" object in the "MasterServer" scene. Note that the times here are in milliseconds so "2500" is 2.5 seconds and "10000" is 10 seconds.

Updating Player Count on Master Server

You always want to have an updated player count for your master server so that users know what size server they are joining. To do this you can use the "UpdateServer" method found on the static ForgeMasterServer class. An easy way to automatically add and remove users is to hook into the "playerConnected" and "playerDisconnected" events found on the NetWorker.

UpdatePlayerCountOnMasterServer.cs

```
// string masterServerIp    = The host address of the master server
// ushort port              = The port number of this server (NOT THE
MASTER SERVER PORT)
// int playerCount          = The current number of players on the server
ForgeMasterServer.UpdateServer(masterServerIp, port, playerCount);
```

Re-Registering Info on Master Server

If a server needs to change its settings on the fly like the max player count, scene name, password or anything else, you can do this by just calling the register function again for the server. The Master Server will identify that this is an update request and update the server listing appropriately.

RegisterOnMasterServer.cs

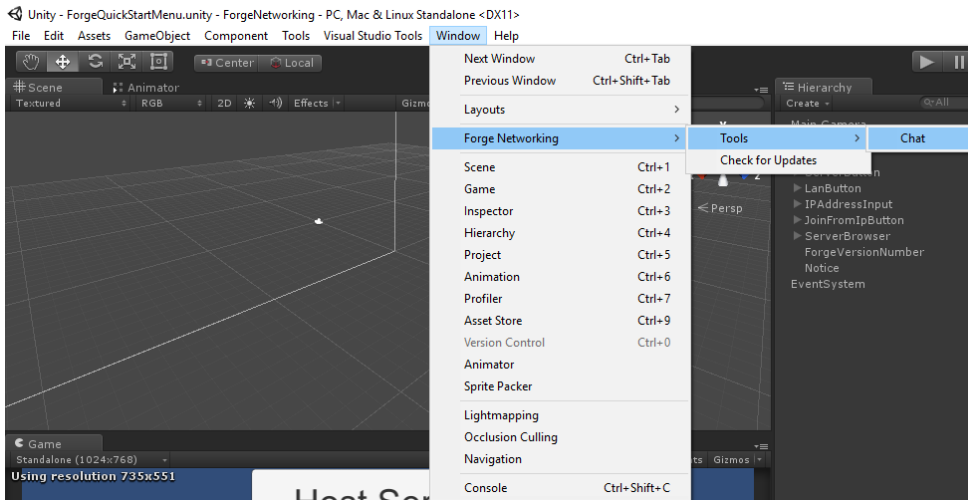
```
// string masterServerIp    = The host address of the master server
// ushort port              = The port number of this server (NOT THE
MASTER SERVER PORT)
// int playerCount          = The maximum player count allowed on the
server
// string gameName          = The name of this server
// string gameType          = The current game type (like "Deathmatch")
// string comment            = The comment for the game (Allow users to post
their comment for something like loading screen)
// string password          = The password that is required to join this
server
// string sceneName         = The name of the scene that is currently
running on the server
ForgeMasterServer.RegisterServer(masterServerIp, (ushort)port, playerCount,
gameName, gameType, comment, password, sceneName);
```

Forge Utilities

As we expand Forge, we will be increasing the Utilities documentation here, for now it just contains [Chat System](#) that you can easily implement into your app/game.

Chat System

Creating a chat system is extremely easy with Forge Networking. Under Windows->Forge Networking->Tools->Chat, you are able to easily slap in a Chat window into the scene that you can customize to your liking.



This is already hooked up to Forge Networking and you can start chatting away with other clients!

Youtube video demonstration - <http://youtube.com/watch?v=mE4SzeI4tZU>