Game Networking for Beginners with Unity3D

A Step by Step Guide to Multi-Player Gaming

By: Laurence Grant, 3DMUVE Indie Games

Table of Contents

About the Author 7

About Unity 8

Introduction 9

Game Networking Engine Designs 11

Master Game Server (MGS) 11

Peer-to-Peer (P2P) 12

Game Server (GS) 12

Game Networking Engine Styles 12

Dedicated / NON Client 12

Non Dedicated / With Client 13

Relay 13

Authoritative 13

Hybrid 14

LAG: 16

1. Prediction 16

2. Smoothing 17

Network Connectivity and Firewalls: 17

IP Addresses and Ports 17

Network IP Address 17

IPv4 addresses 17

Network Port 18

Network Address Translation or NAT. 18

NAT Punch Through / NPT 20

General Coding Overview 22

Converting between C# and Java Script 22

Tags 23

Layers 23

Overview of our Networking Tutorial: 24

Demo Overview 24

Basic Networking API's 24
TCP vs UDP: 25
Unity3D Networking API's 25
NetworkView 25
Instantiating a Networked Object 27
Building our Game: 27
Building the Game Level: 27
Creating the Floor 27
Creating the Scenery: 28
Creating the Player Character 28
Player Controller: 29
Player Spawn Area: 32
Pickup Spawn Area: 32
Overview of Networking Classes we'll create for our Demo 34
1. NetworkMasterServer 34
2. NetworkLoadLevel 34
3. InstantiatePlayer 34
4. NetworkRigidBody 34
5. PlayerSpawn 35
6. BallSpawnManager 35
7. BallController 35
8. BallManager 36
Analysis and Code of each Network Class 37
NetworkMasterServer: 37
Network.InitializeServer 37
MasterServer.RegisterHost 37
Network.Connect 37
ClearHostList 38
RequestHostList 38
PollHostList 38
NetworkMasterServer code 38
NetworkLoadLevel: 45

networkView.RPC 45
LoadLevel........ 45
NetworkLoadLevel code........ 46
InstantiatePlayer........ 49
PlayerAvatar 49
Spawn........ 49
InstantiatePlayerOnNetworkLoadedLevel 50
OnPlayerDisconnected........ 50
InstantiatePlayer code 50
PlayerSpawn Public Variables Setup 52
PlayerSpawn:........ 55
Renaming Remote Objects:........ 56
PlayerSpawn code:........ 56
NetworkRigidbody: 59
NetworkRigidbody OnSerializeNetworkView........ 60
NetworkRigidbody code: (original code strike-through, see new code below under 2nd edition)........ 61
~~2nd Edition:~~........ 63
3rd Edition: 63
Working with Pickups:........ 68
InstantiatePlayer:........ 68
BallSpawnManager:........ 68
BallController:........ 68
BallManager: 68
InstantiatePlayer – Updated for Networking: 70
InstantiatePlayer OnPlayerConnected........ 70
InstantiatePlayer OnPlayerDisconnected........ 71
InstantiatePlayer Code........ 71
BallSpawnManager:........ 74
BallSpawnManager Update........ 74
BallSpawnManager OnSpawnBall........ 75
BallSpawnManager OnInitBall........ 75
BallSpawnManager Code: 75

Ball Controller: 80
BallController Update 80
BallController OnCollisionEnter 80
BallController OnPickupBall 80
BallController OnThrowBall 80
BallController UpdatePlayerState 81
BallController OnUpdatePlayerState 81
BallController Code: 81
Ball Manager: 86
BallManager Update 86
BallManager OnCollisionEnter 86
BallManager ReSetBallBackToSpawnArea 86
Ball Manager Code: 86
Updating Prefabs: 89
Updating the Player Prefab 89
Updating the BallPickup Prefab 90
Initializing Public Variables: 91
BallSpawnManager Public Variables: 91
1. BallSpawnArea 91
2. BallPrefabToSpawn 91
3. SpawnedBalls 91
4. SpawnDelay 91
BallController Public Variables: 92
1. Pickedup 92
2. Righthand 92
3. ThrowForce 92
4. Pickedup Ball Spawn ID 92
5. The Ball Throw Model 93
6. Player Body 93
BallManager Public Variables: 93
1. Thrown 93
2. Pickedup 93

3. Ball Spawn Element ID ... 93
4. Thrown Duration Timer ... 94
5. Just Spawned ... 94
6. Pickup Delay ... 94
CONGRATULATIONS, we're done: ... 94
Appendix A - Managing dynamic Objects ... 95
Creating a Throwing Spear ... 95
Spear Code: ... 96
Spear Controller ... 96
SpearController Code: ... 96
Spear Manager ... 100
SpearManager Code: ... 100
Spear Controller Variables: ... 101
Spear Manager Variables: ... 103
Buffered RPC commands ... 104
Networking Groups ... 105
SetLevelPrefix ... 106
Appendix C – Try This ... 107
Appendix D – Object to Script Associations ... 108
MasterGameServerLobby Level ... 108
TestGameLevel Level ... 108
Conclusion ... 109

About the Author

Laurence Grant is an accomplished IT professional with nearly 30 years professional IT, programming, database, management and technology specialist. In 2005 Larry was recognized as CTO of the year by both Oracle Corporation and Info World Magazine. Today Larry is an Enterprise Apps Cloud Computing Specialist, working with leading fortune 500 companies building the latest, state of the art, enterprise clouds. Larry has also been an avid Indie Developer focused on building his own engines from scratch. Larry started developing early share ware titles in DOS and assembly language and eventually embraced C++/DirectX on Windows when it became available. Larry later moved to C#/XNA where he released BLOCKBuster on Xbox Indie Games. Unity3D was the first commercial engine Larry embraced and quickly learned to master its capabilities. Today Larry is working on "Pulsar" a team based multi-player space combat game using Unity3D. Larry is an accomplished author with the very successful Unity3D Networking eBook. Larry is the founder and proprietor of 3DMUVE Indie Game Studio. Larry is a member in good standing with the IGDA, GameDev.net and more.

About Unity

Unity (also called **Unity3D**) is a cross-platform game engine with a built-in IDE developed by Unity Technologies. It is used to develop videogames for web plugins, desktop platforms, consoles and mobile devices, and is utilized by over one million developers. It grew from an OS X supported game development tool in 2005 to a multi-platform game engine.

As of this publishing, Unity currently supports development for iOS, Android, Windows, Blackberry 10, OS X, Linux, web browsers, Flash, PlayStation 3, Xbox 360, Windows Phone, and Wii U.[2][3] The game engine is downloadable from their website in two different versions: Unity and Unity Pro. Unity provides a free license for indie developers, or the commercial Unity Pro license which for a fee provides additional features. In this eBook we use just the features provided in the free version of Unity.

Introduction

Hi everyone. Thanks for reading my eBook. There are several networking tutorials I've found on Unity3D Networking, but they were inadequate to me. They mostly said things like:

- copy this
- paste this
- do that
- delete this
- etc…

Although it was ok for me as I've had networking experience in the past I thought goodness, anyone who hasn't had experience with networking in the past might really struggle with these tutorials as they don't explain what they are having you do or why. Therefore, I've decided to write my own to help anyone who wants to add networking to their Unity 3D game. I'm sure for some game networking experts, they might have different opinions on a lot of what I introduce here, and I don't claim to be an expert by any stretch. I design very practically and implement a good-enough approach as we are making games after all ☺. My intent with this tutorial is to serve as an introduction to game networking and to provide the skills and knowledge required to get you started developing multi-player games in Unity3D. Where you take it from there is up to you. I will have a few areas of "do this", "paste that", but they will almost always be lead with an explanation of what you're doing and why.

Special Thanks

Much of the original networking code I'm using was taken from the Star Trooper Multi-Player Tutorial by Andrius Kuznecovas. Without his sample I would have had a much harder time getting up to speed on Unity3D networking. I've taken his concepts and tried to provide a lot more detail on what I'm doing and why I'm doing it. I've also replaced the use of Star Trooper with a very simple demo that we will build inline ourselves in this tutorial. I hate tutorials that use existing code that does a whole lot more than what we're trying to teach. I like to dumb it down to the basics and show what's needed without a lot of extra complexity. I hope you find this tutorial beneficial.

To accompany this eBook I've included two additional files:

Accompanying material can be download from:
http://www.3dmuve.com/ebook/networkingforbeginnerswithunity3d_content.zip

3DMUVE Networking Tutorial for Unity3D - Code Only Package.unitypackage

This is a Unity3D Package File containing all the source code discussed in this eBook. Instead of retyping the code, or trying to cut/paste, it's best to use this package. Leverage the eBook to understand what's being done and following the steps on how to create the ingame objects and assemble everything. If you run into difficulty the next file includes the completed project in its entirety.

3DMUVE NetworkingTutorial for Unity3D full Unity3D Project.zip

This is a copy of the full Unity3D project that I developed for this eBook. I would recommend only using it if you get stuck and need to look at a completed project to try and figure out where things differ. If you find any mistakes between what's documented in the eBook and this project please let me know.

Let's start with understanding some gaming concepts. I'm going to focus on just one, the "Hybrid Engine", which in my opinion is the most functional style of networking for most multi-player games, especially real-time games. It's important to understand there are many ways to implement a networked game, and even though we might use the same underlying low-level calls, one design isn't necessarily better than another. Each game has different needs and as a lead designer and developer we need to work together to decide which is best for our games. I lean towards what I've coined as the "Hybrid" style networking engine, but this isn't an industry term, and I'm sure a lot of other gaming implementations do something similar. We will cover "Authoritative", "Relay" and "Hybrid" and even develop a little of each since "Hybrid" is basically a combination of the other two.

To start with we'll discuss:

1. What a Master Game Server is
2. What a Game Server is
3. LAG, Prediction and Smoothing
4. Connectivity and challenges with firewalls,
5. Coding Sample

Building a sample application to test our code

Finally we'll discuss the Unity 3D API's used to implement the code.

- NetworkView
  - State Synchronization
  - RPC calls
- Instantiating of networked objects
- Managing networked objects

This tutorial develops a Windows based game, but from a networking perspective I believe everything covered will apply to all the Unity3D supported platforms. The only real thing that makes this Windows specific is the Input checks of the keyboard and mouse. If you wanted to port this tutorial to run on an iPad for example it should work just fine but the Input would have to be updated to use a less brute force approach to checking Input. I haven't tried this yet but unless the Unity3D networking API's are doing something under the covers, I believe you should be able to play cross-platform too, which means a user on an iPad should be able to interact with a user on a PC for example. I haven't read anywhere that this is blocked.

Game Networking Engine Designs

There are many ways you can design a networked game. Some games run a single game server on a publicly accessible computer somewhere and gamers can connect to it remotely without entering anything special because the client is already configured "at the factory" if you will to connect to the one hosted game server. In other cases the user might be required to enter the credentials of the game server to control what game they connect to and in another design there might be a menu displayed which allows the game client to choose what they want. We'll examine these in a little more detail, and then we'll develop a sample that uses a dynamically generated selection menu.

Master Game Server (MGS)

A Master Game Server (MGS) can be thought of as a lobby server. It's where your game client first connects to see what games are already available. You can choose to join an existing game or start a new game. Unity provides a MGS for testing as well as the source code for you to run your own MGS customized for just your game, available here (http://www.unity3d.com/master-server/index.html). In its simplest form an MGS provides a list of existing games. It might show the game name, a description, number of connected players, information about location or network lag, or other details to help you decide what game to join. You might create/join a private game for just your friends to play. You can extend the MGS to support buddy lists, friend chat, and other features to help the gamers with game matching or the social aspects of cooperative play. The MGS does NOT perform any game related tasks, except with initial connectivity needs. It's strictly the central hub for matchmaking and coordination of players joining games. Once a game is underway it's handled by the Game Server (GS) and has no further coordination with the MGS.

Peer-to-Peer (P2P)

A P2P game networking design is when each game client has a network connection to every other game client and they're able to send commands to one another to provide appropriate updates. Consider a two player game such as chess, each player client would tell the other player client of its move, and they would continue to exchange updates throughout the game. In the P2P design each client is equal in power and makes decisions for the game. Its common in a turn based game where the gamer takes a turn and the local game client performs 100% processing of that turn then tells the other gaming clients what it did. As each gamer takes their turn, 100% of the processing is performed by the local game client and the results are passed on. It's less common to see this style implementation used in a fast paced game such as a FPS.

Another challenge with P2P is an increased risk of cheating. If there are no controls in place across all game clients to ensure each move being performed is valid, then a compromised game client could conceivably cheat with no restrictions.

We also don't see P2P used often with games requiring large numbers of users due to the complexity of ensuring all clients can connect directly to all other clients. When we get into coding we'll show how to setup NAT Punch Through (we'll discuss in more detail later), a technique used to bypass firewalls which would otherwise prevent gamers from connecting to a non-publicly hosted game server. This process is rather tricky for game clients connecting to a single game server. If all clients had to connect to all other clients we'd have to perform this step again and again which can be very complicated, prone to failure, and not necessarily needed.

Game Server (GS)

The Game Server (GS) is the main coordinator for a new game. The main purpose of the GS is to coordinate connectivity of all the clients. Most games don't have every client connect directly to every other client, unless it's only a two player game (or very few players.) We usually design a GS as the central hub for all clients to connect. A client talks to other clients by sending commands to the GS and the GS relays those commands to all the other clients.

Game Networking Engine Styles

There are several common types of Game Servers people implement:

Dedicated / NON Client

The Dedicated Game Server doesn't also run a client. It's often used for MMOPG's where the players themselves don't have the option to host their own game, but can only join an existing game. Another use case is to let the player host their own game on a dedicated computer. Because there is no client also running performance is improved which can help support more clients, or possibly reduce lag due to the server running slow and contending for resources with the client. We don't see this use case often as it requires a second dedicated server which isn't usually available. Some games focused at a small population of hard core games, such as Half

Life's Counter Strike, will sometimes provide the option for a player hosted dedicated Game Server.

Non Dedicated / With Client

Most games that allow the player to host their own game allow the player to also run as a player (client) on that game at the same time, from the same computer. The benefit is that it makes it easy for a player to host their own game without needing a second computer or complex networking setup. The disadvantage is that the GS is also coordinating all networking to all clients and can affect the game play for everyone if the local client is slowing things down. It can also affect the local client because the server is contending for resources.

We'll be developing a non-dedicates game server for our game, but we'll be offloading some of the traditional Game Server processing to other clients using my Hybrid approach.

Regardless of Dedicated or Non-Dedicated, the following GS types can exist with either of the above mentioned types:

Relay

A Relay GS acts more like a peer-to-peer implementation where all the work is being performed on the clients, and the GS is just used to send packets between a client and all other clients. This provides the best performance as actions can occur locally as soon as the player does something, and gets updated to other clients asynchronously. However, because of this it lends to hacking/cheating and can also provide unexplained results due to lag and other issues. (Ie: a player explodes but on the remote client nothing caused the explosion due to lag or packet drop, at the same time on a different client a bomb went off killing the player, and on a third client the player was away from the bomb but dies a short time later for no apparent reason, when the session state gets replicated)

These days with controlled client devices such as smart phones, ipads or consoles, we don't need to be as concerned with hacking the way we are on PC/MAC titles.

Authoritative

Authoritative GS is just the opposite of the relay server. In this model the client sends all commands to the GS and the GS decides if the command is acceptable and performs the appropriate action, such as walk forward, then updates all the clients with the new position and orientation of all objects. This is common in an MMORPG where the action is more turn based, but less than ideal for a real-time FPS which has fast paced action and expects near real-time response. All clients are in lock-step with each other, but due to lag of the GS, can feel slow and unnatural to the player. Some games use an Authoritative approach but add a local prediction so it feels more accurate. In this case a player requests to move. It sends the request to the

server but at the same time assumes the server will approve it so the local move is started. This way it feels responsive to the player but the GS still has the final say and will put the player back where he was if it doesn't approve the request.

Hybrid

What I recommend for most multi-player games is what I've coined as the Hybrid style. This incorporates the best of the relay and authoritative servers in a well balanced way providing the most flexible gaming experience. In Hybrid the player performs an action which occurs immediately then tells the other clients through the GS so they can refresh what the other players are doing appropriately. This is similar to the peer-to-peer. However, when it comes to things that affect other players, then each player is responsible for the decision of things it controls, which is similar to the Authoritative approach but instead of the GS being the authoritor, each client is the authoritor for the objects they control/manage. Some objects in the scene/level aren't tied to a player such as the trees, or power-up items. In this case the GS would be the authoritor for those items. Let's go through an example:

- Let's say we shoot a missile projectile. We start the animation immediately on our client so we experience no lag.
- We tell the other clients so they too can instantiate the projectile. We'll refer to this remote copy of the missile as the missile's AVATAR.
- As the missile travels on the originating client, the client is sending updates through the GS so each remote client knows the current position and orientation
- On the remote clients we use something called prediction and smoothing (covered later) to update the missiles avatar on the remote clients
- Now back on the original client the projectile hits another player. The appropriate response is to show the collision which we do "LOCALLY". Next we'd want to apply damage to the other player, but we DON'T do that. Since we don't control the other player that was hit we aren't responsible for applying damage, we're NOT authoritative for the hit player.
    - We could choose to either send a message (through the GS) to the player we hit and ask it to apply the damage, but this is a slightly more advanced topic for another ebook. Instead what we'll let happen is whenever a "**local**" player is hit by a remote projectile, that local player will be its own authoritor, and will apply damage to itself. Therefore in the above case the missile was local not the player being hit, so we just show the explosion for aesthetics and we're done. Only when a missile hits a local player or other local object will that client process damage.

- If you're paying attention and getting it so far you might realize the missile was just destroyed on the original client, so we're no longer sending/receiving updates. Therefore doesn't the missile avatar just stop on all the other clients?
- Well it would except we're still using prediction which will keep the avatar moving along its previous trajectory for a short period. The only extra thing we do on the avatar is manage a timer to self destruct the avatar after a certain period of time since we're no longer receiving updates from the original client.
- We could send a command from the client where the missile hit to all other clients instructing them to explode the missile, but unless we hit a local object we'd never hit anything official and never apply damage. Therefore, the only time we tell all other clients to explode the missile is when the hit object is a local object that takes damage. In that case we need to destruct the missile for everyone.

A few other more advanced approaches we can implement include:

- When the originating missile is destroyed we send a packet to all the remote missile avatars to take over their own navigation until they either hit something or their timer expires.
- Alternatively, from the time we instantiate the missile on the remote clients we can let each remote client handle its own local movement, never needing to send network updates as the missile travels. This is also a perfectly acceptable technique since a missile's trajectory is mathematical and very consistent. It's actually more efficient to do it this way. However, when it hits something on a client that controls what was hit and therefore applies damage to what was hit, we should send a command to all other clients instructing them to terminate the missile so we don't perform additional damage on a secondary client because we hit something there too, at a later point in time. This works great for a dumb fire missile, or laser, or gun. However, if it were a heat seeking missile with AI directing it, then I'd probably only run that on one client and let networkView.observed properly control the missile movement on the remote clients.
- Expect some of these to be covered in more detail in a future eBook.

There are many ways to consider implementing this and I'm just suggesting a few ways to do it. At the end of the day it's up to the developer and game designer to decide how they want it to work. There's no magic here. This is the tricky part of network programming and why it's so crucial to understand all then anomalies and how to properly design for them.

- If the missile avatar hits a local player on the client controlling the player, then that player applies damage to itself and decides if it was destroyed. If it was destroyed the client will send a destroyed message through the GS telling all the other clients it was destroyed.
- You could potentially have a situation where every client has the missile hit the player except the client where the player is local. In each of these instances the missile will explode locally, but no damage will be incurred and the player will go on unaffected. This is done for aesthetics only. We only process damage when the missile hits on the client where the player being hit is local.
- Someone has to be in charge of the decision making and I prefer each player to be in charge of his own character along with any other objects that character owns/controls. ***This is the crux of the Hybrid model.***
- In the case of an object that's neutral, such as a tree that can take damage, then the GS is the owner of all neutral objects and will be responsible for performing any damage incurred. This is a good reason to have a dedicated GS, so the extra processing for scene objects wouldn't affect a player client running on the same computer.

LAG:

A challenge many games experience is lag. Gaming Lag is when a client falls behind and doesn't keep up with other clients usually due to network latency and/or different speed client computers. Lag can result in players appearing to jump from place to place and not walk smoothly, among other issues. In some cases it's due to some players having slower computers than others. In other cases it's due to some players having slower networks that can't keep up with the amount of packets being sent. LAG can also be caused by a poor game networking design that send too many network packets creating an issue that impedes all clients from keeping up.

A way to limit the affect of lag is to implement what's known as prediction and smoothing. One reason games experience lag is because it takes time for a client to send a message to a GS then the GS to send the message to all the clients. By the time a client receives it, the player is no longer at the spot being reported. Consider a game running at 60 fps. The client screen is updating every 16 ms. If it takes 30 ms to deliver an update to a remote client, the local player has already processed through 2 additional frames. It might have stopped, sped up, or turned and the remote client doesn't know it yet.

1. **Prediction** is the act of moving the player where you think it'll be. If the client can calculate the lag is 30 ms, and the current frame rate is 60 fps, then it can be

deduced that if the remote player is moving in a certain direction and it keeps moving in the same direction then by the time the packet was received its really two frames further along. Prediction updates the location to reflect this predicated new location based on the previous location, direction and speed.

2. **Smoothing** is moving the player from its current position to the new position over the course of time before the next packet arrives. If you are able to estimate 30 ms between network updates on average, then you can calculate to move the remote avatar from point A to point B will need to take 2 frames. You therefore move the player half way in the first frame then the remaining distance in the next frame. This helps reduce seeing the remote avatar appear to JUMP due to bad lag.

Network Connectivity and Firewalls:

IP Addresses and Ports

Network IP Address

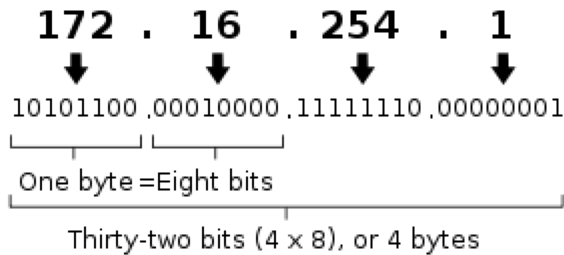
(http://en.wikipedia.org/wiki/IP_address)

An **Internet Protocol address** (**IP address**) is a numerical label assigned to each device (e.g., computer, printer) participating in a computer network that uses the Internet Protocol for communication.[1] An IP address serves two principal functions: host or network interface identification and location addressing. Its role has been characterized as follows: "*A name indicates what we seek. An address indicates where it is. A route indicates how to get there.*"

Two versions of the Internet Protocol (IP) are in use: IP Version 4 and IP Version 6. Each version defines an IP address differently. Because of its prevalence, the generic term *IP address* typically still refers to the addresses defined by IPv4. The gap in version sequence between IPv4 and IPv6 resulted from the assignment of number 5 to the experimental Internet Stream Protocol in 1979, which however was never referred to as IPv5.

IPv4 addresses

Main article: IPv4#Addressing



Decomposition of an IPv4 address from dot-decimal notation to its binary value.

In IPv4 an address consists of 32 bits which limits the address space to 4294967296 (2^{32}) possible unique addresses. IPv4 reserves some addresses for special purposes such as private networks (~18 million addresses) or multicast addresses (~270 million addresses).

IPv4 addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by dots, e.g., 172.16.254.1. Each part represents a group of 8 bits (octet) of the address. In some cases of technical writing, IPv4 addresses may be presented in various hexadecimal, octal, or binary representations.

Network Port

A network port is basically a unique number on a computer to separate network traffic across different applications. It's not uncommon to have several applications using the network at the same time, and in order for each application to separate the traffic from other applications a unique port is specified. When we get into the code we'll specify the ports as part of the network setup. It is possible you could run into a port conflict with another application, but we'll choose a port # that is rarely used, and since we don't normally run additional programs when we're also running a game it's not likely that you will run into a conflict.

Network Address Translation or NAT.

For most gamers, they're running the game on their computer or console from home and are therefore behind a router. Even smart phones and pads using 3G/4G service are behind a router and have a non-public IP address assigned to them. Most cable/internet providers provide a router which is connected to the outside line and all computers in the house need to connect to the router before they go out to the Internet. The connection might be a physical cable or a wireless connection, but in any case everything goes through that router. Since the normal gamer is using a residential internet service, they don't get a unique Public IP for every computer in the house. Instead all the computers in the house are on a private network called

a LAN. This way all the internal computers can share files, printers, music and other things in a secure way. When any computer in the house needs to connect to the Internet the request goes through the router. The computer's internal HW address is encapsulated in the packet, and the request is sent using the one Public IP that is on the router. To the world all the computers in the house seem to be the same because they all share a single IP to the world. When a response comes back to the computer, the router receives the packet and accesses the original internal HW address that was encapsulated in the original network packet. This allows the router to return the network packet to the correct computer. *This idea of encapsulating the HW address into the packet and using a single shared IP to the outside world is known as Network Address Translation or NAT.*

Hopefully that made sense to everyone. You might wonder why you need to know or care. Well the problem with a router is that it also blocks all incoming traffic if it didn't first originate from an internal computer (similar to a firewall). That makes it hard to run a home **gaming server** and allow remote connections. You see the GS can get out to the MGS, but once a client establishes itself as a GS we need all the remote clients to connect to it. To do that those remote clients need to tell the GS they want to connect, but because of the router or firewall that is between the GS and the outside Internet, the clients probably can't reach it. That's where NAT Punch Through can help and we'll discuss that next.

NAT Punch Through / NPT

NAT Punch Through (NPT) is a way to get around a firewall or router blocking incoming traffic; a default configuration for both. Most residential routers and firewalls allow outbound traffic but not inbound traffic except where it's a return from an original outbound request. In order to host a game you'll need other game clients to connect to your game server. Unless they're on the same home network they won't be able to reach you because of the router/firewall blocking things, and that's where NPT comes in. NPT redirects the original outbound connection from the GS that was used to connect to the MGS, and uses that as a connection source for new clients. This way the clients don't have to go directly to the GS, trying to open a new connection through the router or firewall. Since the connection from the GS to the MGS is already established, that same connection is used to connect all the Game Clients. When the connection is established between each client and the GS, the connection is a direct connection, and doesn't utilize the MGS for any additional work.

NPT doesn't always work because some commercial routers and firewalls will block NPT redirection. These more advanced systems see the new inbound data is from a different address (each game client) than the original connection to the MGS, and therefore block it. The only other way to make a GS accessible to the public is to either make sure its IP address is on a public network, or reconfigure the router or firewall (Not covered in this tutorial as every router/firewall is a little different) to allow the traffic through. To get a public IP usually requires going to your service provide and obtaining a public IP which you usually need to pay for. Then you could configure the computer that will host the GS with this address. Unless you're a professional gaming company who is going to host an MMOPG, this isn't usually done.

To configure the router to allow new inbound connections is called IP/Port Forwarding, and requires a skilled person to log in to the router and modify the router to allow traffic for a specific port (whatever port the game server is configured for) and configure the router to redirect any new incoming connections that aren't already associated to a computer to first allow the incoming request and second to redirect the request to a specific computer in the house that's already running the GS. This isn't hard to do, but you really don't want to require your gamers to have to deal with this. A lot of the time your gamers either aren't skilled in networking and would have no idea how to manually configure the router, or they might be kids and their parents control the router and won't let the kids touch it anyway. I won't go into additional detail here as every router is going to be a little different, and for most residential systems NPT works just fine. I have a pretty complex home network with several layers of routers, and NPT works fine for me. Most consoles such as an XBox 360 or PS3 use NPT as their primary way of supporting multi-player games, so I wouldn't worry much about IP/Port Forwarding.

For our game we'll be using a MGS to provide a list of available Game Servers we can connect to. Each client will be able to decide if it wants to join an existing game or host a new game. We'll automatically check if our GS is on a public or private IP and attempt to connect directly or using NPT automatically. If NPT is needed it'll automatically use it, otherwise it'll use a direct connect method.

General Coding Overview

Converting between C# and Java Script

I have a C/C++/C#/Java/Java-Script background (in that order) so I've decided to write all the sample code in C#. I know for many, Java-Script is used because it's considered the Unity3D Scripting language. However, I've also been told there are some things you can't do in Unity Java-Script that you can do in Unity C#. I'm not sure what they are, but I know for me I like C# better so I chose to do everything in C#. In either case both are converted to something internal Unity can work with, so neither is really using the native compilers anyway. If you prefer Java-Script over C# it's really not a big deal to convert my code, and each class is fairly small with just a few small methods in each. The basic things to think about when converting from C# to Java-Script includes:

1. RPC calls – "Remote Procedure Calls" allow one game client to talk to another (including the game server itself)
   - In C# use --- [RPC]
   - In Java-Script use --- @RPC
2. Variables Declarations
   - In C# use --- float this_is_a_float_variable;
   - In Java-Script use --- var this_is_a_float : float;
3. Methods in C# are Functions in Java
   - In C# use -- void some_method_name(…);
   - In Java-Script use – function some_method_name(…);
4. Foreach in C# use "FOR" in JavaScript
   - In C# use -- foreach (HostData gs in data)
   - In JavaScript use -- for(var gs:HostData in data)
5. CoRoutines
   - In C# use --- IEnumerator as the return value
   - In Java-Script it's automatic
6. Yielding in a CoRoutine
   - In C# use --- yield return 0;
   - In Java-Script use --- yield;
7. Required Components
   - In C# use --- [RequireComponent(typeof({component name}))]
   - In Java-Script use --- @script RequireComponent({component name})

Tags

Tags are a feature of the Unity3D editor that allow you to group certain objects together to make it easy to find them or for performing specific game logic based on the tag setting. If you choose any game object in the scene then look at the top of the Inspector Window you'll see two drop downs, one for Tag and one for Layer. Tags are tricky to define at first for people new to Unity3D so I thought I'd take a moment and explain it here. To add a Tag click on the TAG dropdown then choose "Add Tag". When you click it you see what looks like the "Layer" editor. However, the first thing shown is the word tag with a little arrow next to it. Clicking it drops down the tag and shows size = 0. Change size to something larger. I use 20 to start. This will give you 20 new elements for tags. Click to the right of the first one "Element 0" and enter "Projectile". That's it, we now have a "Projectile" tag that can be used to mark various objects.

Layers

Layers are used to optimize Rendering, Lighting and Collision and are a more advanced feature of Unity3D that we won't cover in this tutorial. In short you can use Layers to instruct the camera not to render certain layers, or lighting not to light certain layers and even use it to control collision detection so certain object might naturally pass through an object while others don't.

Overview of our Networking Tutorial:

Demo Overview

Let's take a few minutes now to think about what we're going to build. When I look at sample code there's nothing that frustrates me more than an overcomplicated sample that has way more functionality then I care about, making it really hard to follow the code and understand what's going on. I don't want to do that to you, so we're not going to use an existing demo. Instead we're going to develop a very simple sample from scratch as we go. I'll explain what we're doing each step of the way. It won't be pretty, but it's not meant to be. That'll be an exercise for you when we're all done, to make a real game that looks nice and uses what you've learned in this tutorial. For our needs we're just going to keep it simple.

What we'll build is a simple game with the following:

1. We'll create two levels
2. First level will be our Master Game Server Lobby Level
   - We'll have a Create Server Button
     - We'll have a text box to provide a name for the newly created game, which will be displayed for others to see as the name of an available game to join.
   - We'll have a list of available games and clicking on a game will join the game
3. The second level will be the in-game level
   - The in-game level will start with a flat floor to walk around on.
   - The in game level will have a box in the middle of the floor we can bump into
   - We'll create a couple pickups we can walk over to pickup
   - We'll create a couple spawn areas for character spawning and a couple spawn areas for pickup spawning
   - The player will be randomly instantiated when you start or join a game at one of the spawn areas
   - The player will be able to shoot a projectile at other players
   - We'll provide a hook for applying damage but we'll leave the actual game logic to you as an exercise

I think that should cover most of the basic elements necessary to give you an idea of how networking in unity works. From here you should be able to go on and explore on your own and hopefully build some pretty amazing games. Remember to let me know how you do; I'd love to hear from you at http://www.3dmuve.com.

Basic Networking API's

An Application Programming Interface (API) is a set of programming commands we can use to perform specific functions. When working with Unity3D they have many different API's for

different tasks, and we'll be focused on their Networking API's that are the specific set of Unity3D commands for performing networking requests. Lower level languages such as C/C++ use different API's that are much more complicated to setup and use, but provide a lot more control. Unity3D simplified using these lower level networking API's by creating their own wrappers around them and controlling some of the lower level requirement which we don't need to burden ourselves with.

TCP vs UDP:

Both are networking API's and TCP is probably the most popular due to its use in general home networking and surfing the web. TCP provides many advanced features for error detection and correction. It's what's referred to as a lossless protocol, because it guarantees never to lose or drop packets, and always keeps packets ordered. UDP on the other hand has no error checking which makes it a lot faster, but prone to errors, lost packets or packets arriving out of order. You might expect we'd use TCP for games since it provides error checking, but this checking makes TCP slow. Instead Unity3D and other Network Gaming API's create wrappers around UDP. These wrappers can add features such as error checking; while at the same time still improve performance over TCP because they are purpose built for our needs instead of general purpose. If we weren't using the Unity3D API's we'd have to learn much lower level API calls, which would take us a lot longer to master. Unity3D has made it easy for us.

Unity3D Networking API's

NetworkView

Before we get into coding in Unity I want to cover some basic fundamentals. First is the NetworkView Component in Unity, found under Component/Miscellaneous. NetworkView identifies an object as being able to send and receive network requests. These requests can be automated movement requests which are handled by the NetworkView component itself using the Observed property, and via RPC calls (covered later) that allow you to execute specific code for an object on a specific remote client (or all remote clients).

Let's consider for a second why NetworkView is so important. If we were using something other than Unity3D such as C/C++ and we didn't have a NetworkView component then how would we direct a request to a specific object on a specific remote client? We'd have to send a request to another client, and probably provide some sort of global identifier. Then that remote client would have to loop through all of its objects to see if it finds a matching global identifier and if it does, then it knows what object the network request was targeted for. This is a lot of work, and the beauty of using the NetworkView component is that this is done automatically for us.

Any object that needs to be synchronized over the network needs to have a NetworkView component added. Therefore any player characters will need a NetworkView and any vehicles they can get in and drive, or objects they can pickup/drop. This also includes weapons and projectiles. What doesn't usually need a NetworkView are things that add aesthetics to a scene but don't affect game play such as particles, a waving flag or banner, rippling water, trees, building, etc… Really anything that is static and doesn't move and can't be destroyed or animations that don't need to be in-sync across clients such as particles or a waving flag.

viewID

A critical parameter to NetworkView is **viewID**. viewID is similar to the previously discussed Global Identifier that is used to identify an object on a remote client. Although we don't normally have to loop through them to find the object we're looking for we still need to ensure the value is set correctly. If we instantiate a new object using **Network.Instantiate** then viewID is automatically set for us and for any given object it'll have the same viewID value on all remote clients. Alternatively, if we create an object manually using an **RPC** call along with a **local "Instantiate"** call instead of Network.Instantiate, then we need to generate a value for viewID ***just once*** using **"Network.AllocateViewID()"** and then pass that value in the RPC call to all other remote clients so they too can use the same value on their locally created avatars (copies).

Specific to NetworkView objects there are two ways to communicate changes between game clients.

Observed

The first way is the default "Observed" property of the NetworkView, which is usually the object the NetworkView is added to. This works in conjunction with the State Syncronization property which tells the NetworkView if it should send updates for position and orientation about the "Observed" object or not, and if so if it should be sent as reliable or unreliable. Reliable ensures it's delivered and in order whereas unreliable says you can lose a packet and it's no big deal because the next packet replaces this packet anyway. For example a death packet must get through and not be dropped, but a position update can be less accurate as it'll just update with the newer position the next refresh cycle. Observed update packets are sent and processed automatically and by default you don't need to do anything with them.

Remote Procedure Calls (RPC)

The other way to communicate updates is to use a Remote Procedure Call (RPC) which is a way to send specific commands to specific players (or all players) and allows you to control what detailed information gets sent. We'll use RPC's to do things such as instantiating a new player or weapons fire, transmitting death or damage if the player needs to be rendered differently

when it takes damage. We could also use an RPC to signify something was picked up or dropped, a door opened, or other triggered events.

In our game the following objects need a networkView. I don't think I forgot to tell you when to add each networkView but incase I missed any, make sure these objects are set correctly:

1. MasterServerMenu Object in the MasterGameServerLobby Level
2. TestGameLevel Level objects:
   - PickupSpawn Game Object
   - BallPickup Prefab
   - Player Prefab
   - Spear Prefab

Instantiating a Networked Object

To instantiate a new object we could use ***Network.Instantiate***, but I don't like it under all circumstances because we can't provide any details other than the type of object, location and orientation. Most of the time we'll have other information such as parent object, object name, maybe something about the health, current inventory, current team or allegiance, etc... Therefore we'll instantiate everything using an RPC call so we can control things more completely.

We'll be developing a hybrid GS/peer-peer model as described above. We'll have some objects that are part of the scene and controlled by the GS, then each player will control itself.

Building our Game:

Let's start by creating a new project using File/New Project. When you create the new project choose "Standard Assets (Mobile)". That's just so we have some textures we can use to fill our scene. I use the 2x3 screen layout, choose-able in the upper right corner of the screen.

So now you have an empty scene. First things first, let's name the scene. Call it "TestGameLevel". Use "File/Save Scene as" to save the level.

Building the Game Level:

Let's create a new plane that will be our floor:

Creating the Floor

1. Use "Game Component/Create Other/Plane".
2. Now rename the Game Component to "Floor"
3. Now with the Floor chosen in Hierarchy view go to the Inspector and change the scale to (1000,0,1000). Leave the position and rotation set to all zeros.

4. Now expand the "Standard Assets (Mobile)" Folder under the Project Window then expand the sub folder "Textures"
5. Now drag the "Grid" texture and drop it on the floor plane so we can better see the floor

Great, we now have a floor. Ok, next let's drop a BOX into the center of the floor:

Creating the Scenery:

1. Use "Game Component/Create Other/Cube"
2. You can rename it if you want but leaving it Cube works fine
3. With the cube selected in the hierarchy view change the scale in the Inspector view to (300,300,300)
4. Position the cube at (0,150,0). This will put it perfectly on the floor, in the center.
5. Now expand the "Standard Assets (Mobile)" Folder under the project area then expand the sub folder "Textures"
6. Now drag the "JoystickThumb" texture and drop it on the cube so we can better see it. Really any texture can be used, but I've just picked a couple included with the engine by default to make things easy
7. Make sure the Box Collider on the Cube is positioned at (0,0,0) and sized at (1,1,1)

Creating the Player Character

Now we'll create a player character very easily:

1. First lets choose "Game Object/Create Empty" then rename the new Game Object to "Player"
2. Now put the "Player" at (0, 30,-300), rotation at (0,0,0) and scale at (1,1,1). This sets the player to be next to and looking at the cube
3. Lets create a player body simply by choosing "Game Object/Create Other/Capsule"
4. Drag the "Capsule" into the Player
5. Rename the "Capsule" to "PlayerBody"
6. Make sure the position for the "PlayerBody" is set to (0,0,0)
7. Set the PlayerBody scale to (15,30,15) and rotation to (0,0,0)
8. Change the "PlayerBody" Capsule Collider to radius=1 and height=2. This will properly cover our player
9. Now let's add a rigid body to the player. This is so we'll properly collide with the box and other objects in the scene, and not pass right through them, and properly slide around them.
   a) First choose the "Player" in the hierarchy, not the "PlayerBody"
   b) Next Choose "Game Component/Physics/RigidBody", to enable physics on our player so it correctly reacts to gravity and collision with other objects.

c) The “Player” is not as big or heavy as the box so change the following properties in the RigidBody:
    - Mass = 10
    - Drag = 5
    - Angular Drag = 2
    - Under the constraints dropdown change “Freeze Position” and activate the “Y” and change “Freeze Rotation” and activate X and Z so when we do collide we remain solidly on the ground and don’t twist and fall or float away.

10. Now drag the default “Main Camera” into the player object
11. Change the Main Camera to be positioned at (0,150,-250) this places it slightly above and behind our player.
12. Now with the Camera selected choose the “ROTATE” widget in the upper left hand corner and slightly rotate the camera down so it’s looking towards the player but in front of the player. For me the rotation is at (7.436944, 1.77822, 0)
13. Now we need to add some special networking code to the player. Since the player object will be moving and we’ll need to track and coordinate those moves over the network, we’ll need to add a network component to the player. This is the ***networkView***, and any object needing to be synchronized over the network needs its own networkView. Also, the networkView.viewID is a unique value for an object that is consistent for the object on all clients. This is how send/receive commands know what object they’re targeted at. If we’re on the client where we have a local player moving and we want to update the remote clients with the movement, the networkView component does the work for us, and it uses viewID to know which remote player avatar on each remote client should be updated.
    a. To add the networkView component choose “Component / Miscellaneous / NetworkView”

Player Controller:

14. We’re almost done with our player but it can’t move yet; we haven’t added player controls. We could use one of the available player control scripts but I prefer to write my own and keep things really simple. Let’s do that now:
    a. Right click in a blank area in the “Project” Window and choose “Create/Folder”
    b. Name the Folder NetworkSampleCode
    c. Now right click on the new folder and choose “Create/C# Script”. You could use Java Script but I prefer C#. If you choose Java Script you’ll need to change a few things such as variable declaration and using @rpc instead of [rpc].
    See here for more details on converting between C# and Java Script.
    d. Name the new script “PlayerController”
    e. Double click on the new script to open the Unity3D MonoDevelop Editor/IDE

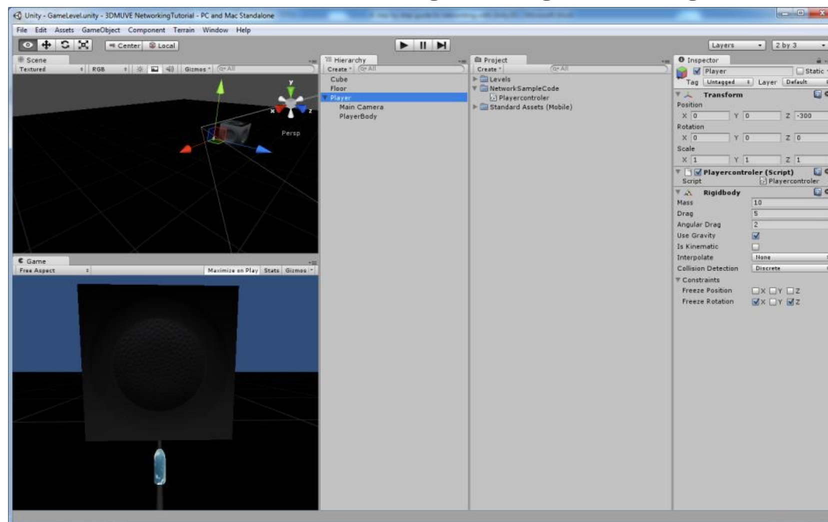
f. Now for our player control here's the code. The code is fairly well commented so hopefully it'll make sense to you. This is just a very simple controller script to read WSAD key presses and update the player accordingly based on the key presses:

Player Controller Code:

```
using UnityEngine;
using System.Collections;

public class PlayerController : MonoBehaviour {

// Use this for initialization
void Start () {

}

// Update is called once per frame
void Update () { // Process the keyboard input for moving/turning every frame
ProcessForwardMove();
ProcessTurnMove();
}

void ProcessForwardMove()
{
// When moving Forward or Backward we'll move along the Z axis, so we TRANSLATE
// along X, Y and Z with X and Y = 0 and Z the amount we want to move.
// In this sample we want to move 100 units a second, so we multiply that by Time.deltaTime
// to get how much we move in the frame.
// Ideally instead of hardcoding the 100, we should make it a variable but I'm trying to keep
// things simple.
// In unity positive Z moves forward, so to move backward we just use a -Z value

if ( Input.GetKey(KeyCode.W) ) // Press or Hold W to move forward
{
transform.Translate(0,0, 100 * Time.deltaTime);
}

if ( Input.GetKey(KeyCode.S) ) // Press or Hold S to move backward
{
transform.Translate(0,0, -100 * Time.deltaTime);
}
}

void ProcessTurnMove()
{
// When turning Left and Right we'll rotate along the Y axis (up/down axis), so we ROTATE
// along X, Y and Z with X and Z = 0 and Y the amount we want to turn.
// In this sample we want to move turn units a second, so we multiply that by Time.deltaTime
// to get how much we turn in the frame.
// Ideally instead of hardcoding the 100, we should make it a variable but I'm trying to keep
// things simple.
```

```
// A left turn is a negative rotation and a right turn is a positive rotation

if ( Input.GetKey(KeyCode.A) ) // Press or Hold A to turn Left
{
transform.Rotate(0, -100 * Time.deltaTime, 0);
}

if ( Input.GetKey(KeyCode.D) ) // Press or Holda D to turn right
{
transform.Rotate(0, 100 * Time.deltaTime, 0);
}
}
}
```

g. Hopefully that made sense and wasn't too difficult. Now save it, clicking save in the upper left corner then return to the Unity3D Level Builder.
h. Now in the hierarchy click on the "Player"
i. Now Drag the PlayerController script into the Inspector for the "Player" at an open area at the bottom
j. If no errors we should have things looking something like this



k. Let's go ahead and test the game so far. First choose File/Save Project and File/Save Scene
l. Now click the Black arrow at the top center of the screen that looks like a VCR or tape recorder play button (I guess I'm showing my age :J)
m. Using WSAD you should be able to walk around the scene. Try walking into the block and you should smoothly slide along the side of it until you get around it.

Player Spawn Area:

Now we'll create the player spawn area. For this and the pickup spawn area, I'm not going to do a lot of explaining because this is all general Unity3D level creation steps. I'll take time to explain the details when we get into more of the network code specifics.

1. We'll create 2 Player Spawn areas:
    a. Choose Game Object / Create Empty
    b. Rename the Game Object to "PlayerSpawn"
    c. Position "PlayerSpawn" at (0,0,0)
    d. Choose Game Object / Create Empty
    e. Rename the Game Object to "PlayerSpawn1"
    f. Position "PlayerSpawn1" at (500,30,500)
    g. Repeat steps 1-3 but call this "PlayerSpawn2"
    h. Position "PlayerSpawn2" at (-500,30,-500)
    i. Now Drag both PlayerSpawn1 and PlayerSpawn2 into PlayerSpawn

Pickup Spawn Area:

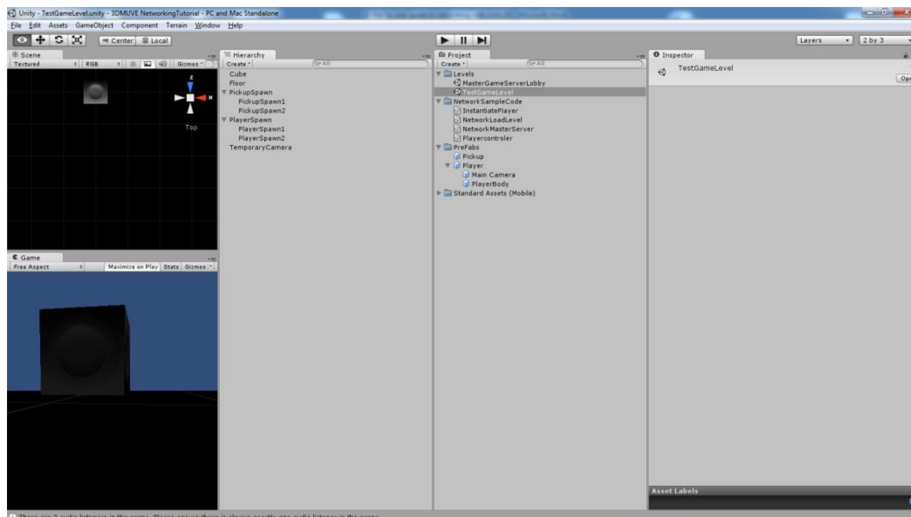
Now we'll create the pickup spawn area.

1. We'll create 2 Pickup Spawn areas:
    a. Choose Game Object / Create Empty
    b. Rename the Game Object to "PickupSpawn"
    c. Position "PickupSpawn" at (0,0,0)
    d. Choose Game Object / Create Empty
    e. Rename the Game Object to "PickupSpawn1"
    f. Position "PickupSpawn1" at (500,15,-500)
    g. Repeat steps 1-3 but call this "PickupSpawn2"
    h. Position "PickupSpawn2" at (-500,15,500)
    i. Now Drag both PickupSpawn1 and PickupSpawn2 into PickupSpawn
2. Now add a networkView component to the PickupSpawn using "Component / Miscellaneous / Network View"
3. Now we'll need a prefab to spawn and pickup
    a. Create a new sphere using "Game Object / Create Other / Sphere"
    b. Rename the object to BallPickup
    c. Position the object at (0,0,0)
    d. Change the scale to (10,10,10)
    e. Now go to the Project Window, right click on an open area and choose "Create / Folder".
    f. Call the Folder "PreFabs"

g. Now Drag the “BallPickup” Sphere we just created from the hierarchy view and drop it onto the “PreFabs” Folder
h. Now right click on the “BallPickup” in the hierarchy window and choose “Delete”

We’re just about ready to start adding the networking code. One last thing, let’s move the “Player” object to a PreFab and delete the “Player” from the scene hierarchy. Doing this will make the level non playable as the camera is tied to the player so we won’t have a functioning level anymore. However, since the camera is associated to the player we’ll be fine once the player is instantiated into the scene. We could choose to have a temporary camera to show a part of the scene and when the player is instantiated just disable the temporary camera and enable the player camera, but for this tutorial we’ll keep it simple and just have the one player camera.

Ok you’re doing great. You’re current level should now look something like this:



At this point let’s save this level. It should already be called TestGameLevel. If you didn’t already save it in the beginning make sure to save it now and call it “TestGameLevel”

Overview of Networking Classes we'll create for our Demo

Ok so now we want to get started with networking so first we'll create a new scene and this scene will function as our Master Game Server (MGS) Lobby, where we'll pick a game from the list or start a new game. Choose "File / New Scene". Now let's save this scene as "MasterGameServerLobby", using "File / Save Scene as".

NOTE:* Before I forget, let's make sure we add our two scenes to our project, so with "MasterGameServerLobby" opened (double click on the scene in the Project Window***) go to "File / Build Settings" and choose "Add Current". Ok now double click back on "TestGameLevel" to reopen it and add it at the next (second) level in our project using "File / Build Settings" and choose "Add Current". Great, now we can reload "MasterGameServerLobby", so do that now by double clicking on it in the Project Window.***

Next we're going to create some scripts and associate those scripts to the level. Below is the code fully commented so hopefully you'll be able to understand what's going on. Before we get to the code let me explain the various modules:

1. **NetworkMasterServer** – The NetworkMasterServer script does a few things but the main things it does is to show the MGS Lobby Menu and perform the selected actions. It also has the test connection code which is used to determine if NAT Punch Through (NPT) is required and if so pass the appropriate parameters to make it happen

2. **NetworkLoadLevel** – As its name indicates, this is where we load the level, but we do so properly for a networked level, so we properly manage network state, then once the level is loaded we also send a message to spawn all our players into the new level. Consider a player joins a game and there are already 5 other players in the game. It's not enough to spawn just the new player, we need to also instantiate the other 4 players appropriately into our scene, and any other networked objects such as vehicles or projectiles, etc…

3. **InstantiatePlayer** – Here we actually take care of instantiating the player and cleaning up the player when a remote player disconnects

4. **NetworkRigidBody** – This handles the prediction and smoothing for an object that is a remote avatar. We'll receive periodic network updates on the objects current position and orientation, and we'll use NetworkRigidBody to smooth the transition from the current location to the new.

5. **PlayerSpawn** – When the InstantiatePlayer script instantiates a player we need to do several things. In short, if the player being instantiated is the local player then we need to ensure the camera is configured as enabled and if we're first person we might want to make sure things like a HUD are properly enabled, and the player model is not enabled. We also need to make sure player control is enabled so we can control the player.

   However, if the player being instantiated is a remote avatar then we need to make sure the camera is disabled since we only want the local player managing the camera. We also need to make sure there's no HUD, but we DO want to render the player model. We also need to make sure player controls are disabled for this remote avatar so we don't accidently control it when we're controlling our own local player. Finally we'll want to enable prediction and smoothing for the network updates we do receive.

6. **BallSpawnManager** – This controls spawning a group of balls that can be picked up, carried and thrown at other players.

7. **BallController** – This is associated with the local player, and monitors the player colliding with a ball to pick it up or pressing the fire button to throw a ball if the player is carrying one. Only a local player can throw a ball, not a remote avatar. We use an RPC to appropriately update all the remote avatars, so all clients see the ball being thrown. To throw the ball we apply a force to the physics component of the ball and let the physics engine do the rest. We only apply the force on the original ball, not the remote ball avatars, and we let the NetworkRigidBody script properly update the ball movement on the other clients.

   We could make an argument to let each client manage the ball navigation on its own and handle the collision on its own, but still only apply damage based on the owning client of the object that was hit. This would reduce network lag which can be more obvious with a fast moving object such as a missile and the results to the end user probably wouldn't be noticeable unless they have their monitors right next to each other. Since our ball doesn't move that fast we'll rely on NetworkRigidBody to update the ball on the other clients, primarily so you can learn the technique.

   - *If you choose to let each client update the ball then you'll need to disable NetworkView:Observed and NetworkView:State_Synchronization, and also disable NetworkRigidBody which we currently toggle on/off in code so you'll need to search out all reference to it pertaining to the ball and remove it. Finally instead of applying*

force just on the local player throwing the ball, you'll need to apply the force on all clients.

8. **BallManager** – The Ball Manager is responsible for detecting ball collision with a player when the ball is thrown, to instill proper damage to the hit object. It also takes care of respawning a ball after it's thrown and various timers to manage how long to wait after a respawn before it's okay to pick up a ball and an auto destruct/respawn on a ball after it's thrown but doesn't hit another player.

Analysis and Code of each Network Class

NetworkMasterServer:

There are a few important commands to understand in the NetworkMasterServer:

Network.InitializeServer is the main command to use when you want your client to become a Game Server (GS). When you create a server you specify the number of allowed client connections, the network port to listen on for new connections, and whether NPT is permitted.

```
Network.InitializeServer(
    number_of_allowed_clients,
    serverPort,
    allow_NAT_Punch_Through );
```

MasterServer.RegisterHost is used after you create yourself as a GS, then the register host goes back to the Master Game Server to tell it you're now a GS. This way your GS will show up as an available choice to other clients.

```
MasterServer.RegisterHost(
    gameType,
    gameName,
    gameDescription);
```

Network.Connect is used on a non-GS client when that client has selected a GS from the available list and decides to join the game. If the GS selected has useNat, then you should connect using the guid of the GS. If the GS doesn't have useNat then you should attempt to connect directly using the GS's IP address and the GS's listen port.

```
// Direct connect to a GS
    Network.Connect( gs.ip, gs.port);

// Use NPT to connect to the GS
    Network.Connect( gs.guid);
```

ClearHostList will clear the local array of available game hosts. It's a good idea to periodically refresh the list so you aren't dealing with stale data

MasterServer.ClearHostList();

RequestHostList is used to pull a list of available Game Servers from the MGS. You specify the game type and retrieve GS's for just that type

MasterServer.RequestHostList (gameType);

PollHostList is used to extract the list of available game servers into a local array for processing. Use this to get the list and display the screen display list, allowing any to be chosen. I prefer to show the Game Name, Game Description and Active vs Maximum allowed players. You can choose to get more advanced and also show ping or latency information and sort lowest to highest so a gamer can choose a GS closest to him to minimize LAG.

MasterServer.PollHostList

Create a new c# file under your NetworkSampleCode folder and call it NetworkMasterServer.

NetworkMasterServer code

Here's a full sample of the code in use. Save it into the NetworkMasterServer code file you just created:

```
// You can just copy and paste this entire function and use it as is.
// However, some of the GUI menu items will need to be customized with the proper game name such as:
// gameType and the string: "Your Game Title Here"

using UnityEngine;

public class NetworkMasterServer : MonoBehaviour {

// Keep track of various menu and ingame states
public enum menustate
{
networklobby,     // Master Game Server Lobby
ingame,           // In a game as either the GS or Client
}
public menustate gamemenustate = menustate.networklobby;

// NOTE: gameType is used to distinguish your game from any other game that might
//          also be registered to the MGS. I also like to add a version number as part of the
//          game type. This way as you release new version, the client will have to match the
//          same version in order to connect
public string gameType = "SampleUnityNetworkingDemo_v1.0";
```

```
// NOTE: gameName is an arbitrary name that will show in the game lobby so other players
//          find and distinguish your game from another.  We also add a text box so the
//          game client can customize it before creating a new game
private string gameName = "LetsPlay";
public int serverPort = 25002;  // The port must be unique to the GS and not conflict with other
// apps running on the game server.  This is how clients will
// connect to the GS

// NOTE: The rest of the variables below are various attributes used to automatically refresh
//          the discovered game servers, and to keep track of the testing and results when
//          checking to see if NPT is required
private float lastHostListRequest = -1000.0f;
private float hostListRefreshTimeout = 10.0f;
private ConnectionTesterStatus natCapable = ConnectionTesterStatus.Undetermined;
private bool filterNATHosts = false;
private bool probingPublicIP = false;
private bool doneTesting = false;
private float timer = 0.0f;
private string testMessage = "Testing NAT capabilities";
public GUIStyle format = new GUIStyle();
private bool useNat = false;


// Enable this if not running a client on the server machine
// MasterServer.dedicatedServer = true;
void OnFailedToConnectToMasterServer(NetworkConnectionError info) {
//Debug.Log(info);
}

void OnFailedToConnect(NetworkConnectionError info) {
//Debug.Log(info);
}

void OnGUI () {
ShowGUI();
}

void Awake () {
DontDestroyOnLoad(this);

// Start connection test
natCapable = Network.TestConnection();

// What kind of IP does this machine have? TestConnection also indicates
// this in the test results
//if (Network.HavePublicAddress())
//          Debug.Log("This machine has a public IP address");
//else
//          Debug.Log("This machine has a private IP address");

// The game can have several menus and states, so I like to use
// an enum to keep track of what state we're in.  In this case we're
```

```
// in the MGS lobby
gamemenustate = menustate.networklobby;
}

void Update() {
// If test is undetermined, keep running
if (!doneTesting) {
TestConnection(); // Testing is to evaluate the IP's of the client and server to determine if
// we need NPT to connect
}

if (Time.realtimeSinceStartup > lastHostListRequest + hostListRefreshTimeout)
{
MasterServer.ClearHostList(); // Clear the current local list of GS's prior to refreshing it
MasterServer.RequestHostList (gameType); // Get an update list of available GS's from
// the MGS
lastHostListRequest = Time.realtimeSinceStartup;
//Debug.Log("Refresh Available GS List");
}
}


// Test Connection is used to check the connection to the Game Server (GS) to determine if it's behind a
// router/FW with a different public IP and private IP. This helps us determine if we need to try using NAT
// Punch Through to connect Network.useNat will be set based on the tested findings. This is used on the
// client to figure out how to connect On the GS we check !Network.HavePublicAddress() and pass it into
// the MGS when we register so it know if NPT is required
void TestConnection() {
// Start/Poll the connection test, report the results in a label and react to the results accordingly
natCapable = Network.TestConnection();

switch (natCapable)
{
case ConnectionTesterStatus.Error:
testMessage = "Problem determining NAT capabilities";
doneTesting = true;
break;
case ConnectionTesterStatus.Undetermined:
testMessage = "Testing NAT capabilities";
doneTesting = false;
break;
case ConnectionTesterStatus.PublicIPIsConnectable:
testMessage = "Directly connectable public IP address.";
useNat = false;
doneTesting = true;
break;

// This case is a bit special as we now need to check if we can
// use the blocking by using NAT punchthrough
case ConnectionTesterStatus.PublicIPPortBlocked:
testMessage = "Non-connectble public IP address (port " +
serverPort +" blocked)," +
" running a server is impossible.";
```

```
useNat = false;
// If no NAT punchthrough test has been performed on this public IP, force a
// test
if (!probingPublicIP)
{
//Debug.Log("Testing if firewall can be circumvented");
natCapable = Network.TestConnectionNAT();
probingPublicIP = true;
timer = Time.time + 10;
}
// NAT punchthrough test was performed but we still get blocked
else if (Time.time > timer)
{
probingPublicIP = false; // reset
useNat = true;
doneTesting = true;
}
break;
case ConnectionTesterStatus.PublicIPNoServerStarted:
testMessage = "Public IP address but server not initialized," +
"it must be started to check server accessibility.” +
“ Restart connection test when ready.";
doneTesting = true;
break;
default:
testMessage = "Error in test routine, got " + natCapable;
if ( string.Compare("Limited",0,natCapable.ToString(),0,7) == 0 )
useNat = true;
doneTesting = true;
break;
} // end switch
//Debug.Log(testMessage);
}

// Here we paint the menu screen so the player can choose to join a game or start a new game
void ShowGUI()
{
// in the AWAKE method we used: DontDestroyOnLoad(this);
// so when we load our game level this ShowGUI is still running
// so here we can see what menu we should be showing and display the appropriate menu

// When we're ingame we want to show a Disconnect button to quit the game
// and return to the lobby
if ( gamemenustate == menustate.ingame )
{
if (GUI.Button (new Rect(10,10,90,30),"Disconnect"))
{
Network.Disconnect(); // Tell all the other clients you're disconnecting
MasterServer.UnregisterHost();
gamemenustate = menustate.networklobby;
// Return to the Master Game Server Lobby
// because we pressed disconnect
Application.LoadLevel("MasterGameServerLobby");
```

```
}
}
else  // ensure we're in the lobby, and not off somewhere unexpected
if ( gamemenustate  == menustate.networklobby  )
{
if (Network.peerType  == NetworkPeerType.Disconnected)
{
format.fontSize  = 28;
GUI.Label(new  Rect(((Screen.width/2)-80)   * 0.2f,
(Screen.height/2)-200,
400,
50),
"Your Game Title Here",
format);

gameName = GUI.TextField(new  Rect((((Screen.width/2))   * 0.2f)+200,
(Screen.height/2)-100,
200,
30),
gameName);
// Start a new server
if (GUI.Button(new  Rect(((Screen.width/2)-100)   * 0.2f,
(Screen.height/2)-100,
200,
30),
"Start Server"))
{
// If start server is chosen then we want to initialize  ourselves as a
// server and then register with the master server so other players will
// see us as a choice in the list of game servers
// We'll use Network.InitializeServer  to enable our Game Server (GS)
// functionality  but first we need to determine if we'll require clients to
// connect to us directly or by using NPT.

// The first two parameters to InitializeServer  are how many clients can
// connect and the port to use The port needs to be unique, but on
// desktop/laptop  pc's it's not usually an issue as there aren't a
// lot of network listeners running to contend
// Once the game starts you can consider using
// Network.maxConnections  to stop any new players from connecting

if ( doneTesting  ) // If done testing use the more thorough  results of
// the testing to setup the server
Network.InitializeServer(32,  serverPort,  useNat );
else   // otherwise setup the server and specific NPT based on
// public/private  IP this is not as accurate as the full test, but is
// sufficient for most needs
Network.InitializeServer(  32,
serverPort,
!Network.HavePublicAddress());
MasterServer.updateRate   = 3;

MasterServer.RegisterHost(  gameType,
```

```
gameName,
"This is early network testing for my game");
}


// Extract the list of available GS's into a local variable array for processing
HostData[] data = MasterServer.PollHostList();
int _cnt = 0;

// Loop through all the available GS's provided by the MGS and
// display each one so we can choose one to joing
foreach (HostData gs in data)
{
// Do not display NAT enabled games if
// we cannot do NAT punchthrough
if ( !(filterNATHosts && gs.useNat) )
{
// Build a name string to use for displaying the GS in the list
string name=gs.gameName + "-" + gs.comment + " (" +
gs.connectedPlayers + " / " + gs.playerLimit +")";

if (GUI.Button(new Rect(((Screen.width/2)-100) * 0.2f,
(Screen.height/2)+(50*_cnt),
600,
30),
name ))
{
// Enable NAT functionality based on GS host
// we're connecting to
useNat = gs.useNat;
if (useNat)
{
print("Using Nat punchthrough to“ +
“ connect");
Network.Connect(gs.guid);
}
else
{
print("Connecting directly to host");
Network.Connect(gs.ip, gs.port);
}
}
}
}
}
}
}
}
```

Now that you've created a new script let's go back to our **MasterGameServerLobby** Level, which should just have a single default camera in it. Now create a new empty game object

using "**GameObject/CreateEmpty**" and then rename the new **GameObject** to "**MasterServerMenu**".

Now let's add a **NetworkView** component to the **MasterServerMenu**. This will enable networking for us. To do so choose "**Component/Miscellaneous/Network View**".

Good, now we want to also add the **NetworkMasterServer** script to the same "**MasterServerMenu**" we just created, so with **MasterServerMenu** selected in the "**hierarchy**", drag and drop the **NetworkMasterServer** script from the Project Window to a blank spot in the "**Inspector**" window of the **MasterServerMenu**.

If we now run this level we'll see the menu selections, but we can't do much with it beyond that yet.

NetworkLoadLevel:

NetworkLoadLevel is where we perform the actual level load, along with maintaining proper state within the client and appropriately spawning our player once the level is loaded.

Let's add "NetworkLoadLevel" to our scene. So let's create a new class by right clicking on the NetworkSampleCode in the Project area and choosing "Create / C# script". Name the new script "NetworkLoadLevel".

Now let's associate the new script to the scene by choosing "MasterServerMenu" in the hierarchy window for the "MasterGameServerLobby" level, and drag our NetworkLoadLevel script into an empty section in the Inspector Window.

Now we should see three components associated to the "MasterServerMenu"

1. NetworkView
2. NetworkMasterServer
3. NetworkLoadLevel

The commands that are important to understand here include:

networkView.RPC*("LoadLevel", RPCMode.AllBuffered, "test", lastLevelPrefix + 1);*

The RPC command sends a request to run the "LoadLevel" method (or function in java), which as its name indicates will load the level requested. As previously discussed an RPC is used to send commands to remote players or just the game server. Those remote commands then perform a local action, but in general it's being performed equally on all clients.

The next parameter "RPCMode.AllBuffered" does two things. First the "All" indicates all clients receive this request, including the local client sending the command. Therefore, we don't also load the level manually. Instead we'll receive the network request telling us to load the level and load it just like everyone else. The other part of the command is the "Buffered" which tells the Game Server to hold onto the command and rerun the command for any new clients that connect. This is useful for initializing new objects such as loading a level or creating a new object.

The remaining parameters are sent to the "LoadLevel" method and will reflect whatever parameters that method needs. In this case it's the name of the level to load and a level prefix.

LoadLevel

For good measure we disable sending network packets and message queuing while we load the new level. This is to prevent picking up objects from the old level that we're trying to get rid of

as part of loading the new level. Once the level is successfully loaded we'll reactivate both sending of network packets and message queuing. Note: We use "yield" to just give a little time to ensure the level completed loading successfully before we active the networking again.

The level prefix is used to set the network level prefix using **Network.SetLevelPrefix(levelPrefix);** This ensures that we don't receive ghosts from another level and objects are properly filtered so we only see what belongs in the current level.

The last important thing we do in NetworkLoadLevel is send a command to the level's Player Spawn area telling it to instantiate our new player. Now let's think about the flow for a minute.

1. The RPC to load the level is called once by the Game Server to load the first level
2. This RPC to load the level is stored as a Buffered request so all new clients connecting will also receive a "LoadLevel" event.
3. As each client loads its level it then needs to load its local player character.
   a. We accomplish this by using **SendMessage("InstantiatePlayerOnNetworkLoadedLevel", SendMessageOptions.RequireReceiver);** which sends a local message to the InstantiatePlayer class which is attached to the PlayerSpawn area. We haven't actually setup the InstantiatePlayer script to the spawn object yet, so we'll explain this in more detail shortly.
4. The "InstantiatePlayerOnNetworkLoadedLevel" method performs the player instantiate by calling Network.Instantiate which will instantiate this player on all clients.
   a. Since each client only loads its level once we only call the local "InstantiatePlayerOnNetworkLoadedLevel" once for each client, which is what creates the local player on that client
   b. Since the player is created with a Network.Instantiate it'll be buffered on the GS and then also called on all other clients to instantiate the remote player avatars.

NetworkLoadLevel code

Here's a full sample of the code in use. Save it into the NetworkLoadLevel code file you just created:

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(NetworkView))]
public class NetworkLoadLevel : MonoBehaviour {


// Keep track of the last level prefix (increment each time a new level loads)
private int lastLevelPrefix = 0;
```

```
private NetworkMasterServer mgs;

void Awake () {
// Network level loading is done in a seperate channel.
DontDestroyOnLoad(this);
networkView.group = 1;
//Application.LoadLevel("EmptyScene");
mgs = GameObject.Find("MasterServerMenu").GetComponent<NetworkMasterServer>()
as NetworkMasterServer;
}

//void OnGUI () {
void Update() {
// When the server is started then issue a "LoadLevel" RPC call for the local GS client
// to load his own level then also buffer the LoadLevel request so any new client that
// connects will also receive the "LoadLevel"
if (Network.peerType != NetworkPeerType.Disconnected &&
mgs.gamemenustate == NetworkMasterServer.menustate.networklobby &&
Network.isServer )
{
// Make sure no old RPC calls are buffered and then send load level command
Network.RemoveRPCsInGroup(0);
Network.RemoveRPCsInGroup(1);
// Load level with incremented level prefix (for view IDs)
// NOTE: The RPCMode.AllBuffered sends the request to all clients,
// including the client sending the request. Also, the Buffered command
// tells the game server to hold onto the RPC command and also
//      send it to any new clients that connect. This is important for level
// loading or player instantiation so we make sure all objects are correctly
// created on all clients, but would not be used for example to indicate a
// death as the new client would never have seen the player in the first place and
//      therefore doesn't need to see the death.
networkView.RPC( "LoadLevel",
RPCMode.AllBuffered,
"TestGameLevel",
lastLevelPrefix + 1);
}

}

[RPC]
IEnumerator LoadLevel (string level, int levelPrefix)
{
// If we already loaded the level don't load it again
if ( mgs.gamemenustate != NetworkMasterServer.menustate.ingame )
{
mgs.gamemenustate = NetworkMasterServer.menustate.ingame;
//Debug.Log("Loading level " + level + " with prefix " + levelPrefix);
lastLevelPrefix = levelPrefix;
// There is no reason to send any more data over the network on the default channel,
// because we are about to load the level, because all those objects will get deleted
// anyway
Network.SetSendingEnabled(0, false);
```

```
// We need to stop receiving because first the level must be loaded.
// Once the level is loaded, RPC's and other state update attached to
// objects in the level are allowed to fire
Network.isMessageQueueRunning = false;

// All network views loaded from a level will get a prefix into their NetworkViewID.
// This will prevent old updates from clients leaking into a newly created scene.
Network.SetLevelPrefix(levelPrefix);
Application.LoadLevel(level);
yield return 0; //new WaitForSeconds (1);
yield return 0; //new WaitForSeconds (1);

// Allow receiving data again
Network.isMessageQueueRunning = true;
// Now the level has been loaded and we can start sending out data
Network.SetSendingEnabled(0, true);

// Once the level is loaded we then need to instantiate our local player. To do that we
// could use Network.Instantiate, but that only lets us send
GameObject go = GameObject.Find("PlayerSpawn");
InstantiatePlayer io = (InstantiatePlayer)go.GetComponent(typeof(InstantiatePlayer));
io.SendMessage("InstantiatePlayerOnNetworkLoadedLevel",
SendMessageOptions.RequireReceiver);
}
}


void OnDisconnectedFromServer () {
// If we lose the connection to the server then return to the Master Game Server Lobby
Application.LoadLevel("MasterGameServerLobby");
}

}
```

If you haven't done so yet, NetworkLoadLevel needs to be associated to the MasterServerMenu object in the MasterGameServerLobby Level. Please load the MasterGameServerLobby level, then choose the MasterServerMenu in the Hierarchy, then drag and drop the NetworkLoadLevel script from the Program Window to a free area in the Inspector Window.

InstantiatePlayer

Fantastic, I hope you're getting everything so far. If not please go back and reread some of the more subtle details both in this tutorial and in the code comments. It's important to understand everything up to this point.

Now we'll work on the InstantiatePlayer code discussed above.

Before we instantiate the player we need to enable networking for the player:

1. Pick the Player prefab in the Project view then choose "Component / Miscellaneous / Network View" to add the networking to the player.
2. We also need the BallPickup to be a networked object so let's do the same thing for the BallPickup, choose "Component / Miscellaneous / Network View" to add the networking to the BallPickup.

The InstantiatePlayer code needs to be associated to the right object in our scene. First let's create a new c# file by right clicking on NetworkSampleCode in the Project Window and choosing "Create / C# Script". Name this new script "InstantiatePlayer".

Now let's open the "TestGameLevel" (not the MasterGameServerLobby) level then choose "PlayerSpawn" in the hierarchy view. Now Drag the "InstantiatePlayer" script into an empty area of the Inspector.

Terrific, we're making really good progress now and can actually run the game soon.

Let's take a look at some of the code details:

We create two public variables in this code sample:

```
public Transform PlayerAvatar;

public Transform[] spawn;
```

PlayerAvatar will be set to the PlayerPrefab. This is so we know what player to spawn. We could optionally have this be an array with many different player prefabs mapped to it, and choose the player based on some game logic.

Spawn is an array. Because we have two child spawn areas under PlayerSpawn, we'll change the array size from 0 to 2, then drag each of the child PlayerSpawn1 and PlayerSpawn2 objects into these array areas. The idea is to randomly spawn our player from either spawn point to make it a little more interesting.

InstantiatePlayerOnNetworkLoadedLevel

Next we have the method InstantiatePlayerOnNetworkLoadedLevel, which gets called by the NetworkLoadLevel class as part of the LoadLevel event. This is what actually instantiates our player. A few things are happening here.

1. We pick a random integer number between 0 and 1 to represent one of the two spawn areas.
2. We align the spawn area to the cube, just so the spawned player is facing something interesting when it's spawned. Ideally we could have done this in the level editor, but this is just to show another thing we can do as part of the spawn. When we align the player to face the CUBE we use LookAt, but we keep the original Y of the spawn area, so our player doesn't end up with a pitch to mess up its orientation.
3. We spawn the player using Network.Instantiate. This sets up a buffered request on the GS to instantiate the player on all new client connections. It also sends the instantiate request to us so we too instantiate the local player. We use the spawn's position and the updated spawn rotation, so when we're spawned we end up facing the CUBE.

OnPlayerDisconnected

A separate method is the OnPlayerDisconnected method which gets called on the GS when a client disconnects from the game. As part of the cleanup the GS removes all Buffered client requests from the disconnected client, then sends a DestroyPlayerObject command to all clients so they each properly remove the disconnected client from their scene.

InstantiatePlayer code

Here's a full sample of the code in use. Save it into the InstantiatePlayer code file you just created:

```
using UnityEngine;

public class InstantiatePlayer : MonoBehaviour {

public Transform PlayerAvatar;
public Transform[] spawn;


/****************************************************************
*
*                All Code Below Here is only executed on the SERVER
*
*
*****************************************************************/

// When a new player connects the GS will receive this callback. At that time we'll
// instantiate the player on all clients, with it facing the block in the center of the level
```

```
// at a randomly select spawn point
void InstantiatePlayerOnNetworkLoadedLevel ()
{
Transform spawnarea;

// Let's randomize where we spawn from between the two available spawn areas (0 and 1)
int spawnpoint = (int)Mathf.Round(Random.Range( 0.0f, 1.0f));
Debug.Log("Spawning at : " + spawnpoint.ToString()) ;

// using the generated random #, use the spawn array to find which spawn point to use
spawnarea = spawn[spawnpoint];
GameObject go = GameObject.Find("Cube");

Debug.Log("Instantiate a new player");

// Instantiating Player when Level is Loaded

// First turn the spawn area to be facing the cube. We could have pre done this in the
// level editor, but we'll do it hear just because.
// Note: we only use the CUBE's X and Z position, but keep our
// spawn's Y position so we don't affect the
// pitch orientation of our player, keeping it level to the scene.
spawnarea.LookAt( new Vector3( go.transform.position.x ,
spawnarea.position.y,
go.transform.position.z ) );

// Now Instantiate the player at the spawn area, facing the cube
// Internally this is a BufferedAll RPC call to instantiate the player on all current and
// future clients
Network.Instantiate(PlayerAvatar, spawnarea.position, spawnarea.rotation, 0);
}

// Called on the GS when a remote client disconnects
void OnPlayerDisconnected (NetworkPlayer player)
{
// Removing player if Network is disconnected
Debug.Log("Server destroying player");

// remove any pending Buffered RPC's left from the disconnected player for group 0
Network.RemoveRPCs( player, 0);

// Cleanup just the player and all the objects the player might have instantiated
Network.DestroyPlayerObjects( player ); // send a request to all the remaining clients
// to have them remove the disconnected player
}
}
```

InstantiatePlayer should already be associated to the PlayerSpawn Object in the TestGameLevel Level. If it's not, please repeat the earlier steps and drag/drop the InstantiatePlayer Script to a clear area in the Inspector Window of the PlayerSpawn Object.

Superb! We're almost ready to run and test the game.

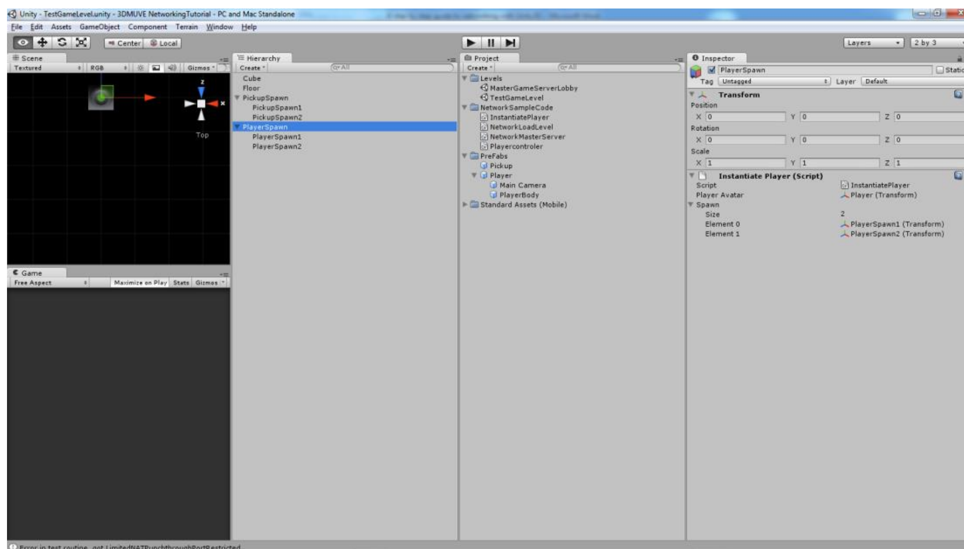
Now save everything and return from the MonoDevelop to the Unity Editor and make sure there are no compilation errors showing up at the bottom.

PlayerSpawn Public Variables Setup

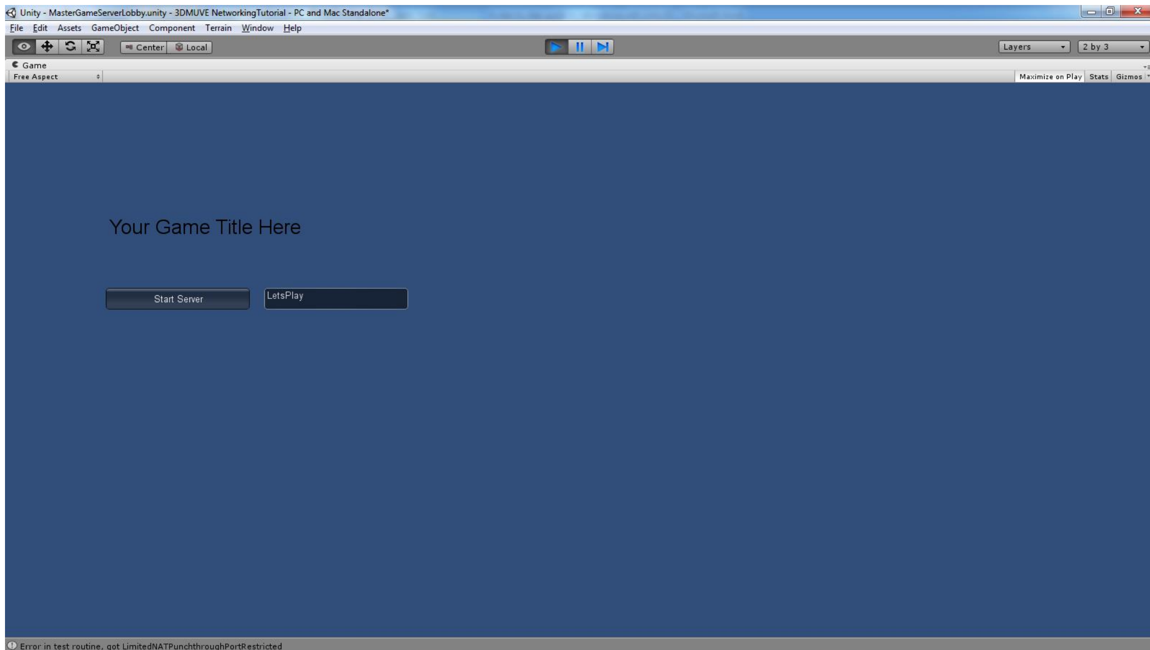
Now if you open the TestGameLevel level and click on the PlayerSpawn object in the hierarchy you should see two public variables under the InstantiatePlayer script in the Inspector Window. Under the Instantiate Player Component, drag the Player Prefab from the Project Window and drop it on the "PlayerAvatar" property in the Inspector.

Next expand the PlayerSpawn in the hierarchy to expose PlayerSpawn1 and PlayerSpawn2. Now in the Inspector change click the spawn property and change the Size from 0 to 2. Now you should have two empty items called "Element 0" and "Element 1". Drag "PlayerSpawn1" from the Hierarchy Window to "Element 0" and "PlayerSpawn2" from the Hierarchy Window to "Element 1".

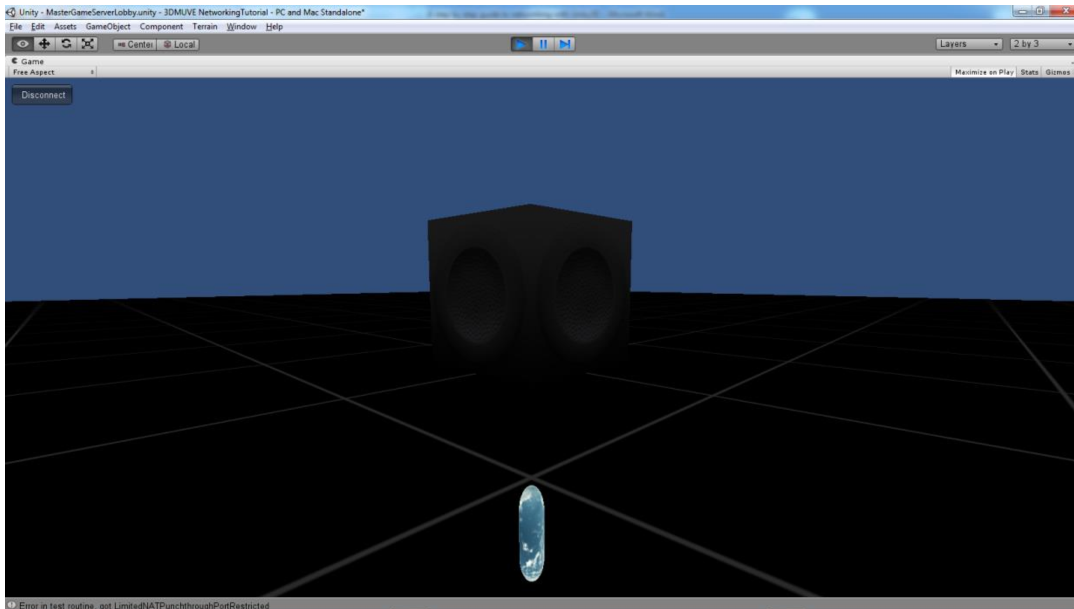
Ok things should look something like this:



If you've got this and no errors we should be able to perform a test run and see our player in the scene. Before you do this however, we want to load the "MasterGameServerLobby" so do that now by double clicking on it in the Project Window, then try and run the level. At this point you should see the main lobby menu appear like this:

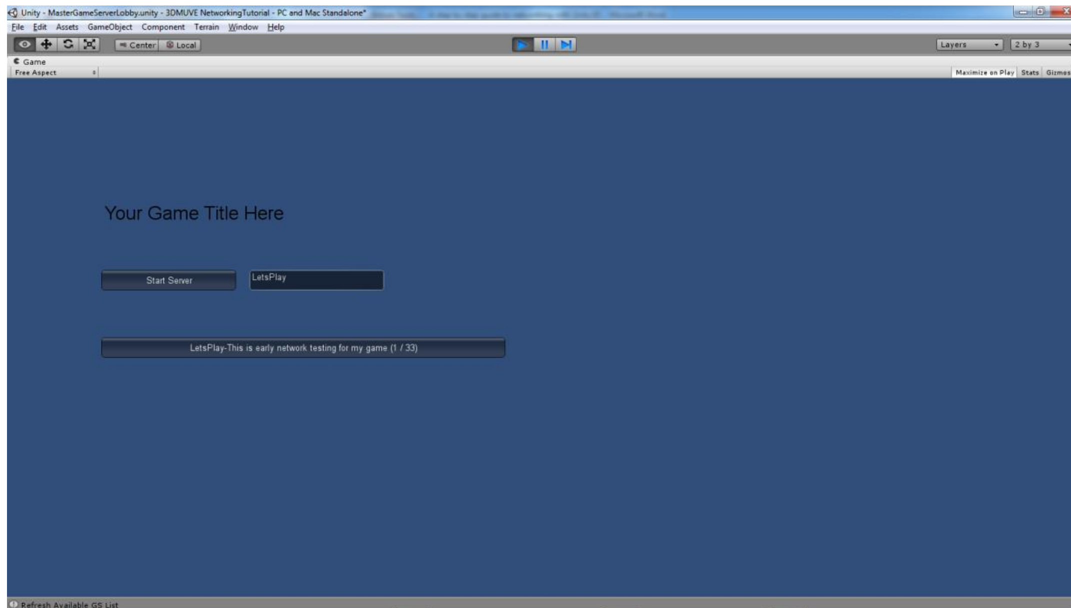


Now if you choose “Start Server” the server should start, the TestGameLevel Level should Load, the player should spawn and you should see something like this:



To test things are fully working you should be able to walk around using the WSAD keys.

Now to really test things out, run the same program from another PC, preferably on the same local home network (just to eliminate any NAT/NPT issues). From the Mater Game Server lobby screen you should see something like this:



You can see here we list the discovered Game Servers underneath the "Start Server" menu selection. If you click on the shown Game you should join that game, but if you do that you'll find the game play doesn't quite work right. What will happen is you'll get two cameras in the scene, one for each player which will generate errors, and you'll also find the controller would control both players. We'll fix these and other issues, and we'll add the BallPickup projectiles to the game next.

PlayerSpawn:

So the reason you get two cameras when the second player joins and an issue with the controls is because the player has the control and camera attached to it so when you spawn more than one character the game no longer knows which one should be the primary. This is a pretty easy fix. What we'll do is add a spawn class to the player so when it spawns it can perform some checks and set things appropriately. Let's take a look.

Let's right click on the NetworkSampleCode folder in the Project Window, and choose "Create / c# script". Name this new script "PlayerSpawn".

Now in the PlayerSpawn code there are several things we need to do:

1. Set the player camera on/off depending on if it's a local player or remote player
2. Set the default audio on/off depending on if it's a local player or remote player
3. Set NetworkRigidbody on/off depending on if it's a local player or remote player.

So the only thing really in PlayerSpawn is the "OnNetworkInstantiate" method, which automatically gets called when an object is instantiated over the network. Therefore this method will be called on both the originating player and the remote player clients. What we do is check if "networkView.isMine" is true which means we're the originating player and therefore we need to set things appropriately for us, or we're the remote player in which case we're instantiating a remote avatar and need to set different things.

In the case of the player we're interested in setting the following:

When we're the originating player we need to:

1. Disable NetworkRigidbody. This is because the Prediction and Smoothing that is performed by this class should only be performed on remote avatars. Our local player will be controlled directly by our key presses and the physics associated to our player.
2. Enable the Player Controls, which ensures we can use the keyboard, mouse and other devices to control our player.
3. Enable the camera that is associated to the player. We could have it on by default but incase the level already has a camera it's better to have it off, and as part of the spawn we would disable the existing camera and enable the player camera.
4. Enable the audio listener. Similar to the camera we could have it enabled ahead of time. However, if the level had a default camera in it that camera probably had an audio listener in it, so we'd want to ensure we disabled the original camera and audio listener before activating the audio listener associated to the player.

When we're a player instantiating a **remote avatar** we basically want to do the opposite

1. Enable the NetworkRigidbody so we receive NetworkView updates and properly apply Prediction and Smoothing.
2. Disable the Player controls because this remote avatar is being controlled by its originating player, and we're just updating its position and orientation via NetworkView updates and NetworkRigidbody to apply Prediction and Smoothing

Renaming Remote Objects:

One more thing to cover about PlayerSpawn. When it's a remote avatar being spawned we change the name to name+"Remote" so later if we're ever searching for local objects we don't accidentally pickup remote objects too.

Depending on what else was associated to the Player character there could be more things to enable/disable. For example in a first person game we might want to enable a HUD for the originating player and enable the model renderer for the remote avatars.

Let's take a look at the PlayerSpawn code:

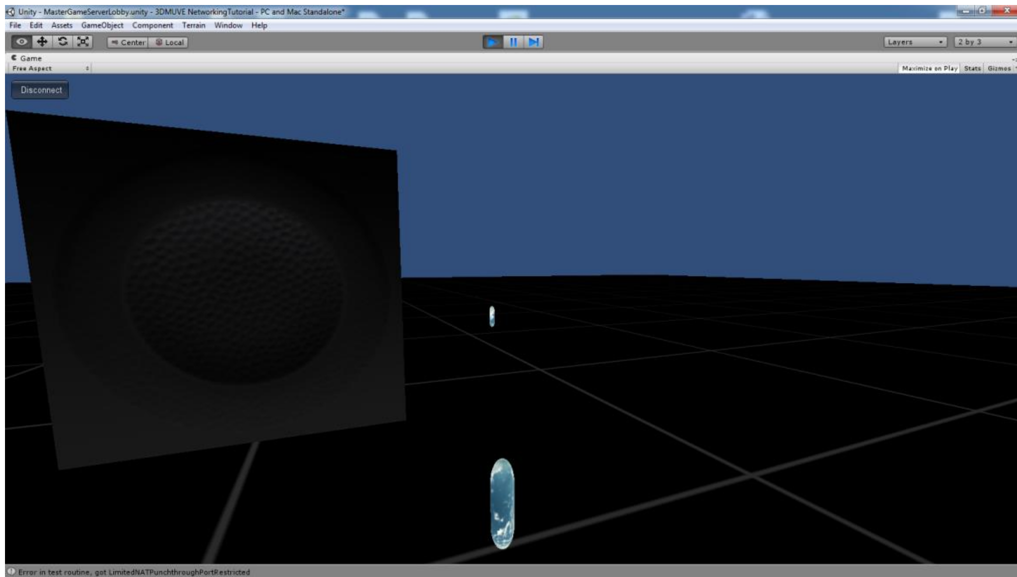
PlayerSpawn code:

```
using UnityEngine;

public class PlayerSpawn : MonoBehaviour {

void OnNetworkInstantiate (NetworkMessageInfo msg)
{
// I like to setup a local variable to our local gameObject so the code is a little more readable
GameObject localplayer = this.gameObject;
//networkView.group = 1; // send all RPC messages for group 1, so only the
// player instantiate is on group 0

if (networkView.isMine)
{
// On the original player since we control the player with the keyboard controls
// so we don't want to use the NetworkRigidbody script which was specific for
// the prediction and smoothing of remote networked character avatars.
// Therefore, let's find that component on the new player and disable it
NetworkRigidbody _NetworkRigidbody =
(NetworkRigidbody) localplayer.GetComponent("NetworkRigidbody");
_NetworkRigidbody.enabled = false;

// Since this is the local player and not the remote avatar we do want to
// ensure the player controls are active.
PlayerController pc = (PlayerController)localplayer.GetComponent("PlayerController");
pc.enabled = true;

// We don't have a Temp Camera setup in the scene, but if we did we would
// need to disable before activating the players camera
//GameObject.Find("TempCamera").SetActiveRecursively(false);
```

```
// Since this is the local player, we want our local player camera enabled as
// the main camera for the scene Since the camera component is in a sub object
// called Main Camera under the Player Prefab we can use GetComponentInChildren
// looking specifically for our "Camera" component. There should only be one, so we
// don't need to loop through multiple components
Camera ca = (Camera)localplayer.GetComponentInChildren(typeof(Camera));
ca.enabled = true; // activate our camera

// Since this is the local player, we want our local player audio enabled.
// Since you can only have one audio enabled in a scene, we need to make sure
// it's not enabled for remote avatars. Therefore we set it disabled
// be default and enable it when we spawn the player
AudioListener al =
(AudioListener)localplayer.GetComponentInChildren(typeof(AudioListener));
al.enabled = true;

}
else
{
name += "Remote"; // Append "Remote" to the name for easier searching of local object

// Since this player object is a remote avatar, we wont be performing any
// manual controls to this avatar. Instead we want all updates to come from
// network updates. The NetworkView will send those updates
// automatically. Therefore we want to enable "NetworkRigidbody"
// component which will process all the prediction and smoothing for our avatar
NetworkRigidbody _NetworkRigidbody =
(NetworkRigidbody) localplayer.GetComponent("NetworkRigidbody");
_NetworkRigidbody.enabled = true;

// Since this is a player avatar for a remote player, we need to make sure its
// camera is disabled. We already have our local player camera so we don't
// need this one
PlayerController pc = (PlayerController)localplayer.GetComponent("PlayerController");
pc.enabled = false;
}
}
}
```

PlayerSpawn must be associated to the Player Prefab so now that the component is created, drag and drop the PlayerSpawn component from the Project Window into an open area of the Player Prefab in the Inspector Window.

If we did everything right when we run the game on two hosts we should see something like this:


Unity - MasterGameServerLobby.unity - 3DMUVE NetworkingTutorial - PC and Mac Standalone*
File
Edit
Assets
GameObject
Component
Terrain
Window
Help
Center
Local
Layers
2 by 3
Game
Free Aspect
Maximize on Play
Stats
Gizmos
Disconnect
Error in test routine, got LimitedNATPunchthroughPortRestricted

NetworkRigidbody:

The Player (and the Ball) objects have physics RigidBodies associated to them. This way they correctly behave to gravity and properly bounce off things. However this adds some complications for a networked game as to where the physics should be applied. Since we're only moving our character on its local client, then using networkView to update the position and rotation on the remote clients, we need to consider when/how we might want the remote updates to provide more details about the state of the objects so we can perform better prediction and smoothing. If you recall I spoke about Prediction and Smoothing earlier. This is the concept of gracefully updating a remote avatar to minimize the effect of LAG and seeing a character appear to jump from one spot to another. What prediction and smoothing accomplishes is to calculate the partial movement from point A to point B over a period of several frames so it moves smoothly instead of unnaturally jumping from location to location.

If you don't want to use NetworkRigidBody things will still work, but your object movement over the network might appear more jumpy/choppy at times and not as smooth. Things will still work exactly the same. NetworkRigidBody doesn't change gameplay at all, it's just about end user ascetics and the illusion of less LAG by moving objects in a more graceful way.

Objects with physics have additional data that can help provide a nicer update on the remote client, such as timing information, velocity forces and angular velocity. To provide this additional data we'll use a script to override the normal behavior of the networkView Observed and State Synchronization properties, so we can include this additional data, and appropriately apply the smoothing and prediction between network updates.

NetworkRigidbody is a class to do this for us. I'm not going to explain the math in detail as our focus is on networking, but you'll see the smoothing uses a Lerp and SLerp function and then performs prediction between network updates.

Just as a reminder there are two ways to update a remote object. The one is to rely on the NetworkView component and its Observed and State Synchronization properties. This sends data automatically (Network.sendRate times per second). The other is using an RPC. We will see how to use the RPC later with the BallPickup Prefab. Think of an RPC as an event driven messaging mechanism where NetworkView is more streaming based.

For now we'll focus on the NetworkView Oberved and State Synchronization parameters. By default when the NetworkView is associated to a game object, the Transform of the object is synchronized which is the position, rotation. If that's all we need to send then we don't need anything special. However, if we want to synchronize custom data then we can override the default data sent by the NetworkView Observed by updating the Observed parameter with the NetworkRigidBody component instead of the default component which is the object that the

NetworkView component is attached to. To do this we first add a new a NetworkRigidBody component to our object, then we drag that NetworkRigidBody component from the **Inspector Window** and drop on the NetworkView.Observed component in the same Inspector Window. This changes the default behavior of the NetworkView Component to use the NetworkRigidBody component to handle sending/receiving the automatic network updates for position and orientation.

The API used to override sending/receiving events is the OnSerializeNetworkView method. Let's cover this in more detail:

NetworkRigidbody OnSerializeNetworkView

The purpose of the OnSerializeNetworkView is to synchronize and stream custom data from the originating player object to all the clients showing the objects avatar. Since our player characters have physics associated to them we want to add some additional data to be synchronized such as timestamp, Velocity and angularVelocity and time. Otherwise just position and rotation are normally sent by default (Transform). By synchronizing this extra data we can do a better job at predicting the continued movement and smoothing the movement between network updates.

The OnSerializeNetworkView provides both a writing and a reading state. The writing state is performed on the original player client and the read is performed by all other clients to perform the prediction and smoothing.

Once you have the script saved lets correctly associate it to the scene. Click on the Player Prefab in the Project Window and then drag the NetworkRigidbody code script into a blank area of the Inspector Window.

It's not enough to just have the NetworkRigidBody script added as a component to our object. Because everything relies on the networkView component, along with the Observed and State Synchronization properties we need a way to override networkView so instead of just updating position and rotation it also updates the other items managed by NetworkRigidBody. To do this we need to change Observed from observing the objects transform to observing the NetworkRigidBody component. To do this:

1. Make sure the Player Prefab is selected in the Project Window
2. Drag the NetworkRigidBody component in the Inspector and drop it onto the Observed property of the networkView component.

NetworkRigidbody code: (original code strike-through, see new code below under 2nd edition)

```
using UnityEngine;

public class NetworkRigidbody : MonoBehaviour
{
public double m_InterpolationBackTime = 0.1;
public double m_ExtrapolationLimit = 0.5;
internal struct State
{
internal double timestamp;
internal Vector3 pos;
internal Vector3 velocity;
internal Quaternion rot;
internal Vector3 angularVelocity;
}

// We store twenty states with "playback" information
State[] m_BufferedState = new State[20];
// Keep track of what slots are used
int m_TimestampCount;
void OnSerializeNetworkView(BitStream stream, NetworkMessageInfo info)
{
// Send data to server
if (stream.isWriting)
{
Vector3 pos = rigidbody.position;
Quaternion rot = rigidbody.rotation;
Vector3 velocity = rigidbody.velocity;
Vector3 angularVelocity = rigidbody.angularVelocity;
stream.Serialize(ref pos);
stream.Serialize(ref velocity);
stream.Serialize(ref rot);
stream.Serialize(ref angularVelocity);
}
// Read data from remote client
else
{
Vector3 pos = Vector3.zero;
Vector3 velocity = Vector3.zero;
Quaternion rot = Quaternion.identity;
Vector3 angularVelocity = Vector3.zero;
stream.Serialize(ref pos);
stream.Serialize(ref velocity);
stream.Serialize(ref rot);
stream.Serialize(ref angularVelocity);
// Shift the buffer sideways, deleting state 20
for (int i=m_BufferedState.Length-1;i>=1;i--)
{
m_BufferedState[i] = m_BufferedState[i-1];
}
```

```
// Record current state in slot 0
State state;
state.timestamp = info.timestamp;
state.pos = pos;
state.velocity = velocity;
state.rot = rot;
state.angularVelocity = angularVelocity;
m_BufferedState[0] = state;
// Update used slot count, however never exceed the buffer size
// Slots aren't actually freed so this just makes sure the buffer is
// filled up and that uninitialized slots aren't used.
m_TimestampCount = Mathf.Min(m_TimestampCount + 1,
m_BufferedState.Length);
// Check if states are in order, if it is inconsistent you could reshuffel or
// drop the out-of-order state. Nothing is done here
for (int i=0;i<m_TimestampCount-1;i++)
{
if (m_BufferedState[i].timestamp < m_BufferedState[i+1].timestamp)
Debug.Log("State inconsistent");
}
}
}

// We have a window of interpolationBackTime where we basically play
// By having interpolationBackTime the average ping, you will usually use interpolation.
// And only if no more data arrives we will use extra polation
void Update ()
{
// This is the target playback time of the rigid body
double interpolationTime = Network.time - m_InterpolationBackTime;

// Smoothing
// Use interpolation if the target playback time is present in the buffer
if (m_BufferedState[0].timestamp > interpolationTime)
{
// Go through buffer and find correct state to play back
for (int i=0;i<m_TimestampCount;i++)
{
if (m_BufferedState[i].timestamp <= interpolationTime || i ==
m_TimestampCount-1)
{
// The state one slot newer (<100ms) than the best playback state
State rhs = m_BufferedState[Mathf.Max(i-1, 0)];
// The best playback state (closest to 100 ms old (default time))
State lhs = m_BufferedState[i];
// Use the time between the two slots to determine if
// interpolation is necessary
double length = rhs.timestamp - lhs.timestamp;
float t = 0.0F;
// As the time difference gets closer to 100 ms t gets closer to 1 in
// which case rhs is only used
// Example:
// Time is 10.000, so sampleTime is 9.900
```

```
// lhs.time is 9.910 rhs.time is 9.980 length is 0.070
// t is 9.900 - 9.910 / 0.070 = 0.14. So it uses 14% of rhs, 86% of lhs
if (length > 0.0001)
t = (float)((interpolationTime - lhs.timestamp) / length);
// if t=0 => lhs is used directly
transform.localPosition = Vector3.Lerp(lhs.pos, rhs.pos, t);
transform.localRotation = Quaternion.Slerp(lhs.rot, rhs.rot, t);
return;
}
}
}
// Use extrapolation (Prediction)
else
{
State latest = m_BufferedState[0];
float extrapolationLength = (float)(interpolationTime - latest.timestamp);
// Don't extrapolation for more than 500 ms, you would need to do that carefully
if (extrapolationLength < m_ExtrapolationLimit )
{
float axisLength = extrapolationLength *
latest.angularVelocity.magnitude *
Mathf.Rad2Deg;
Quaternion angularRotation = Quaternion.AngleAxis( axisLength,
latest.angularVelocity);
rigidbody.position = latest.pos + latest.velocity * extrapolationLength;
rigidbody.rotation = angularRotation * latest.rot;
rigidbody.velocity = latest.velocity;
rigidbody.angularVelocity = latest.angularVelocity;
}
}
}
}
```

~~**2nd Edition:**~~

~~After dealing with some challenges around jitter when updating a remote avatar I worked with a few individuals through various forums and changed NetworkRigidBody to better handle network updates. Special thanks to "Duhprey" on the Unity Forums for his help in getting this code to perform correctly. This version also sends input details for object movement to better simulate the object movement during prediction and smoothing. I'm sure this class can be further enhanced, refined, or modified to meet each of your specific needs.~~

3rd Edition:

My apologies to anyone who received the 2nd edition, as this was the wrong version of the code and referenced Photon Networking. Photon Networking is a 3rd party networking package that is available for Unity, but I did not intend to provide code samples for that. I apologize for the confusion. The below code is the final Unity version of NetworkRigidbody, and in addition to correcting for a jitter problem I encountered in real use.

```
using UnityEngine;
using System.Collections;

public class NetworkRigidbody : MonoBehaviour {

    public double m_InterpolationBackTime = 0.1;
    public double m_ExtrapolationLimit = 0.5;
    private float interpolationConstant = 0.01f;

    internal struct  State
    {
        internal double timestamp;
        internal Vector3 pos;
        internal Vector3 velocity;
        internal Quaternion rot;
        internal Vector3 angularVelocity;
                   internal float speed;
    }

    // We store twenty states with "playback" information
    State[] m_BufferedState = new State[20];
    // Keep track of what slots are used
    int m_TimestampCount;
           NetworkView _pv;
           float queuedspeed=0;

    void Start () {
                   _pv = (NetworkView)gameObject.GetComponent("NetworkView");
        enabled = !_pv.isMine;
    }

    void OnSerializeNetworkView(BitStream stream, NetworkMessageInfo info)
    {
        // Send data to server
        if (stream.isWriting)
        {
            Vector3 pos = transform.localPosition;
            Quaternion rot = transform.localRotation;
            Vector3 velocity = rigidbody.velocity;
            Vector3 angularVelocity = rigidbody.angularVelocity;
                          float speed = queuedspeed;

            stream.Serialize(ref pos);
            stream.Serialize(ref velocity);
            stream.Serialize(ref rot);
            stream.Serialize(ref angularVelocity);
            stream.Serialize(ref speed);

        }
        // Read data from remote client
        else
        {
            Vector3 pos = Vector3.zero;
            Vector3 velocity = Vector3.zero;
            Quaternion rot = Quaternion.identity;
            Vector3 angularVelocity = Vector3.zero;
                          float speed = 0f;

            stream.Serialize(ref pos);
            stream.Serialize(ref velocity);
            stream.Serialize(ref rot);
            stream.Serialize(ref angularVelocity);
            stream.Serialize(ref speed);

                          // Shift the buffer sideways, deleting state 20
            for (int i=m_BufferedState.Length-1;i>=1;i--)
            {
                m_BufferedState[i] = m_BufferedState[i-1];
            }

            // Record current state in slot 0
```

```
                 State state = new State();

                 state.timestamp = info.timestamp;
                               state.pos = pos;
                 state.velocity = velocity;
                 state.rot = rot;
                 state.angularVelocity = angularVelocity;
                 state.speed = speed;
                 m_BufferedState[0] = state;

                 // Update used slot count, however never exceed the buffer size
                 // Slots aren't actually freed so this just makes sure the buffer is
                 // filled up and that uninitalized slots aren't used.
                 m_TimestampCount = Mathf.Min(m_TimestampCount + 1, m_BufferedState.Length);

                 // Check if states are in order, if it is inconsistent you could reshuffel or
                 // drop the out-of-order state. Nothing is done here
                 for (int i=0;i<m_TimestampCount-1;i++)
                 {
                     if (m_BufferedState[i].timestamp < m_BufferedState[i+1].timestamp)
                         Debug.Log("State inconsistent");
                 }
             }
         }

         // We have a window of interpolationBackTime where we basically play
         // By having interpolationBackTime the average ping, you will usually use interpolation.
         // And only if no more data arrives we will use extra polation
         void FixedUpdate () {
             if (m_BufferedState[0].timestamp == 0)
                 return;

             // This is the target playback time of the rigid body
             double interpolationTime = Network.time - m_InterpolationBackTime;

             // Use interpolation if the target playback time is present in the buffer
             if (m_BufferedState[0].timestamp > interpolationTime)
             {
                               //Debug.Log("Performing Smoothing");

                 // Go through buffer and find correct state to play back
                 for (int i=1;i<m_TimestampCount;i++)
                 {
                     if (m_BufferedState[i].timestamp <= interpolationTime || i ==
m_TimestampCount-1)
                     {
                         // The state one slot newer (<100ms) than the best playback state
                         State rhs = m_BufferedState[i-1];
                         // The best playback state (closest to 100 ms old (default time))
                         State lhs = m_BufferedState[i];

                         // Use the time between the two slots to determine if interpolation is
necessary
                         double length = rhs.timestamp - lhs.timestamp;
                         float t = 0.0F;
                         // As the time difference gets closer to 100 ms t gets closer to 1 in
                         // which case rhs is only used
                         // Example:
                         // Time is 10.000, so sampleTime is 9.900
                         // lhs.time is 9.910 rhs.time is 9.980 length is 0.070
                         // t is 9.900 - 9.910 / 0.070 = 0.14. So it uses 14% of rhs, 86% of lhs
                         if (length > 0.0001){
                             t = (float)((interpolationTime - lhs.timestamp) / length);
                         }

                         // if t=0 => lhs is used directly
                         transform.localPosition =
                             Vector3.Lerp (transform.localPosition,
                                           Vector3.Lerp (lhs.pos, rhs.pos, t),
                                           interpolationConstant);
                         transform.localRotation =
```

```
                              Quaternion.Slerp (transform.localRotation,
                                                  Quaternion.Slerp(lhs.rot, rhs.rot, t),
                                                  interpolationConstant);
                          rigidbody.velocity =
                              Vector3.Lerp (rigidbody.velocity,
                                              Vector3.Lerp (lhs.velocity, rhs.velocity, t),
                                              interpolationConstant);
                          rigidbody.angularVelocity =
                              Vector3.Lerp (rigidbody.angularVelocity,
                                              Vector3.Lerp (lhs.angularVelocity, rhs.angularVelocity,
t),
                                              interpolationConstant);

                                                 float newspeed = Mathf.Lerp (lhs.speed, rhs.speed,
t);
                                                 BroadcastMessage("UpdateThrottle", newspeed,
SendMessageOptions.DontRequireReceiver);
                          return;
                      }
                                         //else
                                                 //Debug.Log("Timestamp 0 <= Interpolatin");
                  }
              }
              else
              {
                                 //Debug.Log("Performing Prediction");

                  float dt = (float)(Network.time - m_BufferedState[0].timestamp);
                                 if ( dt == float.NaN )
                                         Debug.Log("dt is NAN");
                  Vector3 extra_pos = m_BufferedState[0].pos + m_BufferedState[0].velocity * dt;
                                 if ( extra_pos.x == float.NaN || extra_pos.y == float.NaN ||
extra_pos.z == float.NaN )
                                         Debug.Log("extra_pos is NAN");

                  float angle = m_BufferedState[0].angularVelocity.magnitude;
                                 if ( angle == float.NaN )
                                         Debug.Log("angle is NAN");
                                 if ( angle == 0 ) angle = 0.001f;
                  Vector3 axis = m_BufferedState[0].angularVelocity / angle;
                                 if ( axis.x == float.NaN ||axis.y == float.NaN ||axis.z ==
float.NaN  )
                                         Debug.Log("axis is NAN");
                  Quaternion extra_rot = m_BufferedState[0].rot * Quaternion.AngleAxis (angle * dt,
axis);
                         if ( extra_rot.x == float.NaN ||
                                         extra_rot.y == float.NaN ||
                                         extra_rot.z == float.NaN ||
                                         extra_rot.w == float.NaN ||
                                         extra_rot.eulerAngles.x == float.NaN ||
                                         extra_rot.eulerAngles.y == float.NaN ||
                                         extra_rot.eulerAngles.z == float.NaN
                                 )
                                         Debug.Log("Rotation is NAN");

                  transform.localPosition = Vector3.Lerp (transform.localPosition, extra_pos,
interpolationConstant);
                  transform.localRotation = Quaternion.Slerp (transform.localRotation, extra_rot,
interpolationConstant);
                         if ( transform.localRotation.x == float.NaN ||
                                         transform.localRotation.y == float.NaN ||
                                         transform.localRotation.z == float.NaN ||
                                         transform.localRotation.w == float.NaN ||
                                         transform.localRotation.eulerAngles.x == float.NaN ||
                                         transform.localRotation.eulerAngles.y == float.NaN ||
                                         transform.localRotation.eulerAngles.z == float.NaN
                                 )
                                         Debug.Log("Rotation is NAN");


                  rigidbody.velocity = Vector3.Lerp (rigidbody.velocity,
```

```
m_BufferedState[0].velocity, interpolationConstant);
                rigidbody.angularVelocity = Vector3.Lerp (rigidbody.angularVelocity,
m_BufferedState[0].angularVelocity, interpolationConstant);

                              BroadcastMessage("UpdateThrottle", m_BufferedState[0].speed,
SendMessageOptions.DontRequireReceiver);
                       }
        }


       void NetworkReset()
       {
              m_BufferedState = new State[20];
       }


}
```

Great, now your player objects should move and look more like a non-networked game, even though there is some network latency that can affect movement. You'll want to do the same thing for the BallPickup and any other objects that will be moving around.

NOTE: Remember to change the Player Prefab networkView.Observed which is normally set to the transform of the object its set to and drag and drop the NetworkRigidBody script in the Inspector onto the Observed property of the networkView in the same Inspector window.

Working with Pickups:

I hope you're feeling pretty good at this point. You actually have a working multi-player prototype that you can take and create your own exciting worlds with. However, we still have one more item to cover and it's a big one. I want to show how to manage pickups, throwing and destroy/respawning them. To do so we'll use the BallPickup prefab we created along with RPC calls to manage the state of events.

From a design perspective you have several ways to handle pickups in your game. You can instantiate, destroy and re-instantiate as part of the spawn, or you could spawn once and just turn the objects on and off as needed. I chose the latter because in this case we know we'll have exactly two balls (two spawn areas) so it's easy to create them once and manage them. After we get through this, I'll offer up a second optional section that shows how to instantiate, destroy then instantiate again. Both techniques have their place.

Aside from which technique we choose, there are also many different ways to implement this code, and no one way is not necessarily better than the other. I'm using a particular style that is comfortable to me. As you learn what I've done and understand the networking techniques you might decide to reengineer certain parts to meet your needs. That's fine as long as you understand and feel comfortable with the process.

For our pickups I've created three new classes and modified the "Instantiate" class:

InstantiatePlayer:

Modified to support new player connection events and lost player disconnection events so we can properly update our scene.

BallSpawnManager:

BallSpawnManager is associated to the PickupSpawn Game Object in the main game level. Its purpose is to Instantiate the initial balls, then appropriately create the balls on new clients, and also update the state of the ball as it's picked up, thrown and respawned.

BallController:

BallController is associated to our player and responsible for keeping track of when the player picks up a ball as well as letting the player throw the ball.

BallManager:

BallManager is for each individual ball that's instantiated and keeps track of the ball specifics such as when it's spawned and available for pickup, when it's picked up, when it's thrown and if thrown what to do if it hits something, and finally working with BallSpawnManager to respawn after it's thrown.

Now before we get into the code, I just want to mention this will be the trickiest thing we've done so far. It might seem trivial but when you think about all that's required to keep things synchronized on every client, update various events and keep track of movement, it can get quite complex and overwhelming at times. Trust that we will work through it and in the end you'll have all the skills you need to create your own networked game in Unity3D.

InstantiatePlayer – Updated for Networking:

We modified the original Instantiate code and added an OnPlayerConnected method and updated OnPlayerDisconnected method. Both of these methods will be called automatically as a networking event. These network events only fire on the Game Server (GS).

InstantiatePlayer OnPlayerConnected

The OnPlayerConnected event is used to perform two primary tasks:

1. The first is to ensure that all the instantiated balls are properly instantiated and also properly updated to the right state such as; sitting at the spawn area to be picked up, being carried, or being thrown. To accomplish this we find the PickupSpawn object, and then grab the BallSpawnManager component. Since all the balls have been instantiated on the GS we know that the BallSpawnManager contains all the balls we need to instantiate.

   We then use the BallSpawnManager object to call the method "PlayerConnectedSpawnBalls" in that component. This method loops through all the spawned balls and sends an RPC to the newly connected player so it can create its own local copy of the ball and set it to the appropriate state.

2. The second is to ensure the remote player avatars are set to the correct state. The players themselves are instantiated using a Network.Instantiate which uses an AllBuffered on the server to queue the requests for new clients. However, each player can have different state values, and in this example it's whether the player is carrying a ball or not. When a player is carrying a ball we actually just turn on the foe ball that represents the player right hand. The foe ball is actually part of the player model we'll be updating soon. By default the right hand is hidden so if the player is supposed to be carrying a ball we hide the original ball in the above step and then show the right hand sub object of the player by toggling the "MeshRenderer" component. Similar to how we spawn the balls, for updating the player we find the local player object on the GS, then grab the BallController and call the method UpdatePlayerState.

   UpdatePlayerState loops through all the players, skips the player that just connected, then checks each player and if that player is holding a ball it sends an RPC to the new client instructing that the specific remote player avatar should be holding a ball. It does this by using the BallControll.networkView for the specific player that is holding the ball. This way when the new client receive the request its automatically receiving it into the proper object and just performs a local update of the player state.

InstantiatePlayer OnPlayerDisconnected

The OnPlayerDisconnected event is used to clean up and remove a disconnected player from all other clients. However, before we remove the player there is specific cleanup we need to do that is associated to the ball. If the player happens to be carrying a ball we need to update the ball to be dropped. When we drop the ball we could put it on the ground right where the player being removed was standing. Instead I chose to return the ball to the original spawn area. You can be creative.

The trick for OnPlayerDisconnected is that we receive a NetworkPlayer object passed in as the player being disconnected. We need to map that NetworkPlayer to the right GameObject player to see if it was carrying a ball. To find it we need to loop through all GameObjects with the "Player" tag and then compare the netorkView.owner of that GameObject to the NetworkPlayer passed in. When they match we found the right player that has been disconnected.

If that player was holding a ball we then get the ball from an array of all spawned balls stored in the BallSpawnManager using the players "pickedupBall_SpawnID" which is the array id for the ball being held. We can then use that balls networkView to send a message to that specific ball on the remaining players using an RPC to RPCMode.All and call the OnInitBall method to update the ball state. This is the same method we call when instantiating a ball or picking up a ball. Instead of writing the same code again and again, OnInitBall is a shared method to update a ball state.

I believe the code is well commented and should provide you the necessary further details you need to see how to perform these tasks. Let's take a look at the code:

InstantiatePlayer Code

```
using UnityEngine;
using System.Collections;

public class InstantiatePlayer : MonoBehaviour {

public Transform PlayerAvatar;
public Transform[] spawn;


/****************************************************************
*
*                All Code Below Here is only executed on the SERVER
*
*
****************************************************************/

// When a new player connects the GS will receive this callback. At that time we'll
// instantiate the player on all clients, with it facing the block in the center of the level
```

```
// at a randomly select spawn point
void InstantiatePlayerOnNetworkLoadedLevel ()
{
Transform spawnarea;

// Let's randomize where we spawn from between the two available spawn areas (0 and 1)
int spawnpoint = (int)Mathf.Round(Random.Range( 0.0f, 1.0f));
Debug.Log("Spawning at : " + spawnpoint.ToString()) ;

// using the generated random #, use the spawn array to find which spawn point to use
spawnarea = spawn[spawnpoint];
GameObject go = GameObject.Find("Cube");

Debug.Log("Instantiate a new player");
// Instantiating Player when Level is Loaded

// First turn the spawn area to be facing the cube. We could have pre done this in the
// level editor, but we'll do it hear just because.
// Note: we only use the CUBE's X and Z position, but keep our spawn's Y position
// so we don't affect the pitch orientation of our player, keeping it level to the scene.
spawnarea.LookAt( new Vector3( go.transform.position.x ,
spawnarea.position.y,
go.transform.position.z ) );

// Now Instantiate the player at the spawn area, facing the cube
// Internally this is a BufferedAll RPC call to instantiate the player on all current and
// future clients
Network.Instantiate(PlayerAvatar, spawnarea.position, spawnarea.rotation, 0);
}

// Called on the GS when a remote client connects
// Technically we could probably spawn the player here, or spawn the balls above,
// but I'm breaking it out to show various techniques
void OnPlayerConnected (NetworkPlayer player)
{
// When a new player connects the level is automatically generated from the
// "AllBuffered" "LoadLevel" that was originally generated
// Then on a new player instantiate we generate all the players
// Now we want to generate the pickup items (Balls) but just on our newly
// connected client, so when the server receives the OnPlayerConnected
// then the server can call the BallManager which can loop through the balls
// that have already been generated and spawn them specifically
// on the new client in their correct position, orientation and state
BallSpawnManager bSM =
(BallSpawnManager)GameObject.Find("PickupSpawn").GetComponent(typeof(BallSpawnManager));
bSM.PlayerConnectedSpawnBalls( player );

// Use UpdatePlayerState to loop through all the players and update the new player with their
// state this is basically to show if a player is holding a ball then the new player needs to know
// for proper rendering
// Get the Local Network Player Ball Controller
BallController _bc = (BallController)GameObject.Find("Player(Clone)").GetComponent(typeof(BallController));
_bc.UpdatePlayerState(player); // Call the UpdatePlayerState in the local Player Ball Controller,
```

```
// which will then update the new player with an RPC
}

// Called on the GS when a remote client disconnects
void OnPlayerDisconnected (NetworkPlayer player)
{
// GameObject _playerGO = null;

// Removing player if Network is disconnected
Debug.Log("Server destroying player");

// When a player disconnects we need to cleanup the player from all other clients.
// But before we do that we need to make sure we reset the state of everything back
// correctly. If the Player was holding a ball, then we want to player to drop the ball before
// cleaning up To do that we need to first associate the NetworkPlayer to the ingame object that
// represents that player. To do so we'll loop through all Player objects and match
// the player passed in to the GameObject.networkView.owner
//
// Loop through all GameObjects of type Player
foreach(GameObject _player in GameObject.FindGameObjectsWithTag("Player"))
{
// Match the player to the NetworkPlayer and if we have a match then this is the player
// we're cleaning up
if ( _player.networkView.owner == player )
{
Debug.Log ("Found player");
// _playerGO = _player;
// Get the players Ball Controller Component
BallController _bc = (BallController)_player.GetComponent(typeof(BallController));
if (_bc.pickedup) // If the Player is holding the ball then let's drop/respawn the ball
{
// To find the ball being held, we have the pickedupBall_SpawnID
// identifier held by the player. This is the ball that originated
// from one of the spawn areas, which is managed by the
// BallSpawnManager. Therefore, we find the root level PickupSpawn
// then get the BallSpawnManager Component which then is mapped
// to the specific spawn area using the players
// "pickedupBall_SpawnID" which gives us the ball object to be
// dropped/respawned.
BallSpawnManager _bsm =
(BallSpawnManager)GameObject.Find("PickupSpawn").
GetComponent(typeof(BallSpawnManager));
BallManager _bm = _bsm.spawnedBalls[_bc.pickedupBall_SpawnID];

// Call the init Ball RPC to update the ball on all clients
_bsm.networkView.RPC("OnInitBall",
RPCMode.All,
_bm.ballSpawnElementID,

_bsm.spawnedBalls[_bm.ballSpawnElementID].transform.position,
_bsm.spawnedBalls[_bm.ballSpawnElementID].transform.rotation,
false,
false,
```

```
Vector3.zero,
Vector3.zero  );
}
break; // Done cleaning up, so get out
}
}

// remove any pending Buffered RPC's left from the disconnected player for group 0
Network.RemoveRPCs(  player, 0);

// Cleanup just the player and all the objects the player might have instantiated
Network.DestroyPlayerObjects(  player  ); // send a request to all the remaining clients to have
// them remove the disconnected player

// Cleanup just the player but not all the objects the player might have instantiated
//Network.Destroy(  _playerGO  ); // send a request to all the remaining clients to have them
//remove the disconnected player
}
}
```

You should have already associated the InstantiatePlayer to the PlayerSpawn object, but if not let's do it now. Choose PlayerSpawn in the Hierarchy then drag and drop the InstantiatePlayer script from the Program Window to a free area in the Inspector Window.

BallSpawnManager:

BallSpawnManager Update

The update routine for BallSpawnManager is used to spawn all the initial balls but just on the GS. We do this by checking Network.Server == true. There are other ways to accomplish this, but I wanted it here as an example of one way to do it. I also chose to use Update and not Awake or Start, to show how we can set a timer and wait for the timer (perhaps to perform other work) before instantiating the initial balls.

From the Update routine we use an RPC OnSpawnBalls with RPCMode.All to instantiate the balls on all clients. However, since players can't join the game until the server is running, when the Update method calls OnSpawnBalls the only client is the GS itself, so this just creates the balls on the GS. By the time a client connects the balls would already be instantiated on the GS, so then when the OnPlayerConnected event fires we call "PlayerConnectedSpawnBalls" which in turn loops through all the balls then calls the same OnSpawnBalls to instantiate the ball on the newly created client.

You may ask then why use an RPC if it's only running on the GS anyhow. Well given the current design you'd be right, we could just call the OnSpawnBalls directly to spawn the balls on the GS,

and then leverage the OnPlayerConnected callback event to do the rest. However, let's consider another option. What if we let all players connect and we had an in-game lobby, then once all players were connected and assigned to teams and chose initial load-outs we then activated the level. In that case using an RPCAll could be the right call as the players are already connected. There are too many variations to consider them all here, and I'm just trying to show you networking related examples. How you apply them in your code will be what separates us from others and makes us more valuable to our teams. Understand what we're doing and why and you'll make the right design decision when it counts.

BallSpawnManager OnSpawnBall

OnSpawnBall runs on each client as it's called using an RPC. It then uses a local Instantiate to create the ball and update the state of the ball with all the data passed in. What I like about an RPC is that you can pass in as many special variables as needed to initialize things the way you need them. We then set the new ball's networkView.viewID to the viewID of the original ball on the GS. By setting it to the same network ID on the GS it allows the ball to receive networkView updates from directed RPC calls as well as the networkView automated network synchronization based on the Observed parameter. That's a feature that automatically updates the position and orientation of an object as it moves.

BallSpawnManager OnInitBall

OnInitBall is called from OnSpawnBall, and a few other areas to setup the state of the ball such as; sitting at the spawn area, being carried and being thrown. As part of setting the state it also toggles on/off various components such as renderers and collision detection. Note: collision detection needs to be toggled off/on as a special switch to reset any settings to "Physics.IgnoreCollision". Although most of the calls to OnInitBall will look like they're non RPC calls, in those cases the parent method making the call is usually an RPC itself.

BallSpawnManager Code:

```
using UnityEngine;

public class BallSpawnManager : MonoBehaviour {
public Transform[] ballSpawnArea;             // Array of valid spawn points
public GameObject theBallPreFabToSpawn;       // The ball prefab to spawn
float timer = 0.0f;                           // timer used to wait for scene to finish loading
// before starting to spawn balls
bool spawned=false;                           // flag set once to signify the balls have been
// spawned
public BallManager[] spawnedBalls;            // Keep an array list of all the spawned balls (sized
// the same as ballSpawnArea since we only spawn 1
// ball per area)
public float spawnDelay;                      // How long to wait when loading a level before
// spawning the balls (this is to give enough time for
// the level to complete loading)
```

```
// Use this for initialization
void Update ()
{
Transform origPos;

// Only the server should spawn the balls initially
if ( Network.isServer && !spawned )
{
timer += Time.deltaTime;
if ( timer >= spawnDelay)  // wait spawn balls.  this is so
{                          // the level finishes instantiating everything
origPos = new GameObject().transform;     // setup a default Identity matrix
// for the starting orientation of
// our Ball
int elementid = 0;         // The first spawned ball is at the first spawn area,
// then we increment by 1 through all the available
// areas
foreach ( Transform spawns in ballSpawnArea ) // loop through all available
// areas
{
// create a new unique network id for each new ball being spawned
NetworkViewID _nvid = Network.AllocateViewID();

// Although I use RPCMode.All here, it's probably really just
// instnatiating the ball on the server. If we let all the players join
// before starting the level, then this would have instantiated the balls
// on all clients, but since I start the level immediately for the server,
// the server is the only client so it's the only one to receive this
//
// However to instantiate the balls on newly connected clients we
// capture the OnPlayerConnected event in "Instantiate"
// then call PlayerConnectedSpawnBalls to loop through all the balls
// we're instantiating here and specifically instantiating
// them on each new client connection
networkView.RPC("OnSpawnBall", RPCMode.All, _nvid, elementid,
ballSpawnArea[elementid].position, // originating position
origPos.rotation,                  // originating rotation
false,                             // not pickedup
false,                             // not thrown
Vector3.zero,                      // no velocity
Vector3.zero                       // no angular velocity
);
elementid++;      // increment the element id so we have one ball per
// spawen area
}
spawned = true;            // when all done spawning, update our flag
}
}
}

// When a new player connects we loop through all balls stored in the spawnedBalls list and then issue a
// command specifically to the new client ONLY to tell it to instantiate each individual ball
// we provide details about the balls location and state so it's properly instantiated on the remote client
```

```
public void PlayerConnectedSpawnBalls(NetworkPlayer player)
{
foreach (BallManager bm in spawnedBalls)
{
networkView.RPC("OnSpawnBall",
player,
bm.networkView.viewID,
bm.ballSpawnElementID,
bm.transform.position,          // current position
bm.transform.rotation,          // current rotation
bm.pickedup,                    // might be picked up
bm.thrown,                      // might be thrown
bm.rigidbody.velocity,          // might have a velocity
bm.rigidbody.angularVelocity ); // might have an angular velocity
}
}

[RPC]
// When a new player connects we need to spawn the pickup balls for that player
// initially the balls are sitting stationary at their spawn area, but the balls can be pickedup and/or thrown
// therefore we need to instantiate the ball and put it corrrectly into the scene so the new player correctly
// sees what's happening in the world
void OnSpawnBall(NetworkViewID _nvid, int spawnElemntID, Vector3 pos, Quaternion _rotation, bool
_pickedup, bool _thrown, Vector3 _velocity, Vector3 _angvelocity )
{
// First we setup our starting orientation then instantiate a local ball. Since this is an RPC it's
// being called on each client therefore we don't need to perform a Network.Instantiate, just a
// local instantiate is sufficient
Transform spawnArea = new GameObject().transform;
spawnArea.position = pos;
spawnArea.rotation =_rotation;
GameObject _ball = Instantiate(theBallPreFabToSpawn, spawnArea.position,
spawnArea.rotation) as GameObject;

// Store the balls network id so it's linked to the right ball on the server and will correctly update
// as the server ball moves
NetworkView nview = (NetworkView)_ball.GetComponent(typeof(NetworkView));
nview.viewID = _nvid;

// Save the ball off in an array of spawned balls
BallManager bmngr = (BallManager)_ball.GetComponent(typeof(BallManager));
bmngr.ballSpawnElementID = spawnElemntID;   // Keep track of what spawn point this ball
// spawned from so when we need to respawn it
// we know where
spawnedBalls[spawnElemntID] = bmngr;        // save off this ball

// If this is a remote avatar of a ball then update the name to append "Remote"
if ( !bmngr.networkView.isMine )
{
_ball.name += "Remote"; // If this is a remote avatar then update the name to reflect
// it's not a primary local active object
}
```

```
// Initialize the ball with the appropriate state
OnInitBall( spawnElemntID, pos, _rotation, _pickedup, _thrown, _velocity, _angvelocity,
Vector3.zero, Vector3.zero );
}

[RPC]
public void OnInitBall( int spawnElemntID, Vector3 _pos, Quaternion _rotation, bool _pickedup, bool _thrown,
Vector3 _velocity, Vector3 _angvelocity, Vector3 forward_force, Vector3 angular_force )
{
// First we setup our starting orientation then instantiate a local ball. Since this is an RPC it's
// being called on each client therefore we don't need to perform a Network.Instantiate, just a
// local instantiate is sufficient
Transform newPos = new GameObject().transform;
newPos.position = _pos;
newPos.rotation = _rotation;

BallSpawnManager _bsm =
(BallSpawnManager)GameObject.Find("PickupSpawn").GetComponent(typeof(BallSpawnManager));
BallManager bmngr = _bsm.spawnedBalls[spawnElemntID];

bmngr.pickedup = _pickedup;                    // store if its currently in a picked up state
bmngr.thrown = _thrown;                        // store if its currently in a thrown state
bmngr.rigidbody.velocity = _velocity;          // store if its currently has a velocity
bmngr.rigidbody.angularVelocity = _angvelocity; // store if its currently has an angular velocity
bmngr.transform.position = newPos.position;     // Update the ball position
bmngr.transform.rotation = newPos.rotation;     // Update the ball orientation

// When a ball is picked up we actually hide it by disabling it's rendering and also disable its
// collision so nothing can bump into it
if ( _pickedup )
{
MeshRenderer mr = (MeshRenderer)bmngr.GetComponent(typeof(MeshRenderer));
mr.enabled = false; // turn rendering off
SphereCollider sc = (SphereCollider)bmngr.GetComponent(typeof(SphereCollider));
sc.enabled = false; // turn collision off
NetworkRigidbody _NetworkRigidbody =
(NetworkRigidbody)bmngr.GetComponent("NetworkRigidbody");
_NetworkRigidbody.enabled = false; // turn the rigid body physics off
bmngr.rigidbody.useGravity = false; // disable gravity for the ball
}
else
{
MeshRenderer mr = (MeshRenderer)bmngr.GetComponent(typeof(MeshRenderer));
mr.enabled = true; // turn rendering back on
SphereCollider sc = (SphereCollider)bmngr.GetComponent(typeof(SphereCollider));

sc.enabled = false; // to turn collision back on we need to toggle it to clear out any old
// No Collision Settings
sc.enabled = true; // turn collision back on
NetworkRigidbody _NetworkRigidbody =
(NetworkRigidbody)bmngr.GetComponent("NetworkRigidbody");
_NetworkRigidbody.enabled = true; // turn the rigid body physics back on
if ( _thrown )
```

```
bmngr.rigidbody.useGravity = true; // enable gravity for the ball
else
bmngr.rigidbody.useGravity = false; // disable gravity for the ball
}

// If we need to add motion to the ball, and we are the local ball (not a remote avatar) then apply
// the force to both the forward and angular direction. This helps provide an arch that looks more
// like a throw
if ( bmngr.networkView.isMine &&
( forward_force != Vector3.zero || angular_force != Vector3.zero )
)
{
// On a throw apply the force on the originating player and let the NetworkRigidBody
// perform the updates
bmngr.rigidbody.AddForce(forward_force, ForceMode.Force);
bmngr.rigidbody.AddForce(angular_force, ForceMode.Force);
}
}

}
```

Now associate the BallSpawnManager script to the PickupSpawn Object in the Hierarchy. Choose PickupSpawn in the Hierarchy then drag and drop the BallSpawnManager script from the Program Window to a free area in the Inspector Window.

Ball Controller:

Ball Controller manages the actions a player performs on a ball such as picking a ball up or throwing a ball.

BallController Update

The Update function simply checks that it's the local player and then looks for a right mouse button press to throw the ball, and of course the ball carried flag being set. The throw ball is performed via an RPC to OnThrowBall, which then uses the OnInitBall to change the state from "picked up" to "thrown".

Something I did differently on the BallController, is that usually we disable a controller for all remote avatar objects and only leave it enabled for the local player. This is so the controller doesn't inadvertently control other avatars. However, in this case because I have code such as "OnInitBall" and "UpdatePlayerState" that get used for local and remote avatar objects alike, so I chose to leave the BallController enabled and check that the player is the local player using "if (networkView.isMine)" wherever needed.

BallController OnCollisionEnter

OnCollisionEnter is an event callback that is used to pick-up the ball. In this case, a local player that is colliding into a ball at rest and is not already holding a ball is able to pick-up the ball.

I use a timer after a ball is spawned to prevent a pick-up too quick. This is because when the ball is thrown into another player we want it to damage that player and then respawn. When the ball is thrown and hits the player two collision events fire, the ball collision to damage the player and the player collision to pickup the ball. If the "**player collision to pickup a ball**" event fires first the ball will be in a thrown state and we won't pickup the ball. However, if the "**ball hit player damage**" event fires first we'll respawn the ball which sets it back to a neutral state, and then the "**player pick-up event"** happens and we think the ball is ok to pickup as it's no longer being carried or thrown, so we end up successfully picking up the ball. This is the wrong behavior so we set a timer after the spawn so if the "**player pickup collision event**" event happens second it'll fail to pick up the ball due to the **justSpawned** timer still running.

BallController OnPickupBall

The OnPickupBall method updates the player state to indicate it's holding a ball and stores the array element ID of the ball being held. It then sets the foe ball subobject of the player to visible, and finally calls OnInitBall to set the ball to a held state.

BallController OnThrowBall

The OnThrowBall method updates the player that it's no longer holding a ball by clearing the picked-up state and then disabling the foe ball sub-object from rendering. It then uses

OnInitBall to put the ball into a thrown state, including the force behind throwing the ball using the physics package.

BallController UpdatePlayerState

The `UpdatePlayerState` method is used on the GS to update any new player clients about the remote player avatars to indicate which players are holding a ball so we toggle that new avatar correctly to a picked up state. The GS loops through all the connected players, and if a player is holding a ball we use that player's BallController.networkView to send an RPC to the newly connected player, specifically for the player holding the ball, and update that remote avatar to the correct state.

BallController OnUpdatePlayerState

The `OnUpdatePlayerState` method is called as an RPC by UpdatePlayerState. As mentioned UpdatePlayerState runs on the GS, but the RPC call to the OnUpdatePlayerState is running on the remote client that just connected.

BallController Code:

```
using UnityEngine;

public class BallController : MonoBehaviour {

public bool pickedup;                    // flag when we picked up the ball
public GameObject righthand;        // foe ball used for showing the ball carry'd in hand and starting spot
// for throwing
public float throwForce;                // how hard to throw the ball
public int pickedupBall_SpawnID;    // originating spawn point captured at pickup and used to respawn
// after destroyed ball
public GameObject theBallThrowModel;       // Model Prefab used when throwing the ball
public GameObject playerBody;       // reference back to the playerbody so we can eliminate collision on
// the originating player when throwing the ball

// Use this for initialization
void Start () {
pickedup = false;
}

// Update is called once per frame
void Update ()
{
// NOTE: We don't DISABLE the BallController on remote avatars as we do the PlayerController
// because the various methods (functions) in BallController are used even on the remote
// avatars to set different local state events. Therefore, on any keyboard, mouse or other
// input events we should only do it if the networkView.isMine is true. This means we're
// performing the action on the local player, not a
// remote avatar. Without this, if multiple players were holding a ball, then pressing throw
// would throw the ball for all players holding the ball

// If we're the local player, and we're holding a ball and we press the right mouse
```

```
// button then throw ball
if ( networkView.isMine && pickedup && Input.GetMouseButtonUp(1) )
{
// Tell all remote players to throw the ball
// Not buffered because whenever a new player connects we catch the
// OnPlayerConnected event in Instantiate
// then use the BallManager.PlayerConnectedSpawnBalls to update the newly
// connected client about the current state of each ball.

// When throwing a ball we update the player that its no longer carrying the ball
// and turn off the foe ball so we aren't holding it anymore
// we then spawn a new ball from our current location
//
// Position the Ball relative to the players mount point (which is their right hand) and
// make the ball visible again
// Throw the ball relative to the player
// Use the playerBody to disable collision with the throwing player
// use the player forward (transform.rotation) to ensure the throw is in the right
// direction pass the pickedupBall_SpawnID which was set on ball pickup
networkView.RPC( "OnThrowBall", RPCMode.All, righthand.transform.position,
transform.rotation, throwForce, pickedupBall_SpawnID );
}
}

// Player collisions with the ball are performed to see if the player can pickup the ball
// If the player isn't already holding a ball, and it's the local player, not a remote player avatar
// then the player picks up the ball
void OnCollisionEnter(Collision gothit)
{
// If we're the local player (networkView.isMine) and we collide with a ball, then proceed to
// pickup the ball but if we're a remote player don't pick it up
if ( networkView.isMine && pickedup == false &&
(gothit.gameObject.tag == "Ball" ))
{
// When the player is colliding with a ball, we're looking for a ball sitting at the spawn
// point which means it wasn't thrown yet. If it's thrown we can't pick it up because if
// it hits us it destroys us (or creates damage)
BallManager _bm = (BallManager)gothit.gameObject.GetComponent(typeof(BallManager));
if (!_bm.thrown && !_bm.justSpawned) // If it's not thrown its at the spawn area so its
// safe to pickup
{
// Tell all remote players to put the ball in their hand by
// enabling the foe ball on all clients
//
// NOTE: It's important to understand that we're using the local
// networkView for our local player, so only it's remote player
// avatars on the other clients receive the command. networkView is specific
// to the object, so All players don't get updated, just the specific player on all
// clients
//
networkView.RPC("OnPickupBall", RPCMode.All, _bm.ballSpawnElementID );
}
}
```

```
}

[RPC]
// When picking up a ball we actually hide the ball by disabling it's renderer and disabling its collision
// we then move it to a neutral position (not that it really matters)
// we then enable the foe ball we've already positioned as part of the player model so it looks like we're
// holding the ball
void OnPickupBall(int ballSpawnElementID)
{
Transform carryPos = new GameObject().transform;

// Setup Player State to be holding the ball
pickedupBall_SpawnID = ballSpawnElementID;
pickedup = true;
// Set the righthand foe ball to render so it looks like we're carrying the ball
MeshRenderer ballInHand = (MeshRenderer)righthand.GetComponent(typeof(MeshRenderer));
ballInHand.enabled = true;

// Change state for the Ball to be pickedup
// This is a shared method used for initial spawn, pickup, throw and respawn of the ball.
// This way we aren't repeating the same basic code again and again
//
BallSpawnManager _bsm =
(BallSpawnManager)GameObject.Find("PickupSpawn").GetComponent(typeof(BallSpawnManager));
_bsm.OnInitBall( ballSpawnElementID,
carryPos.position,  // Set the ball's position to just a neutral location (when picked up
// the renderer will be disabled so it wont be seen anyway)
carryPos.rotation,  // Set the ball's position to just a neutral location (when picked up
// the renderer will be disabled so it wont be seen anyway)
true,               // It is picked up
false,              // It's not thrown
Vector3.zero,       // No motion on the ball when it's picked up
Vector3.zero,
Vector3.zero,
Vector3.zero);
}


[RPC]
// When throwing the ball we update all clients, but the RPCMode is All, not AllBuffered because the ball
// instantiation is handled manually using an OnPlayerConnected even in Instantiate and then
// performing the right steps to instantiate the ball and update the balls state. Otherwise if we used an
// AllBuffered the ball would be instantiated in the wrong place
//
void OnThrowBall(Vector3 newpos, Quaternion newrotate, float throwForce, int spawnElemntID)
{
// Setup the new position and orientation for the ball
Transform newPos = new GameObject().transform;
newPos.position = newpos;
newPos.rotation = newrotate;

// Setup Player State to be throwing the ball
// the ball is no longer picked up so we can pickup another ball
```

```
pickedup = false;
// Set the righthand foe ball NOT to render so it looks like we're no longer carrying the ball
MeshRenderer ballInHand = (MeshRenderer)righthand.GetComponent(typeof(MeshRenderer));
ballInHand.enabled = false;

// Change state for the Ball to be thrown
BallSpawnManager _bsm =
(BallSpawnManager)GameObject.Find("PickupSpawn").GetComponent(typeof(BallSpawnManager));
_bsm.OnInitBall( spawnElemntID,
newPos.position, // Set the ball's position to just a neutral location (when picked up the
// renderer will be disabled so it wont be seen anyway)
newPos.rotation, // Set the ball's position to just a neutral location (when picked up the
// renderer will be disabled so it wont be seen anyway)
false,           // It is picked up
true,            // It's not thrown
Vector3.zero,
Vector3.zero,
newPos.forward * throwForce, // Apply the thrown force so it correctly travels in
// the scene. We probably just need to do this for the
// GS Player
newPos.up * throwForce * 0.1f // as it owns the ball, and then let the
// NetworkRigidBody script update it
);

// Since this RPC was called from a specific local player, then when the RPC is received on the
// remote clients its scope is specific to that same player, now accessing its remote player avatar
// on the other clients. It's because networkView.viewID keeps track of all of this for us.
// Therefore, whe we reference "playerBody" it's referencing the specific object we want to
// interface with
//
// grab the players collider
CapsuleCollider parentcollider = (CapsuleCollider)playerBody.GetComponent(typeof(CapsuleCollider));
BallManager _bm = _bsm.spawnedBalls[spawnElemntID]; // Grab the ball being thrown
// grab the ball's collider
SphereCollider ballcollider = (SphereCollider)_bm.transform.GetComponent(typeof(SphereCollider));
Physics.IgnoreCollision( parentcollider, ballcollider); // disable collision between the two colliders
}

// Whenever a new player connects the GS will loop through all the players and if any of them are holding
// a ball then send an RPC to the new player to tell it which remote player avatar is holding a ball
public void UpdatePlayerState(NetworkPlayer player)
{
// Find all the player objects
foreach( GameObject _player in GameObject.FindGameObjectsWithTag("Player"))
{
if ( _player.networkView.owner != player ) // don't update ourselves with our own
// state, we already know it
{

// Get the BallController for each of the found players
BallController _bc = (BallController)_player.GetComponent(typeof(BallController));
if ( _bc.pickedup ) // we only need to update the player avatar on the new
// client if the player is holding a ball
```

```
{
// send the update to the new player to update
// the specific remote player avatar so it shows to be holding a ball
//
// Note: We're using _bc.networkView to send the update to the
// particular remote player avatar on the new player client
_bc.networkView.RPC( "OnUpdatePlayerState",
player,
_bc.pickedup );
}
}
}
}

[RPC]
// Called on a new client connection to tell this client connection that one of it's remote player avatars
// is carrying a ball. The specific remote player avatar is called, so just update the state directly
void OnUpdatePlayerState(bool _pickedup )
{
// This should always be true, or it should never be called, but we're checking here just for good
// measure if it were false we could raise an assertion.
if ( _pickedup == true )
{
// Grab the BallController object for this player being updated
BallController _bc = (BallController)this.GetComponent(typeof(BallController));
_bc.OnPickupBall(_bc.pickedupBall_SpawnID); // Set this player to be holding the ball
}
}
}
```

Now associate the BallController script to the Player Prefab in the Project Window. Choose Player Prefab in the Project Window then drag and drop the BallController script from the Program Window to a free area in the Inspector Window.

Ball Manager:

The BallManager Class is used to manage specific information about a single ball. It keeps track of the state of the ball such as picked up or thrown. When both are set to false that means the ball is at rest and able to be picked up. It also keeps track of the array ID using ballSpawnElementID, which is the array in BallSpawnManager which represents both the spawn point array in "ballSpawnArea" and the ball itself in the "spawnedBalls" array.

BallManager Update

The Update method is used to manage the timers around both a throw and a justSpawned state. The throw timer is used to auto expire and respawn the ball after it's thrown. If the ball hits a player first it cancels the timer and reverts to the justSpawned timer. The Update method does not check that the ball is a local ball. Therefore this will run on all clients for all balls and reset the local state of the ball. This is in case of LAG or dropped packets we want to be sure the ball resets correctly for the local player so things look to behave correctly.

BallManager OnCollisionEnter

The OnCollisionEnter method is an event action called when the ball hits something. We then check to see if the ball hits a player which is a reset action. Otherwise we let the ball bounce around until the throw timer expires. If the ball hits a player we then check to see if it's a local or remote avatar. If it's a local player we could apply damage to that player and perform other player actions. We also initiate a ball reset. If the player is a local player, we issue the reset using an RPC so all clients are reset. If the player hit is a remote avatar, then we just process the hit locally. We don't apply any damage and just reset the ball so things look coorect. This concept goes back to what I said earlier about a hybrid style GS, where we perform some logic in the GS to handle the balls, but other actions are controlled by the local players, on whatever client they are associated to.

BallManager ReSetBallBackToSpawnArea

The ReSetBallBackToSpawnArea method is basically a wrapper for the OnInitBall method in the BallSpawnManager class. The wrapper needs to find the BallSpawnManager class, locate the right spawn area in the class by looking it up from the ballSpawnArea array, then calling on OnInitBall with the right values.

Ball Manager Code:

```
using UnityEngine;

public class BallManager : MonoBehaviour {

public bool thrown;                 // Flag when the ball is being thrown
public bool pickedup;               // Flag when the ball is being picked up
public int ballSpawnElementID;      // Initial spawn point based on PickupSpawn Points in level used to
// respawn after ball destroy
```

```
float timer;                          // timer once ball is thrown to perform self destruct, and to prevent
// immediate pickup after respawn
public float throwDurationTimer;
public bool justSpawned;              // Don't allow the ball to be picked up immediately after spawning
public float pickupDelay;             // How long to wait before it's ok to pickup the ball after a ball
// respawn

// Use this for initialization
void Awake () {
timer = 0.0f;
thrown = false;
pickedup = false;
justSpawned = false;
}

void Update()
{
if ( thrown )
{
timer += Time.deltaTime;
if ( timer >= throwDurationTimer ) // if after 5 seconds we don't collide with anything
// then perform the respawn anyway
ReSetBallBackToSpawnArea();
}
else if ( justSpawned )
{
timer -= Time.deltaTime;
if ( timer <=0 )
{
timer = 0.0f;
justSpawned = false;
}
}
}


// Once the ball is thrown, look for a collision with a player and upon collision respawn the ball
void OnCollisionEnter(Collision gothit)
{
if (thrown )
{
// If the collision is with a Player (we know we aren't hitting ourselves because we
// disabled collision on the throw)
if ( gothit.gameObject.tag == "Player" )
{
// If the player we hit is a local player, not a remote avatar
// then maybe perform some damage processing here and potentially
if ( gothit.gameObject.networkView.isMine )
{
// pseudo code example of what you might do
// in reality you'd probably call a method associated to the player to
// perform this logic
```

```
//gothit.gameObject.damage  -= ball.damage;
//if ( gothit.gameObject.damage  <= 0 )
//          player dead;

// If we hit a local player, not a remote player then tell all clients to
// reset the ball
networkView.RPC("ReSetBallBackToSpawnArea",RPCMode.All);
}
else
ReSetBallBackToSpawnArea();
}
}
}

[RPC]
public void ReSetBallBackToSpawnArea()
{
// Change state for the Ball to be pickedup
BallSpawnManager _bSM =
(BallSpawnManager)GameObject.Find("PickupSpawn").GetComponent(typeof(BallSpawnManager));
Transform spawnArea = _bSM.ballSpawnArea[ballSpawnElementID].transform;
_bSM.OnInitBall( ballSpawnElementID,
spawnArea.position,   // Set the ball's position to just a neutral location (when
// picked up the renderer will be disabled so it wont be seen anyway)
spawnArea.rotation,   // Set the ball's position to just a neutral location (when picked up
// the renderer will be disabled so it wont be seen anyway)
false,                // It is picked up
false,                // It's not thrown
Vector3.zero,         // No motion on the ball when it's picked up
Vector3.zero,
Vector3.zero,
Vector3.zero);

timer = pickupDelay; // reset the timer
justSpawned = true;
}
}
```

Now associate the BallManager script to the BallPickup Prefab in the Project Window. Choose BallPickup Prefab in the Project Window then drag and drop the BallManager script from the Program Window to a free area in the Inspector Window.

So we're just about to have a working prototype of a fully networked multi-player game, where players can join and depart, players can walk around, collide into objects appropriately, spawn balls to a pickup spawn area, pickup balls, throw balls and apply damage if necessary when a ball hits a player.

Updating Prefabs:

Updating the Player Prefab

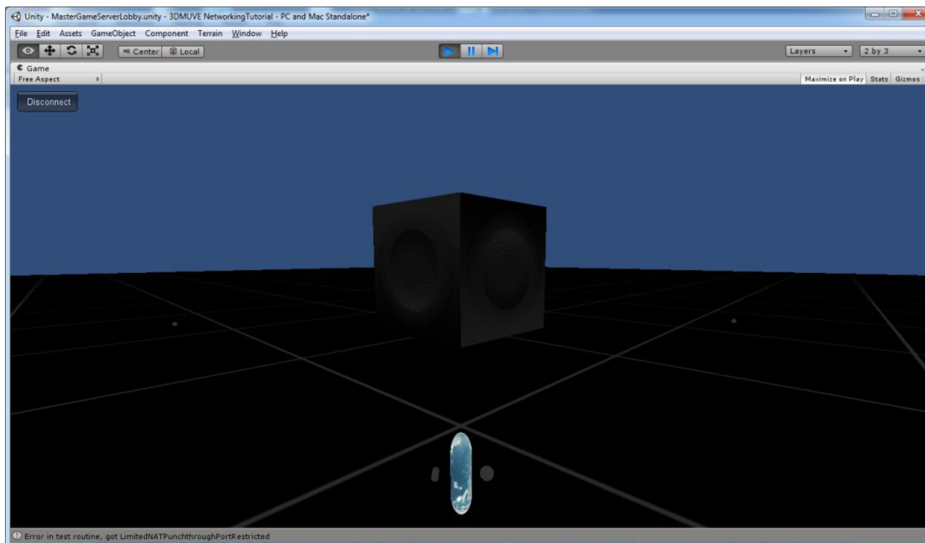
We need to make a quick change to the Player Prefab. We need to add a sub model object in the position of the right hand. Technically it doesn't need to be positioned perfectly, the code will work no matter where we position it, but I set it up more or less as a right hand. To start with create a new Sphere using "Game Object / Create Other / Sphere" and rename it to RightHand. Then Delete the Sphere collider component.

Next Copy the Player Prefab from the Project Window over to the Hierarchy window. Then Copy and drag the RightHand object into the player in the hierarchy. Now modify the transform for the RightHand to be (18,0,0) and set the scale to (10,10,10) and leave rotation to (0,0,0). This should position the right hand object (foe ball) more or less in the position of where the right hand would be.

While we're here I also want to setup the LeftHand. We aren't using it yet, but we will be in our next section so we might as well do it now while we're here. Let's create a new capsule this time using "Game Object / Create Other / Capsule" and rename it to LeftHand. Let's delete the Capsule Collider from the object. Now drag the object into the player and modify the transform for the LeftHand to be (-18,0,0) and set the scale to (5,8,5) and change rotation to (90,0,0). This should position the left hand object (foe spear) more or less in the position of where the left hand would be.

Now drag the new revised Player object from the hierarchy overtop of the Player Prefab in the Project View. This will replace the Old Player with the new one, now containing a RightHand and LeftHand Object. Finally delete the Player Object from the Hierarchy so we are just left with the Player Prefab.

The new player should look something like this:



Now that we see it working correctly let's modify one more thing. Click on the LeftHand object in the Player Prefab and uncheck the MeshRenderer so it doesn't draw. Let's also do the same for the RightHand object. This is so by default we aren't carrying anything, and we'll modify the render state in code at the appropriate time.

Updating the BallPickup Prefab

We need our BallPickup to be a Networked Object, so If you haven't done so yet, let's add networkView component now using "Component / Miscellaneous / NetworkView"

Since we have physics associated to the BallPickup with a "Physics / RigidBody", we need to associate the NetworkRigidBody to the BallPickup to properly handle the physics updates over the network. Click on the BallPickup in the Project Window and then drag the **NetworkRigidbody code** script into a blank area of the Inspector Window.

It's not enough to just have the NetworkRigidBody script added as a component to our object. Because everything relies on the networkView component, along with the Observed and State Synchronization properties we need a way to override networkView so instead of just updating position and rotation it also updates the other items managed by NetworkRigidBody. To do this we need to change Observed from observing the objects transform to observing the NetworkRigidBody component. To do this:

a. Make sure the BallPickup is selected in the Project Window
b. Drag the NetworkRigidBody component in the Inspector and drop it onto the Observed property of the networkView component.

Initializing Public Variables:

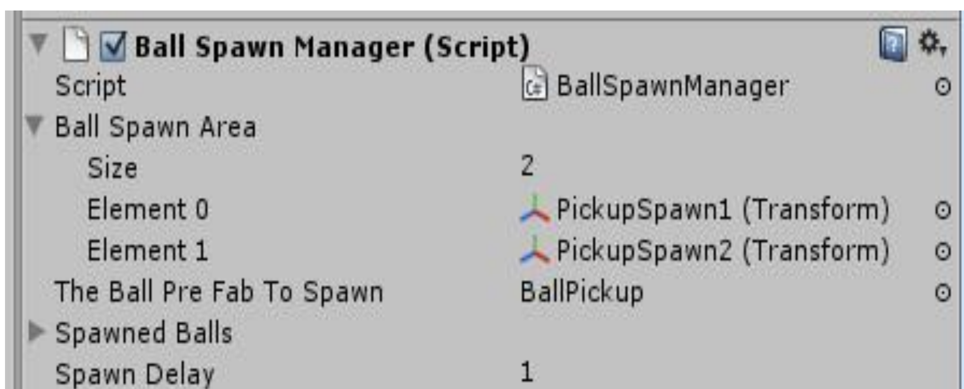
Now that we've completed adding these three new classes there are several public variables that need to be configured for things to work correctly. We'll now take a look at the proper settings for some of the public variables. Note: some are public for easier communications between classes instead of creating a Get/Set method to do it.

BallSpawnManager Public Variables:

As indicated earlier, BallSpawnManager is associated to the PickupSpawn in the TestGameLevel level we created. If you haven't done so already, choose the PickupSpawn in the hierarchy, then drag the BallSpawnManager into an empty area of the Inspector

1. **BallSpawnArea** array
   a. Modify SIZE to 2. This is the two spawn objects that are child objects to the PickupSpawn object.
   b. Drag the PickupSpawn1 to Element0
   c. Drag the PickupSpawn2 to Element1
2. **BallPrefabToSpawn**
   a. This is the Ball Prefab to use when we display the ball. Drag the BallPickup prefab and drop it here
3. **SpawnedBalls**
   a. This will hold the ball reference for each ball once it's spawned. For now just modify the SIZE=2, which has to match the BallSpawnArea size. The individual elements will be set in code
4. **SpawnDelay**
   a. SpawnDelay is how long to wait before the ball can be picked up after it's spawned. I just set it to "1" for 1 second

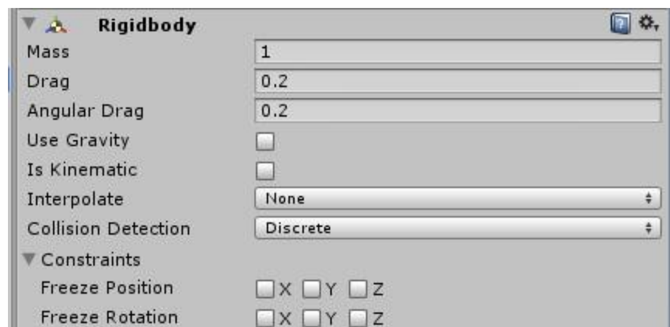
Here's a quick look at how the variables should look:



BallController Public Variables:

As indicated earlier, BallController is associated to the Player Prefab. If you haven't done so already, choose the Player in the prefab section of the Project Window, then drag the BallController into an empty area of the Inspector

1. **Pickedup**
   a. This is used internally to track when the player is holding a ball. Just leave it unchecked and it'll be set in code
2. **Righthand**
   a. This holds the FOE ball in the position of the right hand. Remember the RightHand subobject we created as part of the player prefab. Let's drag that subobject from the Project Window on top of the RightHand public variable. By mapping it here we can just reference it as a local variable in the code. Otherwise, we'd have to search for it and we'd find all the right hands for all the players and then we'd then have to find the one with the matching parent to the object we were working in. That would be really inefficient. By making the linkage this way we can easily use the object in our code without having to search for it.
3. **ThrowForce**
   a. This is how much force to use when throwing the ball. The value really depends on how you set the physics for the BallPickup Prefab. Before we set the ThrowForce lets update the BallPickup.RigidBody.
      i. Mass = 1
      ii. Drag = 0.2
      iii. Angular Drag = 0.2
      iv. Use Gravity = 0 (We toggle this in code so when sitting at rest we float above the ground)



   b. With the above physics set on the ball I use a force = 20,000. You can play with it and try different values until you find something you like
4. **Pickedup Ball Spawn ID**

a. This is set in code and can be left equal to 0

5. **The Ball Throw Model**
   a. This is the Ball Pickup Prefab. Just drag the BallPickup Prefab from the Project Window and drop it here
6. **Player Body**
   a. This is the PlayerBody Capsule we created as part of the Player Prefab. Just drag the PlayerBody subobject of the Player Prefab from the Project Window and drop it here. We use this to easily set "No collision" on the playerbody of the player throwing a ball or spear.

Here's a quick look at how the variables should look:

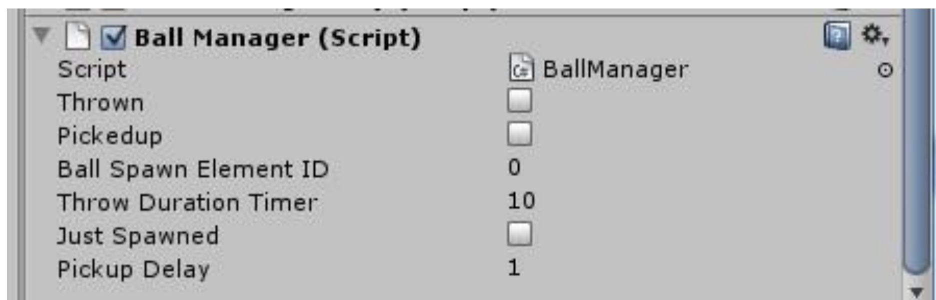
BallManager Public Variables:

As indicated earlier, BallManager is associated to the BallPickup Prefab. If you haven't done so already, choose the BallPickup in the prefab section of the Project Window, then drag the BallManager into an empty area of the Inspector

1. **Thrown**
   a. This is just a flag used in the code. Leave it unchecked.
2. **Pickedup**
   a. This is just a flag used in the code. Leave it unchecked.
3. **Ball Spawn Element ID**
   a. This is set in code when the Ball is created to associate the ball to the spawn area using the array element ID

4. **Thrown Duration Timer**
    a. This is how long to wait after throwing the ball before auto destroying and respawning
    b. Set equal to 10
5. **Just Spawned**
    a. This is just a flag used in code and should remain unchecked
6. **Pickup Delay**
    a. This is how long to wait after respawning the ball
    b. Set equal to 1

Here's a quick look at how the variables should look:



CONGRATULATIONS, we're done:

Wow, I think we've done it. At this point if I haven't forgotten anything I think we're done and ready to run the game. You should now be able to walk around pickup balls, throw them at other players and have full networking support for up to 32 players using the settings in the code. Give it a try and good luck!

Appendix A - Managing dynamic Objects

Creating a Throwing Spear

How about we look at implementing one more thing? I already had you create the left hand on the player holding a foe spear. You probably have enough of the basics to figure it out on your own but there are a few things we've not really touched on so I thought we could add this additional functionality without too much trouble.

Unlike the Balls that were instantiated at the game start-up by the GS and toggled on and off, this time we will actually instantiate and destroy a new spear every time we throw it. This way you'll learn a new technique that is more commonly used for things like weapons fire or other dynamic requirements where the number of objects is unknown.

We already have the foe spear for the player's left hand. Let's go back to the Player prefab in the Project Window and turn the LeftHand object back on. We want the default starting state to have the player holding the spear. Then each time we throw the spear will create a timer then destroy the thrown spear, and use a separate timer to instantiate a new spear.

Now we want to create the actual spear we'll throw so to do that lets use "Game Object / Create Other / Capsule". Rename the new capsule to "Spear". Let's modify its position to (0,0,0) and rotation to (0,0,0) and change its scale to (5, 8, 5).

Next let's add a physics rigidbody to the spear using "Component / Physics / Rigidbody"

1. Set the Mass to 1
2. Set the Drag to 0.2
3. Set the Angular Drag to 0.2
4. Disable Use Gravity

Let's add a Network View Component using "Component / Miscellaneous / Network View"

Now add the NetworkRigidBody script we have in the Project Window to the Spear

1. Select the Spear in the Hierarchy and scroll down in the Inspector to a clear spot or the end
2. Drag the NetworkRigidBody script from the Project Window and drop it onto the bottom of the Spear Inspector Window
3. Now Modify the networkView.Observed and change it from the actual object to this script component by dragging the NetworkRigidBody component in the Inspector and dropping it on the networkView component Observed property, in the same Inspector Window.

Drag the Spear to the prefab, and then delete the spear from the hierarchy.

Excellent. Now we're going to add two new code modules:

Spear Code:

Spear Controller will be associated to the player and used to keep track of the player state of the spear. It'll control when the player can throw a spear, using a reload timer. When the player throws a spear it'll instantiate the spear on all clients using an RPC call. It also toggles the foe spear on/off so it looks like the player is holding the spear.

We do something special here when instantiating the spear. Only a local player can instantiate a spear, but if that player drops while the spear is mid-flight, we don't want to lose the spear. If the local player is controlling the trajectory and updating the spears location, and that player goes away the spear would be orphaned. To rectify this we actually send a directed RPC just to the server asking it to create the spear, and then the server sends an RPC to all clients to create the remote spear avatar. This way the Spear is associated to the GS, and if a remote client player disconnects it won't affect the spear as it travels.

We could have also handled this by letting the local player instantiate the spear and control the spear movement, but if the player dropped we'd have to ensure two things; one that the spear wasn't destroyed as part of the OnPlayerDisconnected and two that the spear kept traveling by using the prediction code. Both can be achieved, but I won't go into detail here. I do discuss orphaned objects briefly in Appendix B under "Advanced Concepts". As for advanced prediction updates, the current NetworkedRigidBody code stops if no update is received after 500 ms. I'll leave it to you to figure it out if you're so inclined ☺

SpearController Code:

```
using UnityEngine;

public class SpearController : MonoBehaviour {

public bool thrown;                   // flag when the spear is thrown
public GameObject lefthand;           // foe spear used for showing the spear carry'd in player's left hand
// and starting spot for throwing
public float throwForce;              // how hard to throw the spear
public GameObject theSpearThrowModel;     // Model Prefab used when throwing the spear
public GameObject playerBody;         // reference back to the playerbody so we can eliminate collision on
// the originating player when throwing the spear
public float ReloadTimer;             // Amount of time after throwing a spear before we can throw again
float timer;                          // general timer

// Use this for initialization
void Start () {
thrown = false;              // When thrown is false, the spear should be in our hand
```

```
}

// Update is called once per frame
void Update ()
{
// NOTE: We don't DISABLE the SpearController on remote avatars as we do the PlayerController
// because the various methods (functions) in SpearController are used even on the remote
// avatars to set different local state events. Therefore, on any keyboard, mouse or other
// input events we should only do it if the networkView.isMine is true. This means we're
// performing the action on the local player, not a remote avatar. Without this, if multiple players
// were holding a spear, then pressing throw would throw the spear for all players holding the
// spear

// When a spear is thrown we use a timer to simulate the reload time so we can then carry
// another spear
if ( thrown && timer > 0 )
{
timer -= Time.deltaTime;
if ( timer <= 0 ) // When the timer reaches 0, we can reload
{
// NOTE: We aren't calling this with an RPC. The timer will run on each client
// and reload the spear into the client hand. We don't need to generate any
// extra network traffic unnecessarily and only the local player can fire the
// weapon anyway
//
ReloadSpear(); // Changes the player thrown flag to false, and toggles the foe
// spear in the players hand
}
}
else
{
// If we're the local player, and we're holding a spear and we press the left mouse button
// then throw the spear
if ( networkView.isMine && !thrown && Input.GetMouseButtonUp(0) )
{
// Tell all remote players to throw the spear
// Not buffered because whenever a new player connects we catch the
// OnPlayerConnected event in Instantiate then use the UpdatePlayerState to
// make sure the players are correctly showing the spear if they're holding it.

// When throwing a spear we update the player that its no longer carrying the
// spear and turn off the foe spear so we aren't holding it anymore
// we then spawn a new spear from our current location
//
// Position the spear relative to the players mount point (which is their left
// hand)
// Throw the spear relative to the player
// Use the playerBody to disable collision with the throwing player
// use the player forward (transform.rotation) to ensure the throw is in the right
        // direction
//
// We Actually do things a little different here. In this case we're sending the
// request to spawn a spear to the server, and then let the server send an RPC to
```

```
// spawn the spear on all the clients.  This is so when a client disconnects,  any
// spear they've  thrown  doesn't also go away  because the owner goes away.  By
// letting  the server send  it, the  objects  seemingly  spawned  by a player  will still
// exist  if the player  should  drop
if ( Network.isServer   )
{
OnServerThrowSpear(   lefthand.transform.position,
transform.rotation,
throwForce   );
}
else
networkView.RPC("OnServerThrowSpear",
RPCMode.Server,
lefthand.transform.position,
transform.rotation,
throwForce   );
}
}
}


[RPC]
void OnServerThrowSpear(   Vector3  newpos,  Quaternion   newrotate,   float throwForce)
{
// Generate  a networkview.viewid   so we can send it to all the clients  and have  them associate
// the  remote  spear avatars  with  the  GS
NetworkViewID   _nvid  = Network.AllocateViewID();

// Call  the  RPC to  instantiate   the spear on all  clients
networkView.RPC("OnThrowSpear",   RPCMode.All,  _nvid, newpos,  newrotate,  throwForce   );
}


[RPC]
// When  throwing  a spear  we need to  update  the player  to hide the foe  spear  and set the reload  timer
// so the player  can reload  a new spear.  We then instantiate   a new spear as a network  object
void OnThrowSpear(  NetworkViewID   _nvid,  Vector3 newpos,  Quaternion   newrotate,   float throwForce)
{
// Setup  the new position  and orientation   for the ball
Transform   newPos  = new  GameObject().transform;
newPos.position   = newpos;
newPos.rotation   = newrotate;  // we passed in the players  rotation,  so we get the right
// orientation
newPos.Rotate(   90.0f, 0.0f, 0.0f ); // We want  the  spear facing  forward,  but by default  a capsule
// is oriented  up and down,  so rotate  it forward  90 degrees so it
// faces  front

// Setup  Player State  to be throwing  the ball
// the  ball is no longer  picked up so we can pickup  another  ball
thrown  = true;
// Set the lefthand  foe spear NOT to  render  so it looks like we're no longer  carrying  the spear
MeshRenderer   spearInHand   = (MeshRenderer)lefthand.GetComponent(typeof(MeshRenderer));
spearInHand.enabled   = false;
```

```
// Instantiate a local avatar of the spear
GameObject _spear = Instantiate( theSpearThrowModel,
newPos.position,
newPos.rotation) as GameObject;
// Get the local networkView component of the new spear
NetworkView nview = (NetworkView)_spear.GetComponent(typeof(NetworkView));
nview.viewID = _nvid; // Save the Global Identifier for the spear so we'll properly receive
// network updates

// don't collide with ourselves

// grab the players collider
CapsuleCollider parentcollider = (CapsuleCollider)playerBody.GetComponent(typeof(CapsuleCollider));

// grab the spear collider
CapsuleCollider spearcollider = (CapsuleCollider)_spear.GetComponent(typeof(CapsuleCollider));
Physics.IgnoreCollision( parentcollider, spearcollider); // disable collision between the two
// colliders

// Because the spear model is rotated 90 degrees to have it look right, we add a force along the
// UP vector instead of the forward vector. Ideally we'd be loading a correctly oriented model so
// we didn't have to do cheats like this, but it gets the job done
//
// Note: we only set the force on the GS, and we rely on the NetworkRigidBody script to perform
// the smoothing and prediction necessary to keep the spear traveling
if ( Network.isServer )
_spear.rigidbody.AddForce( newPos.up * throwForce, ForceMode.Force);

// We need to store the throwing player as part of the spear object, so if the spear hits something
// and we destroy it before the timer runs out we can reset the original player to be holding the
// spear again
SpearManager _sm = (SpearManager)_spear.GetComponent(typeof(SpearManager));
_sm.ThrowingPlayer = this.gameObject;

timer = ReloadTimer;
}

public void ReloadSpear()
{
// Setup Player State to be throwing the ball
// the ball is no longer picked up so we can pickup another ball
thrown = false;
// Set the lefthand foe spear NOT to render so it looks like we're no longer carrying the spear
MeshRenderer spearInHand = (MeshRenderer)lefthand.GetComponent(typeof(MeshRenderer));
spearInHand.enabled = true;
}
}
```

If you haven't already done so please associate the SpearController to the Player Prefab by copying into an open area in the Player Prefab Inspector Window.

Spear Manager will be associated to the spear prefab and used to time how long the spear stays air born and destroy the spear after the allotted time. It'll also check for spear collision and if the spear hits anything it'll destroy the spear immediately. If the spear collision is with a local player then we perform an RPC call to destroy the spear on all clients. However if the spear hits a remote player avatar or a non player, then we just perform a local destroy. This is because LAG can cause us to miss a hit on one client but not another, and the one that counts is the one that is the local player. Remember, we've designed a hybrid style GS, where the individual clients can make some decisions, just not all.

SpearManager Code:

```
using UnityEngine;

public class SpearManager : MonoBehaviour {

public GameObject ThrowingPlayer=null;      // Reference to the player that through the spear, incase we
// need to keep score of who gets hit/kill points
public float flightTime;                     // How long to let the spear travel if it doesn't hit something
// before respawning the spear
float timer;

void Start()
{
timer = flightTime; // Startup a timer when the spear is thrown
}

void Update()
{
timer -= Time.deltaTime;

// When the timer expires perform a cleanup of the spear and reset the player to be holding a
// spear
if ( timer <= 0 )
{
OnSpearRemove();          // After an appropriate flight time (delay) remove the spear
// by destroying it.
}
}

// When the spear hits a local player then we can apply damage to the local player. We also want to
// immediately stop processing of the spear so we use an RPC to stop the spear on all clients and respawn
// it to the players hand
//
// However, if the spear hits anything else we destory the spear locally only. That's because due to LAG we
// might miss the player on one client but hit it on the client where the player is local. Therefore we only
// perform a local cleanup so if we do hit the local player on another client then we'll still award the hit
//
void OnCollisionEnter(Collision gothit)
{
```

```
// If we hit a player and the hit player is local on the client where the hit is occuring
if ( gothit.collider.tag == "Player" && gothit.collider. networkView.isMine )
{
// Apply damage to the hit player
//
// ... do it here

// since we hit a local player perform a full cleanup on all clients using the RPC
networkView.RPC("OnSpearReSpawn", RPCMode.All);

}
else // if we hit a remote avatar or the cube in the center of the level then just cleanup the local
// spear but let it keep traveling on the other clients
{
// This is a local cleanup only because the spear could still hit a local player on another
// client
OnSpearRemove();
}
}

[RPC]
// When cleaning up a spear just get rid of it
public void OnSpearRemove()
{
// Destroy the local spear
GameObject.Destroy( gameObject );
}

}
```

If you haven't already done so please associate the SpearManager to the Spear Prefab by copying into an open area in the Spear Prefab Inspector Window

Now the final step is to initialize the public variables to the two new classes:

Spear Controller Variables:

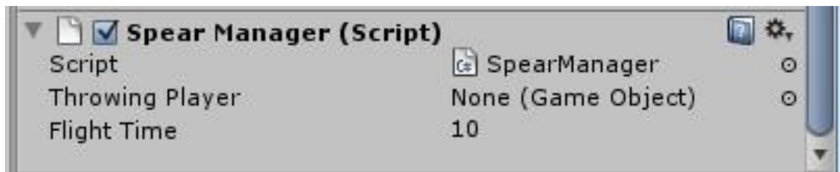
1. Thrown
    a. This is an internal flag, leave it unchecked
2. LeftHand
    a. This is a reference to the LeftHand sub object in the Player Prefab.
    b. Drag the LeftHand sub object from the Player Prefab in the Project Window and drop it onto this field in the Inspector.
3. Throw Force
    a. This is how hard to throw the spear. Feel free to play with it.
    b. I've used 20,000
4. The Spear Throw Model

a. This is the Spear Prefab. Just drag and drop it from the Project Window onto this field in the Inspector

5. Reload Timer
    a. This is how long it takes to reload after throwing a spear.
    b. I use 3

Spear Manager Variables:

1. Throwing Player
   a. This is set by Spear Controller when a new spear is instantiated
   b. It's not currently used, but the purpose is if you ever want to apply rewards for to a player when they deal damage or death, then this tells us what player to reward
2. Flight Time
   a. If the spear never hits anything then we destroy the spear after a certain time
   b. I use 10



That's it, you should now be able to throw a spear around too.

Please see my Conclusion Page at the very end before closing this tutorial.

Appendix B – Advanced Concepts Visited

So I thought it might be appropriate to discuss some advanced concepts that can sometimes be very confusing and overwhelming. These commands (and others) have their purpose and can provide great control over how you manage networking in your game. However they also require a little care and attention. We'll discuss them briefly here, then you can do your own research and maybe you'll write an advanced tutorial for the rest of us:

Buffered RPC commands

Buffered commands such as RPCMode.AllBuffered actually save RPC commands on the server and then execute those commands in the future whenever a **new** player/client connects. This seems useful at first to ensure a proper level load perhaps, but can become confusing if you plan to use it for creating dynamic objects in your scene. In my opinion its best left for loading the level and any dynamic but mostly static objects that once created will rarely be deleted or destroyed. This is because the code to remove them from the buffer can be difficult to keep track of depending on how many places in your code you might do something, and the overhead in updating the buffer often is probably not that efficient. Therefore, I wouldn't use it for highly volatile things like weapons fire, but you might use it to spawn vehicles which don't change too often, they just move to a new location within the scene.

The other issue with buffered requests is that you can't update a request and a value. For example in the case of the Balls, we probably could have used AllBuffered to create the balls, and then they'd automatically be instantiated on any new client that joins. However, we toggle the ball renderer and collider on and off at different points. If we constantly added those commands to the buffer too, it would just grow the buffer too large. If we constantly deleted and created a new ball in the buffer that would be an inefficient use of the buffer causing extra processing in the game. Instead, I think managing the balls ourselves using callback events as we did is a better approach.

On the surface buffers might seem great because you can remove all objects for a player, or all objects for a specific viewID. However, what I've struggled with is wanting to remove just certain objects, or better yet update an object in place (the ball sample mentioned above). I've found it too inefficient to manage the buffer this way. One way around this is to use different networking groups (discussed below), but that comes with its own complications.

I've personally felt better to keep track of specific objects at either the GS level or the player level, knowing anything managed by the player lives/dies with the player and anything that needs to persist even if the player leaves should be managed by the GS. You'll notice the balls are managed by the GS if you look closely and the spear too is managed by the GS. This allows the player to disconnect and the object he threw continues to travel and cleanup as expected.

A few commands to consider if you want to play with buffers include:

1. Network.RemoveRPCs (…)
    a. With this you can remove all Buffered commands for a specific player
    b. … for a specific networkView.viewID
    c. … for a specific player and a specific Network Group
2. Network.DestroyPlayerObjects (…)
    a. Use with caution
    b. If a player instantiates something then you call this you'll end up destroying the player and whatever the player instantiated. So if you instantiate the spear with the player then call this and the player logs out you also lose the spear
    c. You might use it to instantiate things that are part of the player and should leave with the player
3. Network.Destroy (…)
    a. Just destroy the specific GameObject but leave anything that was instantiated by that object. This can be ok as long as that's the behavior you want, and any game specific logic to deal with those instantiated objects, including updates and cleanup is part of the instantiated object itself, and not code that was part of the deleted parent.
    b. There's no in-between this way. Either all are destroyed or non are destroyed

I think the only place I used buffered commands in this eBook was to create the level itself which is all static.

Networking Groups

Network Groups are a way to control sending messages just to a specific group. For example, if the game will have teams and you want to just send RPC calls to other team members, using groups is one way to accomplish it. Another benefit to groups has to do with cleaning up Buffered requests as you can cleanup all requests for a single group. Therefore you might use groups to organize different events that occur and allow all clients to receive the group message, or just s pecific clients such as team members.

You also improve networking overhead by not sending network requests to clients that would just ignore it, so if you planned to send a team request to all players but would accept or ignore the request based on a passed in variable such as teamid, this is less efficient as it consumes more networking overhead that is very precious. Instead think about using groups.

There are two steps to using groups. First you need to enable a client to receive requests for a particular group. Otherwise they only receive group=0 by default. It's ok to enable a client to receive multiple groups at one time. To set a particular group for receive use:

Network.SetSendingEnabled(group id, on/off bool);

Next is to send a command to a specific group. Since we send commands with RPC's, the way we direct an RPC to a particular group is to use the networkView.group command. networkView.group is an int and to set it you use:

networkView.group = {group #}

If you want to send an RPC to all players then use the default group=0

SetLevelPrefix

Using SetLevelPrefix is a nifty technique if you have a multi-player game that can have players switch levels at different times, independent of other players. If you think about what happens for a moment, all players are still connected, and so far we've assumed both players are in the same level. Therefore, we want them to receive all updates. However, if a player switches levels, you don't want bleed-through of updates meant for the other level. For example if a player throws the spear we instantiate a new spear, but we only want to instantiate that spear for players on the same level. We don't want an arbitrary spear to appear for a player that might have switched levels. To let Unity3D do this automatically for you just set the Level Prefix before you switch levels. This will ensure you don't receive any network updates meant for activity occurring on another level. To set it just use:

Network.SetLevelPrefix({integer value});

Appendix C – Try This

1. Add code to properly update the Player State for the spear when a new client connects.
2. Create a scoring system and apply damage for hits
    a. Try 1 point for each spear hit
    b. Try 4 points for each ball hit
    c. Play to 12
    d. Add a game over page
        i. Show the score
        ii. Highlight who won
        iii. Return to Master Game Server Lobby
3. Add a bunch of dynamic obstacles such as columns to navigate around
    a. Use AllBufferd to create the columns
    b. Set the columns to take a damage of 10
    c. Set the columns to only receive damage from a spear, not from a ball
    d. When damage reaches 10 remove the column, and correctly remove the entry in the GS Buffer
4. Add some particle systems
5. Add some sound
6. Or create an entirely new game from scratch using some of the ideas learned here

Appendix D – Object to Script Associations

MasterGameServerLobby Level

MasterServerMenu (Object in Hierarchy)

a. NetworkView
b. NetworkMasterServer
c. NetworkLoadLevel

TestGameLevel Level

PickupSpawn (Object in Hierarchy)

a. BallSpawnManager

PlayerSpawn (Object in Hierarchy)

b. InstantiatePlayer

Player Prefab (Prefab Object in Project Window)

c. RigidBody
d. NetworkView
e. NetworkRigidBody
f. PlayerSpawn
g. BallController
h. SpearController

BallPickup Prefab (Prefab Object in Project Window)

i. RigidBody
j. NetworkView
k. NetworkRigidBody
l. BallManager

Spear Prefab (Prefab Object in Project Window)

m. RigidBody
n. NetworkView
o. NetworkRigidBody
p. SpearManager

Conclusion

So we've covered just about everything I've set out to cover and there's actually still a lot to learn about network programming, but I hope this eBook has provided you with enough of the foundations to get you started creating some really awesome game. Please email me if there is anything I can do for you, and visit my website at http://www.3dmuve.com. Feel free to use my forums to ask questions, or leave comments on my blog.

You can reach me at:

Laurence (Quadgnim) Grant
larry.grant@3dmuve.com
@quadgnim

Thanks All,

Larry