Mike Hergaarden - Last edit: 14-08-2011

Content

About this project
About the author
How to use this project?
- Tutorial 1: Connect & Disconnect
- Tutorial 2: Sending messages
- Tutorial 2A: Server plays, client observes, no instantiating.
- Tutorial 2B: Server and client(s) play, with instantiating.
- Tutorial 3: Authoritative servers
- Tutorial 4: Manually instantiating networkViewIDs

One last practice to master is to manually instantiate network objects, specifically the networkView IDs. To be able to send the first bits of data in a game you need one functioning networkView ID; that is, it needs a valid ID. A networkViewID gets a valid ID when you..

Examples
- Example 1: Chatscript
- Example 2: Masterserver example
- Example 3: Lobby system
- Example 4: FPS game
- Example 5: Multiplayer without any GUI

Further network subjects explained
Adding multiplayer to your own game
Tips

About this project

The goal of this project is to show you everything there is to Unity Networking, this project does not require any previous networking experience. There are three main parts: Tutorials, Example reference projects and additional documentation. This project is all you need to be able to create a networked game and provides you with resources to integrate Unity networking in your own games much quicker and easier.

There are three main tutorials covering the basics of the Unity networking system. After finishing the tutorial I'll introduce my Networking class that you can reuse in your own projects. The examples consist of a Chat example, a Lobby example, a FPS game example and finally an interesting auto matchmaking multiplayer setup. Finally at the end of this document there is some extra documentation about networking issues/tips that were not addresses by the tutorials and examples.

You are advised to read this document from top to bottom, but if you are picking things up quickly you can have a look at the examples yourself and fall back to this document whenever you need more details.

You can discuss this project on the Unity forum topic. Feel free to ask for extra assistance here or leave feedback. I am subscribed to this topic.

About the author

M2H GAME STUDIO

This tutorial is written by Mike Hergaarden (Nickname "Leepo") from M2H. We've been using Unity since about Unity 2.0. We love multiplayer (co-op!) games, that explains why almost all our games feature multiplayer.

Our most notable multiplayer games currently are Crashdrive 3D, Verdun Online and Cubelands.

This project will be maintained for at least the Unity 3.X branch. Feel free to send me feedback/bugs. Is something wrong? did you expect some other features in this project? Let me know!
Have fun adding multiplayer to your games!

Mike Hergaarden

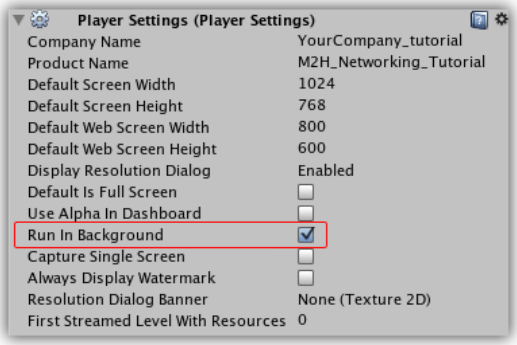
mike@M2H.nl

How to use this project?

This document is bundled with a Unity project. It's assumed you have already made yourself familiar with the Unity editor and basic scripting, if not: check out some basic Unity (video) tutorials first.

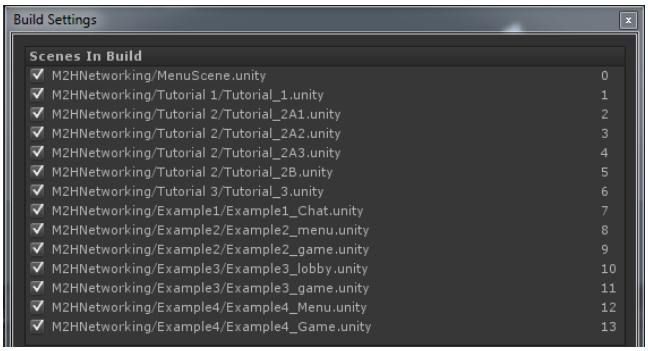
Multiplayer isn't much fun to debug since you need two instances of your game running (server and client). It's often even required to have two clients as you'll also need to check how, and if, clients see each others updates. During this tutorial you're advised to run the server in the editor game view and a client in a webplayer or standalone.

In case you want to use the tutorial assets in a project of your own, do mind that one projects setting has been modified in this tutorial project. For your own projects ensure the option "Run in background" has been turned on to be able to run a server in the background without it "going to sleep". This will keep the servers game running in the background, otherwise you would not be able to connect to your server when running the client in the foreground. You can find this option at "Edit → Project settings → Player". This setting is not required for clients. Even better is to set it in code when starting a server ("Application.runInBackground = true;").



All files in this tutorial are written in both JS and C#. While I *greatly* prefer C# above JS, I have chosen to stick to JS as main language in the documentation etc. However, I do encourage you to **ditch** JS and go the C# way! The C# files are saved next to the JS files.

Important
For the examples and tutorials to properly work you need to verify the build settings as shown below. Sadly, the Unity asset store does not save the build settings or any other project setting. To save you the hassle I've made a tool to set the right buildsettings, simply run it once to fix the build settings (this will overwrite the current selected scenes). Run *M2HNetworking ->Reset buildsettings* from the Unity menu to fix this.

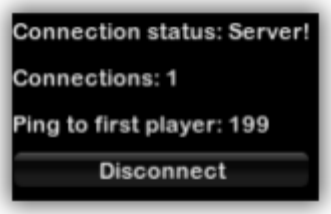


Tutorials

Tutorial 1: Connect & Disconnect

Let's get you going!

1. Open the very first tutorial scene: The scene can be found at "**Tutorial 1/Tutorial_1**". This scene consists of one camera, one gameobject with the connect script attached to it and one gameobject to display the scenes title.
2. Build and run a webplayer
3. Start the scene in the editor as well and click on "Start a server" (using the default values for IP and port)
4. Click on "Connect as client" in the webplayer.
5. You should now see "Connection status: Client!" and "Connection status: Server!" on your two instances. Congratulations, you've connected!



This was all very easy; luckily the code is not much harder. Have a look at the file "**Tutorial 1/ Connect.js**" in your favorite editor. All code that has been used in this tutorial can be found in the OnGUI(); function, have a look at this function and ensure you understand how it works. The code should be pretty much self explanatory, however, we'll deal with the most important parts briefly.

The two variables at the top of the script (connectToIP and connectPort) are used for capturing the input from the GUI field, these values are used when pressing the button to connect. The GUI function is divided in four parts; For servers, connected clients, connecting clients and for disconnected clients. We use the provided networking status "Network.peerType" to check our current connection status. We call the Network.Connect function to connect clients to servers, this function takes IP, port and optionally a password as arguments. To start a server a similarly easy function is called: Network.InitializeServer. This takes a port, a maximum allowed number of connections and a bool for NAT setting as argument. Do note that you can always lower the maximum numbers connection on a running server, but you can never set it higher than the value you used when initializing.

NAT

> **N**etwork **A**ddress **T**ranslation is useful for clients behind a router (inside a LAN) to connect with other users on the internet. This networking demo should only be run inside a LAN; you will probably not be able to connect to your friends computer. In a later section you'll see how to connect to other clients from other networks, it's not much extra work but only a distraction for now. Network.InitializeServer accepts a NAT setting, ignore it for now and set it to false.

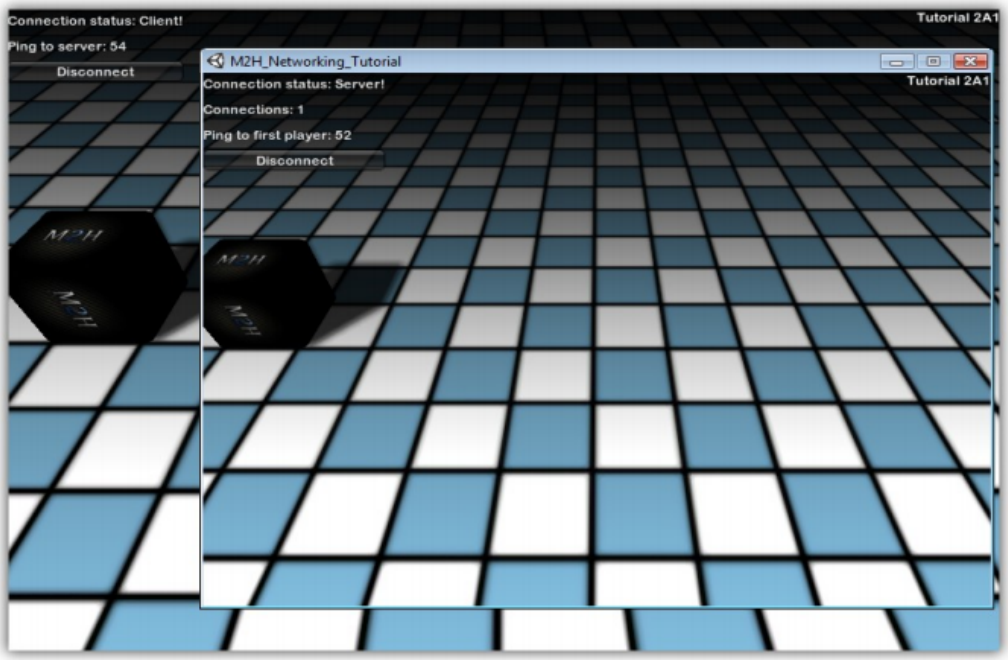
Now, on to the last bit of code in the file; The +/- 10 functions that are automatically called by Unity. It is important to note that you **don't** need any of these functions anywhere in your code if you don't want to use them, you can remove them all and this demo will still work. The first 5 client and server functions should be very self explanatory; They are called **only** on the client(s) or **only** the servers. Only the OnDisconnectedFromServer is called on both clients and servers. If you want to use the parameter passed by the functions, checkout the unity manual entries for these functions.

The last three functions are different. **OnFailedToConnectToMasterServer** is called on a client(or server) when you somehow can't connect to the Unity master server, more details about the master server will follow later. **OnNetworkInstantiate** is called on instantiated objects, we'll also have a

look at this later. **OnSerializeNetworkView** is the first of the two methods for us to send messages across the server and clients. **R**emote **P**rocedure **C**alls are network messages/functions you define yourself. In the next tutorial we'll have a look at both serialization and RPCs.

To conclude this tutorial have a quick look at the Network documentation (“Messages Sent”, “Class Variables” and “Class Functions“) here: http://unity3d.com/support/documentation/ScriptReference/Network.html
This page gives you a quick reference to all Unitys networking functions, remember the link as reference. We've already briefly discussed about 75% of the information over there!

Tutorial 2: Sending messages

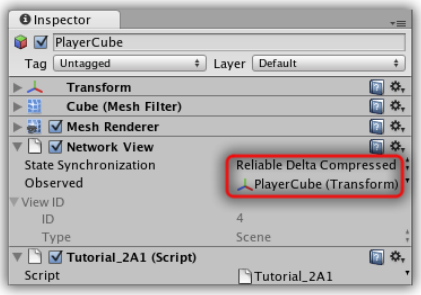


Our very first multiplayer scene..there's just one real player though!

Tutorial 2A: Server plays, client observes, no instantiating.

Tutorial 2/Tutorial 2A1

Don't let the title scare you, just open the scene "**Tutorial 2/Tutorial 2A1**" and have a look around. The simple connection code from tutorial 1 has been attached to the "**Connect**" gameobject. Furthermore the "**PlayerCube**" object has the "**Tutorial 2A1.js"** script and a "**NetworkView**" component attached. Every object that sends or receives network messages requires a NetworkView component. You could just use one networkview for your entire game by referencing it from a global script, but that wouldn't make sense; in general just add a networkview per object that you want networked. Adding a networkview to every player and to your GameManager code would be useful. Adding a NetworkView to every bullet is crazy though; you'll be running out of so called network "view id's".



Run the 2A1 demo with a server and client. The client(s) will be able to look at the server moving the cube around. All magic to this movement is thanks to the observing networkview and the movement code. Have a look at the ` **Tutorial 2A1.js"** script attached to the cube. This code is only being run on the server (hence the Network.isServer check): When the server uses the movement keys; it will move the cube right away.
Now how do the clients know about the servers movement? Have a look at the NetworkView attached to the cube. It is *observing* the "transform" of the cube, meaning unity will automatically send the transforms information over (the position, rotation and scale Vector3's). It only sends the information from the server to the clients(and not the other way around) because the server is the

owner of this NetworkView (client's don't send their information; they know they can only receive). The server is automatically owner of objects that are 'hardcoded'in a scene. Later on you'll discover how clients can be owner of objects they instantiate over the network.

Using the "Observed" property helped us to quickly enable networking of the player movement. However the "observed" property isn't very smart: It's OK for transforms but even has an overhead there as it also sends over scale for example. Except this simple example, we will not be using "Observed" to automatically set up our networking messages as you really need to manually define your network traffic to get the most out of it. Don't completely forget the Observed property though; we need it at 2A1.

By the way, don't worry about laggy movement in the first three tutorials, we'll get to that once we have covered the basics.

Let's look at the rest of the NetworkView options to completely wrap up this subject. The PlayerCube's networkview State synchronization[1] option has been set to "Reliable compressed". This means it will only send over the values of the observed object if the values have been changed; If the server does not move the cube for 15 minutes, it won't send any data, smart éh! The "Unreliable" option will send the data regardless whether is has been changed or not. Finally setting "State synchronization" to "Off" will obviously stop all network synchronization on this networkview. If your networkview is not observing any object, it will not send any data and the synchronization option can (but doesn't have to) be set to "off". If you wonder why you'd use such a networkview; "**R**emote **P**rocedure **C**alls" need a networkview, but don't use the "state synchronization" and "observed" option, both are ignored for RPCs. You can use RPCs on a networkview that is observing something, there are no conflicts. RPC's will be introduced in tutorial 2A3; they are basically custom defined network messages(/calls).

Tutorial 2/Tutorial 2A2

What if we want to move the cube in the y-direction, or if we want more control over what's being synchronized by unity? Open and run "**Tutorial 2/Tutorial 2A2**". The 'game' should play exactly the same as 2A1, but the code running in the background has been changed.

The networkview attached to the PlayerCube is now observing the "**Tutorial 2A2.js"** script. This specifically means that the network view is now looking for a "**OnSerializeNetworkView**" function inside that script. For scripts, this function defines what is being observed. Have a look at that function: We now explicitly define what we want to synchronize. You can use this to synchronize as much as you want, and again, only changed values are actually being sent when using "Reliable delta compressed. The OnSerializeNetworkView function always looks as strange as this. This function is used to send and receive the data, Unity decides if you can send("*istream.isWriting"*) by checking the networkview owner, otherwise you'll only be able to receive(the "*else"* bit). The server is always the owner in this case as it owns all networkviews that are 'hardcoded' in the scene.

[1]State synchronization, which method to choose? See: http://unity3d.com/support/documentation/Components/net-StateSynchronization.html

Tutorial 2/Tutorial 2A3

There's one last method to send messages which I love the most; **R**emote **P**rocedure **C**alls. I've mentioned them before, fire up "**Tutorial 2/Tutorial 2A3**" to see what it's actually about. Again, this demo works exactly like the last two but uses yet another way to send messages. The networkview is no longer observing anything (and the state synchronization option has therefore been set to "off"). The mojo is in "**Tutorial 2A3.js"**, specifically this line: *networkView.RPC("SetPosition", RPCMode.Others, transform.position);*.

A RPC is called by the server, with as effect that it requests the other clients to call the function "*SetPosition*" with as parameter the servers *transform.position* (e.g.: 5.2). Then *SetPosition(5.2);* is called on all clients. This is how the movement is processed:

1. The servers player presses a movement key and moves his/her own player (lines 14-18)
2. The server checks if its position just changed by a minimum value since the last networking update, if so send an RPC to everyone but itself with as parameter the new position. (Lines 20-25)
3. All clients receive the RPC SetPosition with the parameter set by the server, they execute this code in "their own world".
4. The cubes are now at the exact same position on server and clients!

To enable a function to act as RPC you need to add "@RPC" above it in javascript (or [RPC] in C#). When sending an RPC you can specify the receivers as follows:

RPCMode.Server[2]	Only send to the server
RPCMode.Others	Send to everyone, but the caller itself
RPCMode.OthersBuffered	Send to everyone, but the caller itself. Buffered.
RPCMode.All	Send to everyone, including the caller itself.
RPCMode.AllBuffered	Send to everyone, including the caller itself. Buffered

Buffered means that whenever new players connect; they will receive this message. A buffered RPC is for example useful to spawn a player. This spawn call will be remembered and when new players connect they will receive the spawn RPC's to spawn the players that were already playing before this new player joined.

Be proud of yourself if you still roughly understand everything so far; we've finished the basis and hereby covered most of the subjects. We now just need to go into details.

[2]There's one annoying "feature" (**BUG!**) here. You can currently not send RPCs to yourself: A server cannot use RPCMode.Server. See Tips section at the end of this document for more information

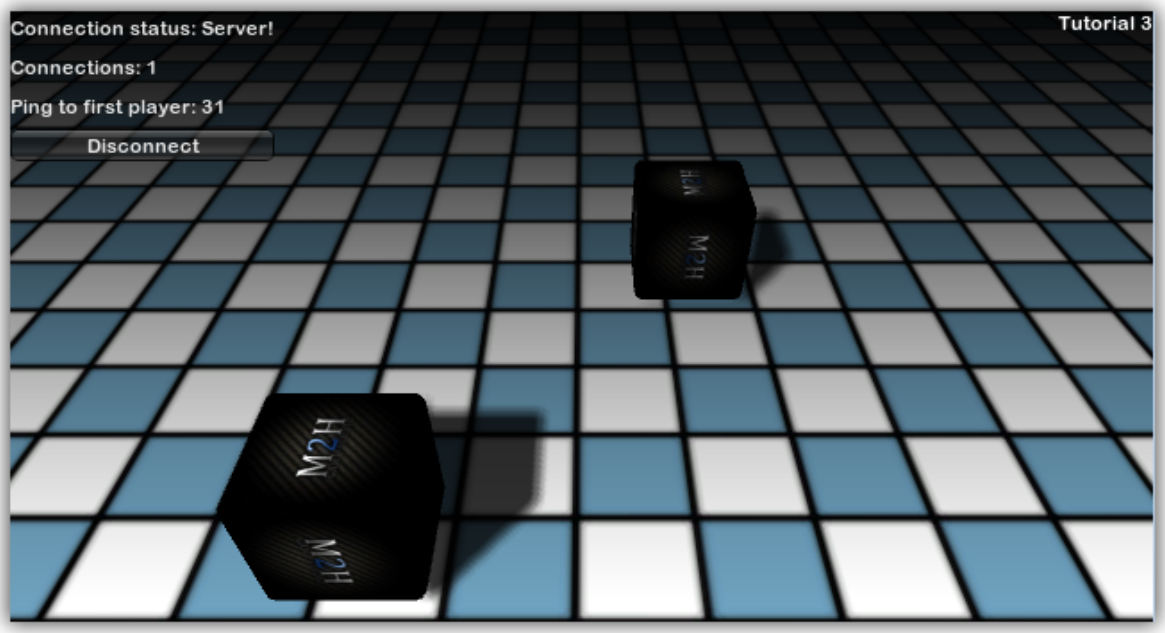
Tutorial 2B: Server and client(s) play, with instantiating.

We're going to get dirty with some details that could form the basis of a real FPS game. We want to enable multiple players. For this purpose we will be instantiating players when they connect, instead of having a playerobject in the scene. Open the scene "**Tutorial 2/Tutorial 2B**" with one server and one client. Have a walk with the two players to verify the movement of both is networked properly. The server (in the editor) should only be able to move it's own cube and the client should only be able to move his cube, the second cube.

The **PlayerCube** has been removed in this scene, instead **Spawnscript.js** has been added to the new **Spawnscript** gameobject. When a player (either server or client) starts the scene, the Spawnscript will instantiate the prefab that we've specified in the script. Instantiate takes position, rotation and group arguments. We'll copy the position and rotation of the Spawnscripts object and use 0 as group[3] (feel free to ignore it for now). On disconnection the spawnscript will remove the instantiated objects. The one who calls a Network.Instantiate is automatically the owner of this object. That's why the movement controls of the playercubes work out of the box: it checks for the networkview owner to decide whether one can control a player. "**Tutorial_2B_Playerscript.js**" uses the movement code from **Tutorial 2A2** with the as difference that only the objects owner input is captured.

[3]See the "Further network subjects explained" section for more information about groups

Tutorial 3: Authoritative servers



Yay; real multiplayer!

The server configurations of the last few examples are what's called “non-authoritative” servers; There was no server-side authorization over the network messages since the clients share their position and everyone accepts (and “beliefs”) these messages automatically. In a multiplayer FPS you don't want people editing their networking packets (or the game directly) to be able to teleport, hovercraft etcetera. That's why server are always authoritative in these games. Setting up an authoritative server does not require any fancy code, but it requires you to design your multiplayer code a bit different. You need the server to do all the work and/or to check all the communication.

Let's first sit back to think what changes the last (2B) example would need to make it authoritative. First of all; the server needs to spawn the players, the players cannot decide when they want to be spawned and where. Secondly, the server needs to tell everyone the correct position of all player objects, the players can't share their own positions. Instead, the server needs to do this and for this reason the client is only allowed to request movement by sending his/her desired movement input.

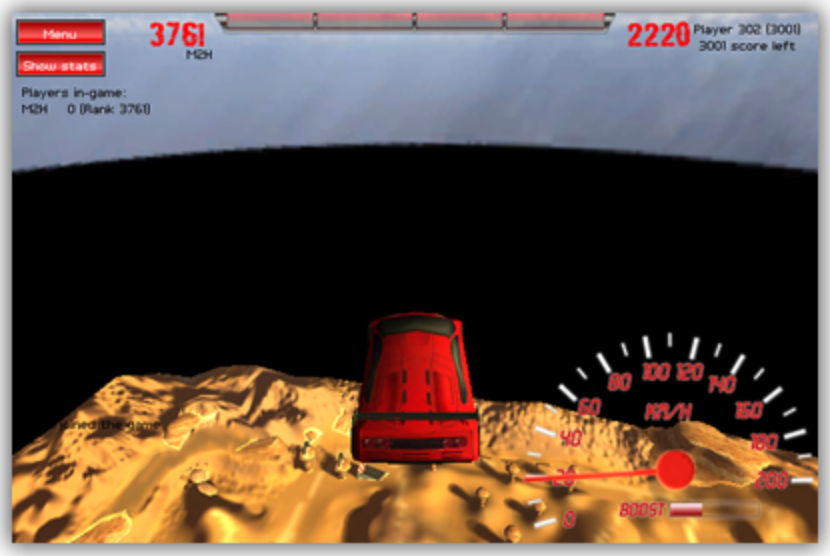
We will send all clients movement input to the server, have the server execute it, and send the result (the new position) back to all the clients. Have a look at the Tutorial3 scene. Again, it'll play just like before, but the background has changed. The movement will probably feel quite a bit more laggy than before, but this is of no importance right now.

No new scripts have been added since the last example, only the playerscript and the spawnscript have been changed. Let's start with the “Tutorial_3_Spawnscript.js”. Clients no longer do anything here, the server starts a spawn process when a new client connects. Furthermore the server keeps a list of connected players using their playerscripts to be able to delete the right playerobject when the player logs off. The spawnscript would have been a 100% server script if it wasn't for the OnDisconnectedFromServer” which is also called on clients.
Moving on to “Tutorial_3_Playerscript.js”, this script is now not only run by the objects networkview owner. Since the server instantiated all objects, it is always the owner of

all networkviews. Therefore we now use our own owner variable to detect what network player “virtually” owns this object according to our own game logic. The owner of the playerscript sends movement input to the server. The server executes all playerscripts to process the movement input and actually move the players. We now have a fully authoritative server!

To get back to the subject of lag: in the previous examples the players cube would move right away when pressing a movement key, but in this authoritative example we need to send our request, wait for the server to process it and then finally receive the new server position. While we do want the server to have all authority, we don't want the clients waiting for the servers response too long. We can quite easily fix this lag by **also** having the client calculate the movement right away, the server will always overwrite its calculated movement anyway and is still authoritative. This is quite easy to add. In “Tutorial_3_Playerscript.js” simply have the client execute “*SendMovementInput(HInput, Vinput);*”where you are sending the movement RPC (uncomment line 56) . Then make sure that the SendMovementInput RPC call actually affects the client by updating the (server) movement code in the bottom of the Update() function; also run it on the local player too by adding “*|| Network.player==owner){*" in the IF statement (see line 64). These two edits will now make sure the clients movement is applied right away, but the servers calculations will still be ultimate in the end.

After applying the client “prediction” the movement will *still* look a bit laggy, especially if a client is watching a different client move. To improve this check out line 100, here's a snippet to “merge” the clients current position and the servers position, with the servers position having more weight. You can take this a step further by saving the real server position in a variable and “lerp”(See Vector3.Lerp) to this position in the Update loop instead of only Lerping once in every OnSerializeNetworkView call (which is executed far less than Update). What this does is that is slowly blends the movement instead of just static stuttering movements.



“Technically” you can make this happen ;).

Do note that you don't always need to make your multiplayer games authoritative. Take our “Crashdrive 3D” game for example, it was made non-authoritative. A player could *possibly* change it's cars position maliciously; but who would really care? It could possibly affects a players highscore; but that's being checked for dubious entries already since you can 'hack' your highscore via memory even more easily. Long story short: Consider whats really vital to make authoritative. Also do not forget that even an authoritative server itself can still cheat. A authoritative setup is always very good practice though!

Tutorial 4: Manually instantiating networkViewIDs

One last practice to master is to manually instantiate network objects, specifically the networkView IDs. To be able to send the first bits of data in a game you need one functioning networkView ID; that is, it needs a valid ID. A networkViewID gets a valid ID when you..

1. ..use Network.Instantiate *(but I advice you not to use it)*
2. ..save it in the scene (e.g. have a GameManager object with one networkView attached)
3. ..manually instantiate a networkView and assign it an ID. You need 1 *or* 2 to be able to do this as every player must receive a message and set the right viewID(!)

Using manual allocation instead of Network.Instantiate() gives you a lot of more power, plus it forces you to write a clear player handling class and I therefore highly recommend this.

Tutorial 4 is all about the Game manager script (GameObject **'code'** -> **Tutorial_4_GameManager.js**). The player movement here is based on 2B and is not of interest here.

The game manager maintains a list of all players and their transforms. When a server starts and when clients connect they will spawn a player for themselves. The 'AddPlayer' and 'SpawnOnNetwork' functions are used for this, they are called on the local player right away. RPCMode.OthersBuffered is used to send the message to all other players. The buffered property makes sure any future clients will receive the messages automatically.

Instead of using OthersBuffered we could also simply use RPCMode.Others to send the AddPlayer and SpawnOnNetwork messages only once, without buffering it. When a new client connects the server should then send all playerdata manually.

My Networking class: MultiplayerFunctions

During the last year I've made a couple of Unity multiplayer games. With this experience I created a reusable class that wraps a few networking functions to simplify some common tasks. To aid your multiplayer development I hereby share this networking class.

Main features:

- Handle network connection testing for you out of the box
- Master server: Automatic setup and easy functions to use it
    - *(You can optionally specify a custom master server)*
- Connect/StartServer wrapper functions
- Allows re-trying failed connection attempts (fallback on the default server port without NAT)
- Reconnect to the last host (e.g. if you need to temporarily disconnect)

How to use it

1) Add the MultiplayerFunctions.cs script to all your scenes
Attach the script to a gameobject. Make sure the script is available in all scenes where you want to use it. You can add it to your preloader and enable "*DontDestroyOnLoad (this);*" in the Awake function to ensure it's available in every scene of your project.

2) Customize the three settings in MultiplayerFunctions.cs
masterserverGameName: **[!]** used by the masterserver ("YourGameName")
defaultServerPort: the default server port. (20000)
connectTimeoutValue: connection timeout value (10 seconds)

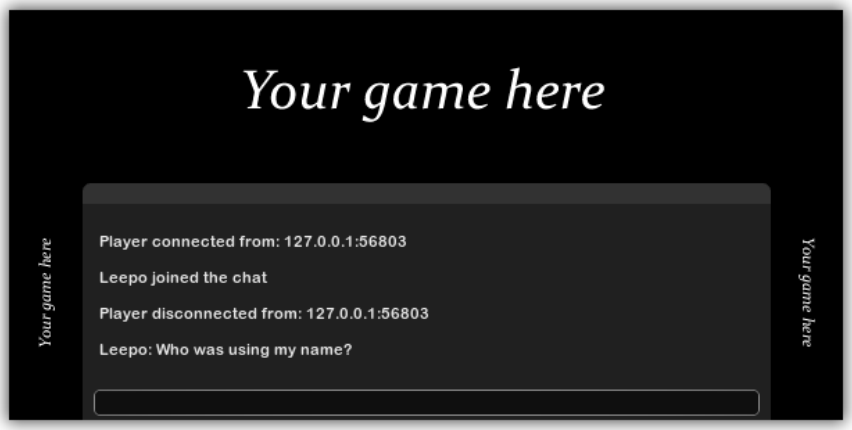
3) Implement the functions you need
See the top of the source file to get a quick overview of all available functions.
See the examples later in this project for various usages of this script.

This class can be found at *Plugins/MultiplayerFunctions.cs*. Since it's placed in the plugin folder you can use it in both JS and C# (Yep there's no JS version, *DIE JS*!). This class will be used in all following examples, I encourage you use it in your own projects too and change it to your own needs. Using a wrapper instead of the Unity functions gives you more power and allows future customizations.

Next to the multiplayer functions class I added GameSettings.cs. The purpose of this class is to easily save multiplayer (or game) settings in the main menu and being able to read them in-game. This is a static class so you don't have to add it to a scene/gameobject. Example two showcases the use of this class. Instead of GameSettings.cs you could also use your own persistent object or PlayerPrefs to save data between scenes.

Examples

Example 1: Chatscript



The scene "Example1/Example1_Chat" is nothing more than the connect script seen in tutorial 1 combined with a new chat script. Adding a chat to you games is ridiculously easy; you can re-use this chat script anywhere with next to no modifications required; just make sure to hook up the player names. The chat currently holds at maximum 4 lines. When you modify the code to show more lines, you could use yield or a coroutine to delete/"fade out" older messages. The server tracks a list of players; in a real game you'll probably want this list in a more central/general script since you'll only need one player list and hiding it in the chat script isn't the best place.

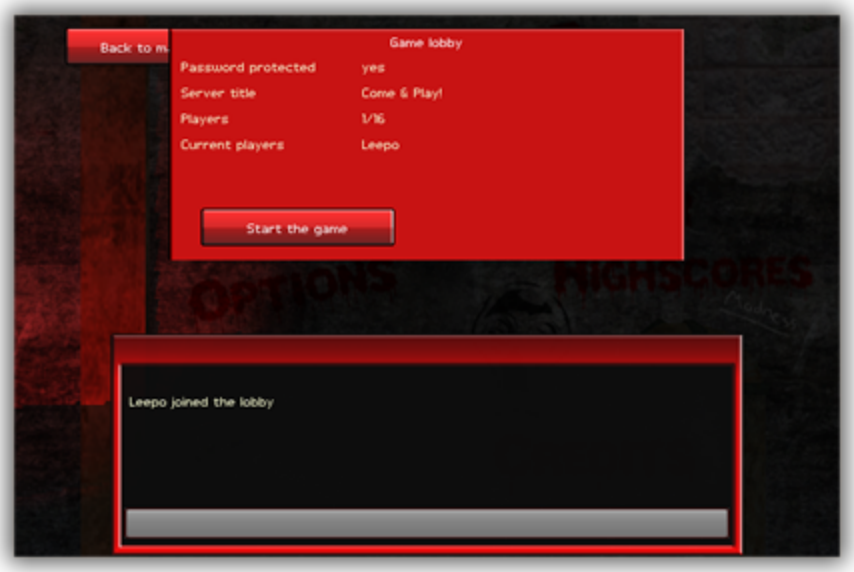
Example 2: Masterserver example

A very similar masterserver implementation in our game Crashdrive 3D

Open the scene "**Example2/Example2_menu**". This example showcases how you can use the master server to show all current running game sessions. The GUI as shown in the example is a basic barebone but it does feature flexible functionality: Under JOIN the player can see a list of all active hosts, refresh this list or enter an IP and port for a direct connection. The quick play option

can be used by player that want to join the first game session available ("Join random game"), this option is only shown when one or more hosts are available. Furthermore under the HOST option a player can start a host
The 'game' in this example only shows you the connection status of the server and clients. You could easily replace the game scene with any other scene and the networking will work right away. You'll only need to set "*Network.isMessageQueueRunning = true;*" since we disabled it in the menu scene to prevent strange things from happening in the main scene when a client joins a game that's in progress. You'll also need to remember to register the game to the master server once the server has started the game scene. Last but not least I say customizing the multiplayer GUI is essential, as this GUI is a pure barebone only.

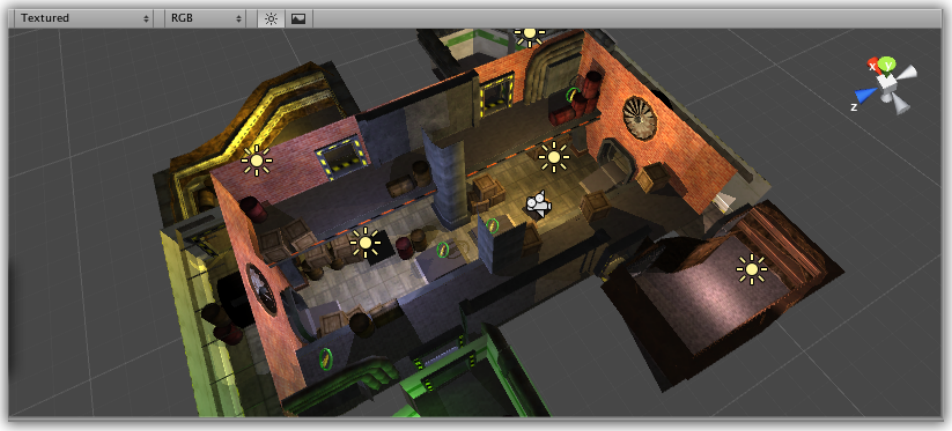
Example 3: Lobby system



A very simple lobby system implementation in "Surrounded by Death"

"**Example3/Example3_lobby**": This example is very much like the first example, except that it feature's a pre-game lobby with optional passwords. Games are only shown in the master server list as long as they are in lobby stage, games that have started are removed from this list right away. Again; you can easily use this code for your networked games by copying the lobby scene and adjusting it to your needs, just don't forget to enable the message queue in your game scene.

Example 4: FPS game

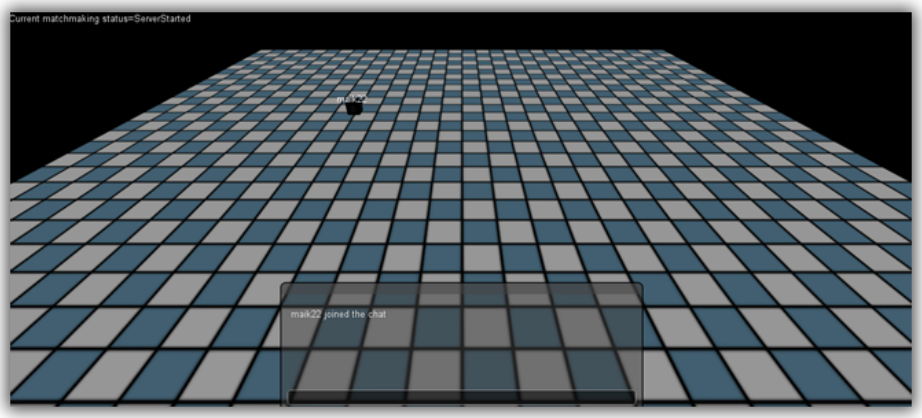


Since most people want to create a FPS game I decided to provide a FPS game base. This FPS example is non authoritative, so it's up to you to redesign the game for a secure authoritative setup ;)!

This example uses the Example 2 main menu script to set up a connection in the main menu. In-game multiplayer features are: chat, scoreboard, movement, shooting, pickups. If you want to use this as a base for your FPS game, possible additions could be:

- Authoritative movement: To prevent cheating
- Animated characters: synchronize the animations or have the clients “calculate” the right animation to play
- Weapon switching
- Not (completely) relevant to multiplayer: Crosshairs, improve the GUI, spectator mode, game rounds and round limits, etc.

Example 5: Multiplayer without any GUI



Crashdrive was our first Unity game, we made it back in 2008. Begin 2010 we relaunched it on several big game portals. In this second version we decided to drop the multiplayer join/host GUI: the majority of the players don't get these kind of GUIs. I developed a system where the player just starts playing right away and while playing the game will do automatic matchmaking. This way players can enjoy multiplayer without any setup hassle.

Example 5 features this same setup. There is just one scene, the game scene. You start playing by simply running the game scene and walk around. In the background the networking code is trying to join all available games from the master server. If none are available the game will host a server itself, making it possible for other players to join. If no players join within X seconds, the game will retry to connect to all available servers again and the cycle repeats itself as long as the player is playing solo.

This setup will require some extra planning for all your games events that should be multiplayer: Depending on your status; Disconnected,Client or Server. Any of those three states are possible at any time(!). This means that the code will have to check if you're a server or client to send a network mesage *or* to execute a local message when you're disconnected. Even so, I didn't have much pain adding gameplay (events) to Crashdrive.

Further network subjects explained

1. Unity editor options related to networking

"Edit → Project settings → Network"
Sendrate: This affects how many times per second the observing network messages are send (Unreliable or Reliable delta compressed). This does not affect RPC messages. To minimize traffic try to set sendrate as low as possible until it visually disrupts gameplay.
Debug level: Changes how much network debug information the editor log will show.

"Edit → Project settings → Player"
Run in background: Yes/No. When running a server it is required to run in background to keep the network communication running.

2. Limiting traffic: Scoping and groups

You can greatly improve network performance by limiting the amount of data you sent. In a big game world players do not need to receive every bit of information. What happens at the far end of the world is irrelevant to most players. There are two techniques you can use to have the Unity server not send information based on groups or players.
First of all the **networkview** component has the SetScope function:

```
function SetScope (player : NetworkPlayer, relevancy : bool) : bool
```

This is of course true by default for every player. You can set it to false if this networkview is far away from 'player' and this player will no longer receive messages that are sent using this networkview. **However**, this only works for the observe property of the networkview. Meaning this does not work on RPCs which is a real pity.

Network functions:

```
static function SetReceivingEnabled (player : NetworkPlayer, group : int, enabled : bool) : void
static function SetSendingEnabled (group : int, enabled : bool) : void
```

These two functions can be used to limit sending/receiving based on network group. You could for example divide your games level/map in 32 tiles and only send/receive information to players about the 8 tiles around the player and its own tile. Here the pity is that you can have 32 groups at max, which isn't a big issue for FPS games and the like, but it's really one of those many limitation that keeps us from creating a MMO with the Unity networking library ;).

3. Securing the network connection

Adding AES encryption, CRCs, randomized encrypted SYNCookies and RSA encryption sounds really hard right? Luckily it's just one code call to add all this security to your game:

```
function StartServer ()
{
    Network.InitializeSecurity();// Does all our magic!
    Network.InitializeServer(32, 25000);
}
```

Just make sure to call InitializeSecurity(); before initialising the server. The security adds up to 15 bytes per packet. Note that my MultiplayerFunctions class includes this security in the StartServer function.

4. Anti cheating

Even with the security layer in place from the previous section it's very easy to cheat. The security only applies to the network messages, one could still easily change it's score in the memory. You should always assume the worst case scenarios when designing your game. Assume the player know as much about the code as you do, and that they can edit the network packets in the worst possible values. So always check all values you receive. You don't really need any special code to combat cheating; only smart game/network design.

By the way, especially the notion that everyone knows your code can be assumed being true; Due to the nature of .NET anyone can read the entire source code of Unity builds using a simple tool. To combat this I have made the Obfuscator package which is also available on the Unity asset store. The Obfuscator obfuscates code and hides stings to make it much harder to figure out.

5. Using a proxy server

The proxy server is a "middleman" that routes network traffic for a client to a server or vice verse. Hereby the proxy server is run on one of your servers. The number one reason you could want to use it for Unity networking is to guarantee that certain clients are able to connect with each other. The manual entry about using a proxy is so clear that I'll link it for you right away: http://unity3d.com/support/documentation/ScriptReference/Network-useProxy.html

Do not forget that it's not allowed to use a proxy on a client and server, only one of them can use a proxy simultaneously. Multiple clients can use a proxy as long as the server isn't using it.
Unlike the master server, Unity is not hosting a public proxy server. Download it here: http://unity3d.com/master-server/index.html

6. Combat Lag: prediction, extrapolation and interpolation

We have already briefly discussed prediction in tutorial 3: When you have an authoritative server that makes the final judgments, you can still have the client also calculate it's predicted movement to reduce wait times. This smoothens the players experience for it's own movements; the server will not need to visibly reset the player while he's controlling his own character.

While watching other player movements they will still show a lot of stuttering. How bad this is depends on how often movement updates are send (lag and network sendrate). This can be smoothed by two ways:

1. In between two updates your application should blend to the latest position instead of setting the position to the latest value right away. This is called interpolation.

2. Predict future movement: if a character was walking forward the last X seconds, keep walking forward, this will be the most likely next position. Wrong predictions can generate visible errors though, so too much prediction is no option either. This prediction is called extrapolation.

From the unity manual:

> *It is possible to apply the same principle with player prediction to the opponents of the player. Extrapolation is when several last (buffered) known position, velocity and direction of an opponent is used to predict where he will be in the next frames.*
> *Interpolation is when packets get dropped on the way to the client and the opponent position would normally pause and then jump to the newest position when a new packet finally arrives. By delaying the world state by some set amount of time (like 100 ms) and then interpolating the last known position with the new one, the movement between these two points, where packets were dropped, will be smooth.*

An example of interpolation/extrapolation is available in the official unity Network Example project(http://unity3d.com/support/resources/example-projects/networking-example) and is also used in the FPS example of this tutorial. Furthermore you can always raise your network sendrate option to gain synchronization, but this is at the cost of bandwidth.

7. Manually allocate networkview ID's

Network.Instantiate doesn't work very nicely for an authoritative setup. To gain more control over your network setup you will want to allocate the networkview ID's yourself. I suggest you to never use Network.Instantiate but to stick to manual allocation.

See Tutorial 4 or the Unity manual:
http://unity3d.com/support/documentation/ScriptReference/Network.AllocateViewID.html

8. Network and level loading

For a network it doesn't matter whats running on every server/clients PC as long as the network communication runs smoothly. The network only cares about the right networkviews being present (and specifically their viewIDs). This means that you can have a server running a game scene, whilst a new client just connects via the game lobby scene. This is usually not a problem, except for when the server tries to send the client all buffered instantiated game objects. For this reason you'd better shutdown the network communication on the client temporary while loading the game. This can be done by calling "Network.isMessageQueueRunning=false;" on the client right after the server connection was successful, this is shownin example 2.

This new knowledge gives you new insight: A server could host multiple game sessions/multiple levels (even in the same scene) by a smart usage of network groups/scoping and message filtering on the server. Just be careful with collision between players from different game sessions.

9. The Unity master server, Connection tester and Facilitator

Unity 3.0 added a few new networking features, most notably regarding the NAT handling and the master server. Let's first go over the basics. Unity provides a few applications regarding multiplayer:

Connection tester:
Provides network connection testing to determine whether a client is running in a public or private IP and whether NAT punctrough is possible. If you want to host this application yourself you need 4 IP addresses(!).

Facilitator:
Does the actual NAT port mapping for clients.

Master server:
Keeps a list of registered servers

All three applications are hosted by Unity per default you are using Unity's publicly hosted master server (and facilitator and connection tester), unless you customize the IP addresses and ports of these services in script. You could want to host them on your owner server to gain control over the performance and uptime.
For the full Unity documentation see: http://unity3d.com/support/documentation/Components/net-MasterServer.html
To download the server applications: http://unity3d.com/master-server/index.html

Do I need to run my own master server?
The Unity master server works fine at the moment, it has just greatly improved in 3.0 and some more stability fixes have been applied very recently. It depends on the size and budget of your game(s) whether you'd want/need to run your own server. It is possible for anyone to read/register the same game name on the public Unity server though, this could be a possible issue you might want to prevent. For development and small webgames using the default Unity master server is

perfectly fine right now.

So what is this NAT stuff?
Before Unity 3.0 NAT was something which was very present. Right now it's hidden a bit more, which is a good thing. You should not worry about NAT too much, but here are the basics: Whenever I say NAT, I actually mean NAT punchtrough or NAT port mapping. NAT stands for Network Address Translation [4]. This is used to open up a port on the servers side so that clients can connect. The server started with NAT enabled because the default port it initialized on was (probably) not connectable; not public.
When initializing a server with NAT enabled, the server connect to the facilitator and keeps a session open on an agreed port. Future clients which want to connect to this server can choose to use NAT or not. The master server list knows which servers are NAT enabled or not. When connecting using a HostData or GUID argument, NAT is used if enabled. Remember the GUID en HostData are received from the master server. When connecting using NAT the client will connect to the port the server and facilitator agreed upon, the client got this port from the master server entry as well. Even if the server started with port 20000 assigned, the connection attempt might be on, say, port 48756 because that was set up by the facilitator. Now, depening on both the client and server the NAT could be succesful. If not, it's always worth a shot to try a reconnection on the same IP and the default server port as the port might be open(or opened by the user). My MultiplayerFunctions class does this reconnection by default.

For connection problems see item 11.

10. Ask a server for data before joining a game

If a client is in the main menu and wants to join a specific server (which is in-game), it will need to know some data, such as what level it needs to load. If you connect to a server it will send all buffered calls and instantiated network objects right away. If a client is still in the main menu those calls will generate errors.

To solve this, there is one possible workaround. First of all, you could use the masterserver host data 'description'/comments field to save all your required data, such as the level number. However, if you do this, a direct IP+port connection will still lack the required data. For example, LAN games are unable to use the master server.

Behold my hacky (but stable!) workaround: Add a networkview to the main menu and make sure it has viewID 1. Also add a script (e.g. GameSetup or GameManager) to your game scenes that always has viewID 1. You are now able to send messages from the server game scene to the clients main menu scene. At every new connection have the server send the client the level number. Upon receiving the data in the main menu, have the client disconnect right away. The client must reconnect to account for all lost instanitation and buffered calls: the client received these in the main menu and these only caused errors here, they are simply dismissed. Have the client reconnect and disable the network message queue after a succesful connection immediately (see Example 2). This will prevent a client receiving the instantiation calls while in the main menu scene. Once in the game scene, enable the message queue again and all network messages will be received and handled properly.

11. Resolving connection issues

[4]Die-hard background information on NAT: http://www.jenkinssoftware.com/raknet/manual/natpunchthrough.html

Connection wise a multiplayer LAN game is no different than a multiplayer internet game, except for the fact that LAN speeds/connections are usually better and setting up the connection is a bit different. Once you are able to connect your game over LAN you'll find out that getting it to work over the internet can be a bit cumbersome. Therefore I've made a list of things you can check to diagnose the problem.

First of all, if you use a firewall on your PC make sure it allows your application to make connections. If you start hosting a Unity game on windows usually a dialog pops up asking whether the program is allowed to do so, allow all connections. Then only remaining hurdle when asking a friend to connect to your hosted game is that your router will block all 'random' incoming connections because of security reasons. You are only able to connect via the master servers NAT punch trough or by configuring your router.

NAT punchtrough
The master server NAT punch trough does not work all the time, but if the router allows it, these settings are required to make it work:

- Initialize the server with NAT enabled and register it at the master server
- Have the client connect via the master server HostData (or it's GUID). This will automatically use NAT puchtrough if available.

Example 2 complies to the above, so this example should allow your friend to connect to your hosted game and/or vice versa. If the connection doesn't work (try both ways) then you and/or your friends router is too restrictive and doesn't allow NAT punch trough. One option remains, you'll have to configure your router to open a port to your PC.

Port forwarding in your router
This option, when properly set up, always works. You can't expect all your players to configure your router though. To do this you specify that e.g. port 20000 always points to 192.168.1.102 where this last IP resembles the LAN IP address of your computer. Then, everyone will be able to connect to your external (public) IP address on port 20000. How you set up port forwarding differs per router. This site could help you figure out how your router works: http://portforward.com/

One issue remains, which I have fixed for you: When players connect to an server using it's HostData (or GUID) as in Example2, then it'll connect using NAT per default and use the NAT supplied port, which is something random like 48936. That's why, on a connection failure, I have integrated a reconnection in my MultiplayerFunctions class. This reconnection will use the default server port and no NAT. Another option is to disable NAT on a server of which you know the port is open. The game will not even try to connect on a diferent port as it only does so if the masterserver provided one due to NAT.

Adding multiplayer to your own game

Hereby a quick reminder on how to apply all these resources to your own project. For a general project I recommend you to copy the Main Menu scene (with scripts) from Example2 and with this also the plugins folder containing my MultiplayerFunctions and GameSettings classes. This is all you need for a regular multiplayer game.

In MultiplayerFunctions customize the game name to make sure it doesn't conflict with other games on the master server. Optionally change the default port.

If you have this in place what lasts is your actual game scene with its gameplay related multiplayer messaging, plus you'll want to improve the main menu.

(Oh, do please try to customize the GUI (skin) of your games multiplayer menu. I've seen a lot of games that copy this projects GUI code 1:1. This is no problem, but your game will look much more polished if you add your own look and feel to the entire project.)

Tips

Best practices

- Always design your games authoritative, unless that has an unacceptable negative impact on performance.
- Never use Network.Instantiate. Do manual allocation instead.
- oh and p.s.: Move to C#!

Open multiple unity instances (for network debugging)

You are not allowed to open the same unity project twice. However you can open a second Unity instance running a different project folder. You can copy your project twice to be able to run the server and client from inside the editor, this does mean you need to apply your changes at both instances.

On Windows:

Open unity while holding the SHIFT and ALT keys **or** create a .bat file with as content:

"C:\Program Files\Unity\Editor\Unity.exe" -projectPath "c:\Projects\AProjectFolder"

Correct the right path to Unity.exe. Executing the bat file should now allow opening the unity editor twice. You can use a nonsense project folder argument to have Unity popup a project window every time.

On Mac OS X:

Open unity while holding the command key **or** run this terminal command:

/Applications/Unity/Unity.app/Contents/MacOS/Unity -projectPath "/Users/MyUser/MyProjectFolder/"

Group limit

Only 32 groups are available to assign, even though you can assign e.g. group number 48 to a networkview. Be warned: assigning 48 works as if assigning group 16 (48%32=16).

Scopes

I was very excited about the new network scopes that were introduced in Unity 2.1. However; they only work on OnSerializeNetworkView **not** on RPC's.

RPC bug?

Don't let this bug kill your time: If you have a game where the authoritative server itself is also a player you might want to use this code:

```
networkView.RPC("SendUserInput", RPCMode.Server, horizontalInput, verticalInput);
```

However, this does not work. You can not send a network message to yourself (Except via RPCMode.All). Instead, use the code below. Hopefully a future unity version will improve this.

```
if(Network.isServer)
       SendUserInput(horizontalInput, verticalInput);
else
      networkView.RPC("SendUserInput", RPCMode.Server, horizontalInput, verticalInput);
```

Starting multiple servers on the same computer

You can only start one host on the same port per computer. If you want to start two servers on the same computer, use different ports. Otherwise Unity will throw the error:

> *Failed to initialize network interface. Is the listen port already in use?*
> *UnityEngine.Network:InitializeServer(Int32, Int32, Boolean)*

NetworkMessageInfo.Sender reports invalid NetworkPlayer

NetworkMessageInfo.sender will report the NetworkPlayer "-1" instead of the real networkplayer when a networkmessage is called on the client/server itself (e.g. when calling RPCMode.All). It is -1 because the message has not really been send. While this is logical, I think it makes more sense to report the real NetworkPlayer, this'll greatly simplify networking coding as well. Checking whether a message was truly send can be done using other methods. Let's hope Unity considers changing this.

Running dedicated servers

A downside of the Unity networking is that the current dedicated server isn't as dedicated as we'd possibly want. Rendering is disabled, but there's still much overhead. Running a dedicated server is possible by passing the batch mode argument to the executable. See: http://unity3d.com/support/documentation/Manual/Command%20Line%20Arguments.html
When running a dedicated server, you should use "Application.targetFrameRate" to make sure Unity doesn't try to run your server at 1000+fps, hogging your resources. Capping it to 20 FPS is fine for most games.

Downloads for the Unity master server and Proxy server

http://unity3d.com/master-server/index.html

Other networking options

Is something of the build in Unity networking really bothering you or do you really lack a specific feature? There are other networking options for your games. Here's a list I gathered to evaluate my choices (November 2010). It's listed in order of my *personal* preference.

Networking alternatives

- Photon from ExitGames
- Electrotank
- Smartfox
- Create your own custom RakNet backend
- Project DarkStar
- Badumna
- Netdog
- Lidgren

Unitys networking solution works great for simple networking setups but it's simple setup it also limiting. These alternatives are required if you need true dedicated servers without any overhead, when you need multiple connections or even P2P... and so forth.