By Mike Hergaarden - Last edit: 29-10-2009

Content

About this tutorial....2
About the author....2
How to use this tutorial?....3
Tutorial 1: Connect & Disconnect....4
Tutorial 2: Sending messages....6
Our very first multiplayer scene..just one player though!....6
Tutorial 2A: Server plays, client observes, no instantiating....6
Tutorial 2B: Server and client(s) play, with instantiating....9
Tutorial 3: Authoritative servers....10
Further network subjects explained....12
Real life examples....15
Example 1: Chatscript....15
Example 2: Masterserver example....15
Example 3: Lobby system....16
Example 4: FPS game....17
Tips....18

About this tutorial

I always thought Unity needed a proper networking tutorial. When I started unity networking the networking example was a bit too confusing; A proper networking example should have independent examples so you know where to look for something within seconds. With that in mind I decided to join the UniKnowledge contest and finally give the community a networking tutorial, that I hope, is all you need to create a networked game.

This tutorial features many examples; from small (but important) techniques to a real FPS game. You are advised to read this document from start to end, 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.

About the author

This tutorial is written by Mike Hergaarden ("Leepo") from M2H (www.m2h.nl). We've been using Unity for over two years now, though most of our unity development time has been spent in the past few months. We've been concentrating on unity multiplayer since the very beginning. In fact the first unity game we made featured multiplayer; it's that easy to add! Our multiplayer games include: Crashdrive 3D, Cratemania, Surrounded by Death, Verdun Online and our current upcoming multiplayer challenge is Hyberon.

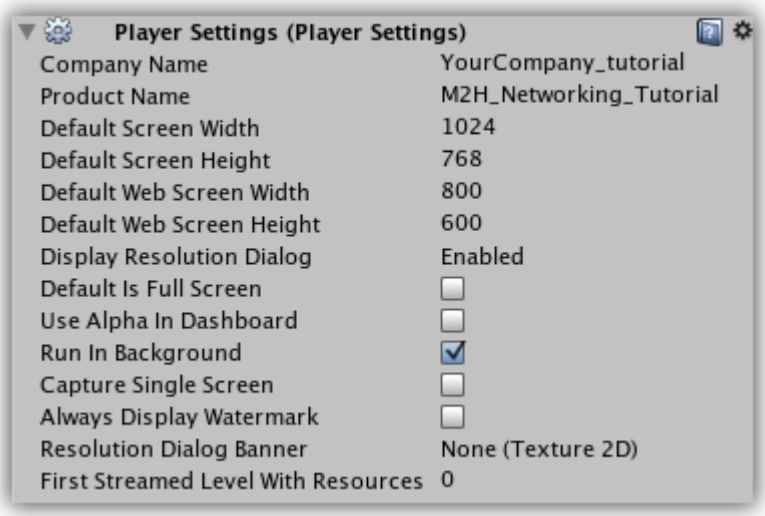
Have fun going trough this tutorial. If you made something exciting thanks to this tutorial, let me know!

How to use this tutorial?

Combined with this document is a zip file containing the unity project which is used in this tutorial. It is assumed you have already made yourself familiar with the unity editor and basic scripting, if not: check out the unity (video) tutorials.

Multiplayer isn't much fun to debug since you need to have two instances of your game running (server and client). During this tutorial you're advised to run the server in the editor game view, and a client in a webplayer.

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" .



Tutorial 1: Connect & Disconnect

Let's get you going!

- 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.
- Build a webplayer and run it.
- Start the scene in the editor as well and click on "Start a server" (using the default values for IP and port)
- Click on "Connect as client" in the webplayer.
- You should now see "Connection status: Client!" and "Connection status: Server!" on your two instances. Congratulations, you have 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 easily 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 and maximum allowed number of connections 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. You need to keep one more setting in mind before connecting to a server or when initializing a server, that is the "Network.useNat" bit you see just above the corresponding connection/initializing code.

NAT connection (Network.useNat)

We have set Network.useNat to false since we do not want to use Network Adress Translation. NAT is useful for clients behind a router (inside a LAN). This networking demo should only be run inside a LAN; you won't be able to connect to your friends house (unless you or he/she has a very unrestrictive firewall/router). For more information about NAT see: http://unity3d.com/support/documentation/Components/net-MasterServer.html

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 6 client and server functions should be very self explanatory; They are called on **only** the client(s) or **only** the servers. If you want to use the parameter passed by the functions, checkout the unity manual entries for these functions.

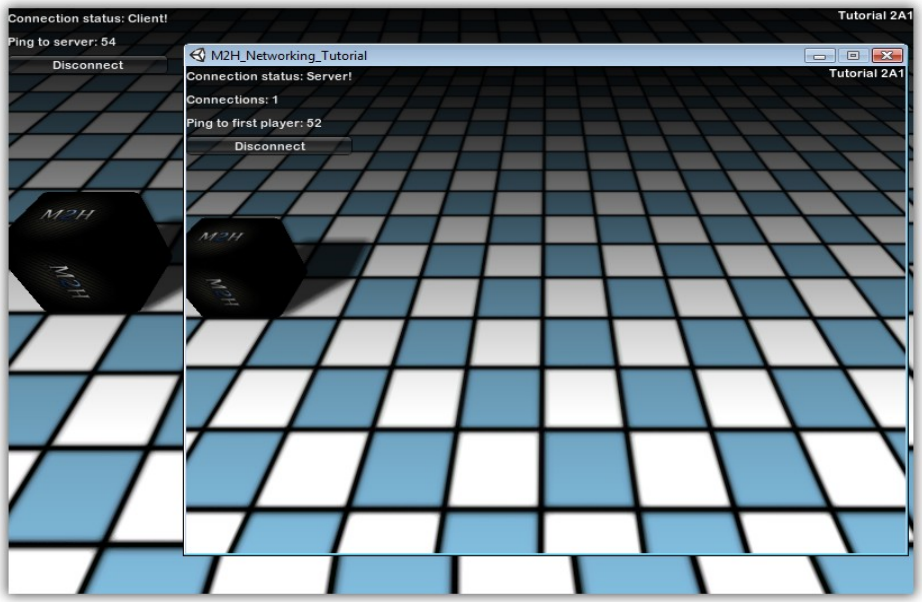
The last three functions are different. **OnFailedToConnectToMasterServer** is called on a client when you somehow can't connect to the masterserver, more details about the masterserver will follow later on. **OnNetworkInstantiate** is called on instantiated objects, we'll also have a look at

this later. **OnSerializeNetworkView** is one of the two methods for us to send messages across the server and clients. **RPC** calls are network messages/functions you define yourself. In the next tutorial we'll have a look at serialization and RPC calls.

To conclude this tutorial have a look at the Network.* “Messages Sent”, “Class Variables” and “Class Functions“ here:
http://unity3d.com/support/documentation/ScriptReference/Network.html

Now you know where to find all this information in the manual as a reference. We've already briefly discussed about 75% of the information over there!

Tutorial 2: Sending messages



Our very first multiplayer scene..just one player though!

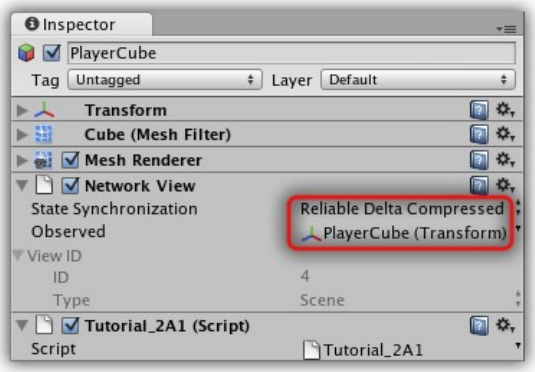
Tutorial 2A: Server plays, client observes, no instantiating.

Tutorial 2/Tutorial 2A1

Don't let the title scare you, open the scene "**Tutorial 2/Tutorial 2A1**". The 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 recieves network messages requires a NetworkView component. You could just use one networkview for your entire game by referencing it from script, but that wouldn't make sense; in general just add a networkview per object that you want networked.

Run the 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 (henche the Network.isServer check): When the server uses the movement keys; it will move the cube right away. Don't worry about laggy movement in the first three tutorials, we'll get to that once we have covered the basics.

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).



Let's look at the rest of the NetworkView options to completely wrap up this subject. The PlayerCube's networkview “State synchronization” 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 eh! 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. You can use RPCs together with an observed object though. RPC's will be introduced in tutorial 2A3; it basically are network messages(/calls) you define yourself.

Tutorial 2/Tutorial 2A2

What if we did want to move the cube in the y-direction, or we wanted more contro over what's being synchronized by unity. Open and run “**Tutorial 2/Tutorial 2A2**”. The game should play exactly the same, 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. 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).

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. This demo should work just like the last two. 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. These s 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/allow 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	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.

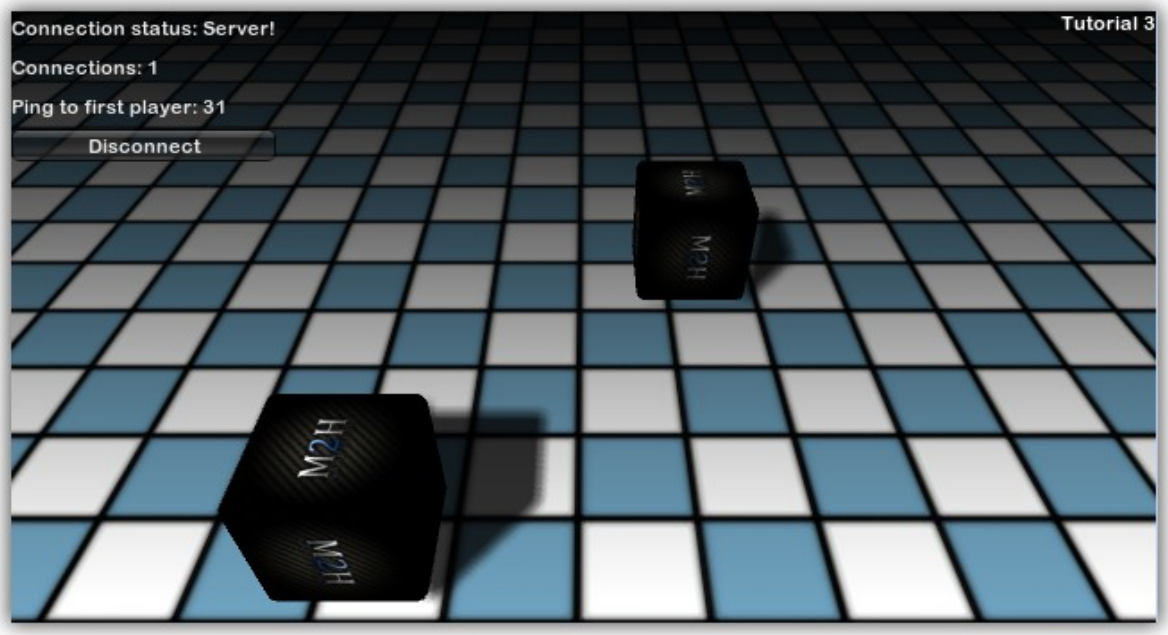
Be proud of yourself if you still (roughly) understand everything so far; we've finished the basis and hereby most of the subjects. We now just need to go into details.

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, including the server, even though the servers player could be easily removed to run the game as dedicated server. 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 **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(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.
"**Tutorial_2B_Playerscript.js**" uses the movement code from **Tutorial 2AB** with the as difference that only the objects owner input is captured.

Tutorial 3: Authoritative servers



Yay; multiplayer!

The server setups of the last 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. 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 usually 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.

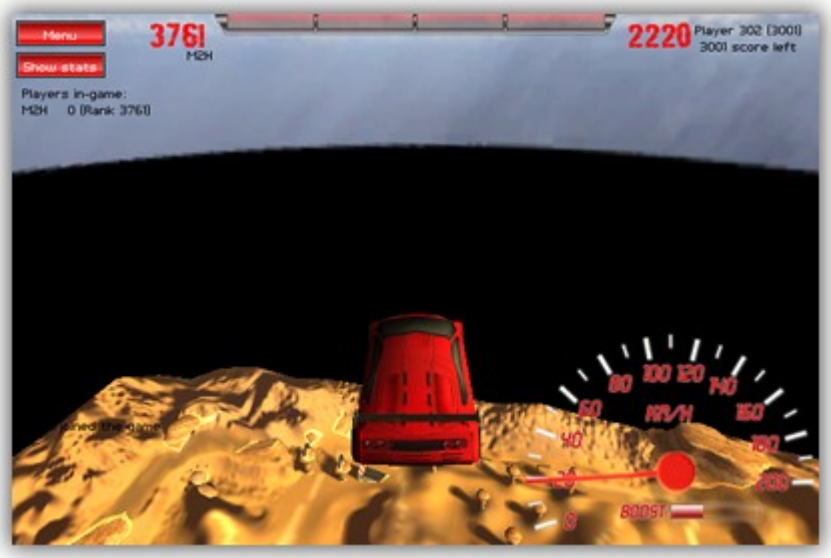
We will send all clients movement input to the server, have the server execute it, and send the result (the new position) back to all the clients. Have a look at the Tutorial3 scene. Again, it'll play just like before, but the inner workings have changed. The movement will probably feel quite a bit more 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 with 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 a client function.
Moving on to “Tutorial_3_Playerscript.js”, the script is now not always 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 owns this object. 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, 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 the OnSerializeNetworkView function (which is executed far less than Update).



"Technically" you can make this happen ;).

Do note that you really don't need to make (all) your multiplayer games (totally) 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 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.

Further network subjects explained

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.

Limiting traffic: Scoping and group limiting

You can greatly improve network performance by limiting the amount of data you sent. In a multiplayered world players do not need to receive every bit of information. What happens at the end of the world is irrelevant to a player which is miles away. 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 problem 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.

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(); once before initialising the server. The security adds up to 15 bytes per packet.

Anti cheating

Even with the security layer in place from the previous section, 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 a smart game/network design.

Using a proxy

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

We are very interested in using a proxy server to improve multiplayer (webplayer) games connectivity, but I haven't looked into this subject myself as of yet.

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.

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 and is also used in the FPS example of this tutorial. Furthermore you can also raise your network sendrate option to gain more precise synchronization.

Manually allocate networkview ID's

Sometimes Network.Instantiate doesn't work well enough for an authoritative setup. To gain more control over your network setup you might want to allocate the networkview ID's yourself.

Code example: http://unity3d.com/support/documentation/ScriptReference/Network.AllocateViewID.html.

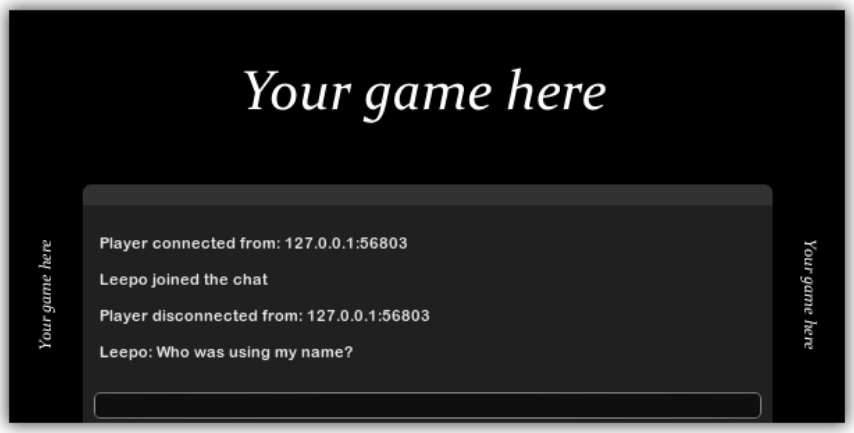
Network loading

For a network it doesn't matter whats running on every server/clients PC as long as the network communication runs smoothly. 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, see the game lobby code example.

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. Just be careful with collision between players from different game sessions.

Real life 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 playernames. 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 playerlist 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 masterserver to show all current running game sessions. The quick play option can be used by player that want to join the first random game session available. Furthermore under the advanced connection options a player can start a host, enter an IP and port for a direct connection or use the

sorted masterserver list to select a game manually. The only missing feature are hosting/joining password protected games. This can be easily added for hosting and direct connecting. For the game list you'll need to add a password popup window though.

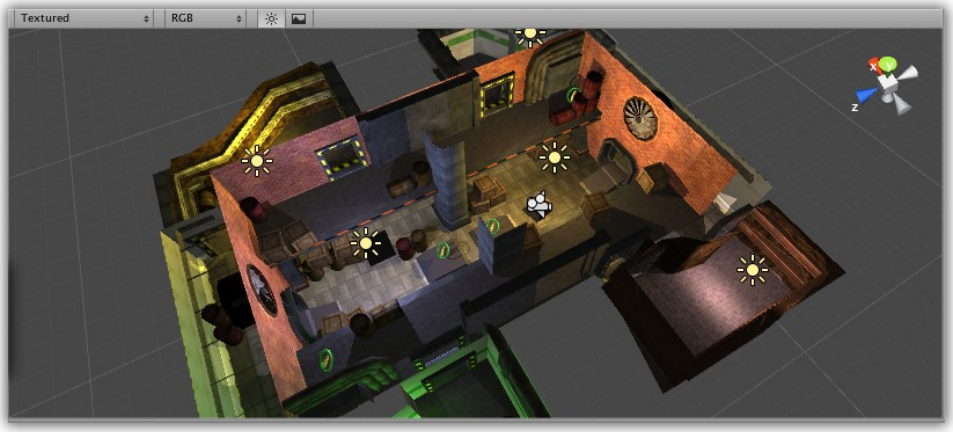
The 'game' in this example only shows you the connection status of the server and clients. You could easily replace the game scene with any other scene and the networking will work right away. You'll only need to set "*Network.isMessageQueueRunning = true;*" since we disabled it in the menu scene to prevent strange things from occurring 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 masterserver once the server has started the game scene.

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 unity users that are new to multiplayer coding want to create an FPS game I decided to provide a FPS game base as the very last example. 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 masterserver code for the connection setup 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, etcetera.

Tips

Common pitfalls and handy techniques.

Open multiple unity instances (for network debugging)

You are not allowed to open the same unity project twice, therefore you need to run a script to open a second unity instance. 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:

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.

On Mac OS X:

Run this terminal command:

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

OnSerializeNetworkView bug in 2.6 (and earlier)

I never wanted to use the OnSerializeNetworkView networking method (loved RPC so much), and when I finally did it appeared to have a big flaw. This flaw only occurs when you allocate viewid's yourself. An extract of my bug ticket:

"When using "OnSerializeNetworkView" with an observing networkview (reliable delta compressed in this case), it all works fine with 1 player (as host). As soon as a second client joins, it will complain about not knowing about the first assigned networkviewID of the first player.

*"Received state update for view ID ******random info here about your specific number*** but no initial state has ever been sent. Ignoring message."*

The problem is that these networkviews are bugged: the new client never get the "initial state" of the networkviews that have been assigned BEFORE that new client connected. This affects all clients."

Also see: http://forum.unity3d.com/viewtopic.php?p=77193

Group limit

You can only have 32 groups, even though you can (code wise) assign e.g. group number 48 to a networkview. Be warned: assigning 48 works like assigning 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. Instead, use:

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

Run dedicated servers

A downside of the unity networking is that the current dedicated server isn't as dedicated as we'd possibly want. However; it still works. Running a dedicated server on Mac OS X is possible by giving the batch mode argument to the executable. A windows equivalent option has been added in Unity 2.6.

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.

Connection issues: How to connect over the internet

Connection wise a multiplayer LAN game is no different then an multiplayer internet game, except for the fact that LAN speeds/connections are usually better. 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.

Connections over the internet do not work:

- Does the game work over LAN?
- Do both computers have a working internet connection?
- Ensure that both PC's have opened up their firewalls for your application/port. You can try to temporarily disable the firewall(s) to ensure it's not causing problems.
- Try a direct connection. Start a server and have the other PC connect to your **external** IP address ("internet IP address")**.**
- If this does not work; your network router is probably blocking unknown incoming connections as security measure. There are two options:
  1. Use NAT punch trough (See the masterserver example) and hope that your router supports NAT punch trough.
  2. You can open up the port you are using in your router and/or forwarding all the connections to that port to your internal LAN IP address. This will always work, but not all your players/users will know how to configure their router.

Other networking options

Is something of the built 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 (August 2009).

- Create your own custom RakNet backend
- Smartfox
- Photon & Neutron from ExitGames
- Project DarkStar
- Netdog
- Lidgren