Photon Unity Networking Guide

02 July 2013

Introduction

My first asset store project was the “Ultimate Unity Networking Project”. It was quite a success, especially considering the niche multiplayer market. I was a big enthusiast of Unity Networking and tried to get the most out of it. No matter how easy the built-in Unity Networking(*UN*) works for simple games, I knew it was inevitable that I’d have to look for an even better networking solution for any of my future projects (*who doesn’t want to make an MMORPG ;)?*).

I needed a better networking solution, but all of the available networking solutions looked very scary to use. I discussed this problem with ExitGames and the result of this is Photon Unity Networking(PUN) which I made in collaboration with ExitGames. PUN is a C# “API” that implement the Photon client. The API is very similar to Unity Networking. While PUN itself is written in C#, it can also be used via Unity javascript.

Now that PUN has shaped into a powerful product, the next important task on my list is to help *you* using it. This guide should help you rocket start your multiplayer implementation using PUN. The general outline of this document is quite similar to my earlier UN tutorial, however, many PUN specific details and inside knowledge has been added. Oh, and this time I didn’t need a “known-bugs / limitations” section ;).

Have fun!

Mike Hergaarden
Founder of M2H
@M2HGames

Basic PUN concepts

Photon? PUN? Photon Cloud?
I can understand that at a first sight it's quite confusing how the various product link to each other, so I'll sum them up:

Photon is the network technology ExitGames has developed. This consists of a server application and a client side API.
http://www.exitgames.com/

PhotonUnityNetworking (PUN) is a wrapper I have made around the Photon default client API. The PUN wrapper implements commonly used features to make multiplayer development much easier. You can use PUN in combination with the cloud or with self hosted server(s).
http://doc.exitgames.com/photon-cloud/PUNOverview/

The Photon cloud is a service run by ExitGames to make multiplayer game development even easier. Hosting, server operations and scaling is all taken care of in the Photon Cloud. You can use the cloud via PUN, or by using Photons loadbalancing client API.
http://cloud.exitgames.com/

Scripting languages
This guide uses C# only. However, it should be no problem to use "Javascript"/UnityScript instead as PUN fully supports both Unity scripting languages.

Need a free C# tutorial? http://u3d.as/content/m2h/c-game-examples/1sG

Networking architecture
For your games, you will use one or more dedicated Photon servers. Clients connect to this server and the server routes all traffic between players connected to that same server.
Facts:

- Photon uses a client-server model (just like Unity Networking, although with Photon the server is dedicated)
- You can choose to host a Photon server yourself, or use the Photon Cloud service.
- Thanks to Photon's default load balancing app, you can easily scale the amount of Photon servers to meet your multiplayer demands.

Photon Cloud service

Photon Cloud is a fully managed service run by Exit Games. Hosting, server operations and scaling is all taken care of in the Photon Cloud. You simply pay for a certain volume of maximum concurrent players. Using the Photon Cloud you'll only have to work on the client side of your game.

> Self hosting vs Cloud hosting
> Especially if you're new to PUN I highly recommend the cloud hosting because it allows you get started with PUN right away. At the moment of writing the PUN cloud hosting does not support server side logic. However, for the majority of games cloud hosting should suffice.

The master client concept

At the moment it is not possible to run server side logic in the PUN cloud. PUN itself also does not support it out of the box. Instead, for authoritative decisions the master client can be used. PUN has a special client named the 'master client', this is always the first client in a room. The master client will automatically switch when the current master client leaves. The master client can be used if you need one client to make authoritative decisions as it has some more options than regular clients:

- Every client can *only* destroy its own PhotonViews/objects. The master client is the only client that can also Destroy the PhotonViews/objects of other clients.
- The master client controls objects that belong to the scene (AI etc.)

Rooms and lobbies

When connecting to the Photon server you'll enter in the 'lobby' state: You cannot send any messages between clients in this state. However, you will receive information about available rooms automatically. *(In the lobby you are connected to the LoadBalancing app on the Photon masterserver.)*

From the lobby state you can create or join a room. Only after successfully creating or joining a room you can send messages to other players. Note that you will no longer receive room updates when connected to a room *(Internally PUN has disconnected from the LoadBalancing server and connected to a Gameserver).*

Typically, you will be in the lobby state in your main menu scene (where you have a multiplayer games browser). After connecting to an existing game you have entered a room and you should load your game scene.

Support

If you have any Photon / PUN related question you can use the official forum. ExitGames is very active on their forum.

ExitGames forum: http://forum.exitgames.com/
PUN subforum: http://forum.exitgames.com/viewforum.php?f=17

Getting started

1. **Optional: Check PUN version**
   For your convenience this package already contains the PUN package which can be downloaded for free from the Unity asset store. I'm trying to keep this up to date with the latest release, but you'd best check if this is the latest PUN version. (Compare PhotonNetwork.versionPUN with the latest PUN release). The latest PUN should always be compatible with this guide, if there are any problem, let us know!
2. **Setup PUN**
   After importing PUN, run the PUN wizard to configure your server connection (cloud or self hosted)
   - See: Window -> Photon Unity Networking

   I recommend using the cloud hosting trial. If you don't see the PUN entry under Unity's "Window" menu then you should check your console for compile errors that prevent Unity from compiling the editor script.
3. **Build settings**
   Because the asset store does not properly save build settings you'll have to correct the scene settings. Please run "*PUN guide>Reset build settings*" in Unity. If this option is not present in your Unity menu, make sure you have no open compile errors in your log that prevent Unity from compiling the editor scripts.
4. **Confirm your PUN connection works**
   To confirm that everything works, go ahead and try the first tutorial (see next page) to connect to a Photon server! To troubleshoot any setup problems you can use the developer forums.

Tutorials:

Tutorial 1: Connecting

The goal of this tutorial is to show how to connect to the Photon server and to clarify the lobby and room states.

Part 1A

1. Open the scene “Tutorial_1/Tutorial_1A”
2. Start the scene in the editor
3. Press CONNECT to connect to the Photon server.
4. The game view should now show that you are connected and display the Ping between you and the server.

Now, to see how this actually worked, check out Connect1A.cs. All that you really need is ‘PhotonNetwork.ConnectUsingSettings(“1.0”);’. This method automatically uses the connection settings that you specified using the PUN setup wizard.

At the end of the Connect1A.cs file you’ll find a ‘reference’ of all the events that PUN can call with descriptions on how to use them.

In this part we have connected to the Photon server. We connected to the ‘lobby’ state. In part B we will join a room to move from lobby to room state.

Part 1B

What we’ve done in part A is connecting to the Photon server. By default, this will connect to the server’s lobby. As long as you are connected to the lobby, your Room list will automatically be updated. You can use *PhotonNetwork.GetRoomList();* to get the list of active rooms, furthermore the two callbacks, OnReceivedRoomList and OnReceivedRoomListUpdate also notify you of updates to this list.

From the lobby you can start/join a room. Part B shows this with a simple script that resembles what your main menu code could look like. While the Connect1B script is quite large, it’s 90% Unity GUI code.

1. Checkout the behaviour of the Tutorial 1B scene
2. See how the code works in Connect1B.cs

It is important to discuss networking traffic while inside a mainmenu scene. You’ll want to join an existing multiplayer room from inside your mainmenu scene. However, you could be joining a game that is in-progress. This can cause problems as you’ll receive networking messages that are supposed to be executed in a certain game scene. Therefore, before joining a room you should know what scene to move next to (set the game level in the room properties). Then after successfully connecting immediately pause the network message loop and start loading the level. Once inside the game scene, you can continue the message loop. This might sound complicated, however, this process will be shown in all the examples later on.

Part 1C

Part B has largely covered all you need to know of the transition from lobby to room state. In your game this would be the transition between the main menu multiplayer lobby to joining a game and moving to the game scene.
In this last part we'll simplify the connect script to allow for automatic joining of an active room. The only purpose of this part is to speed up testing in the next tutorials, so that we don't have to manually connect to a room for our two test clients.

1. Once again, checkout the behaviour of the Tutorial 1C scene
2. ..and have a look at how Connect1C.cs works.

Tutorial 2: Sending messages using PhotonViews

In the second tutorial we'll work with some real multiplayer messages to control movement of our player (a cube).

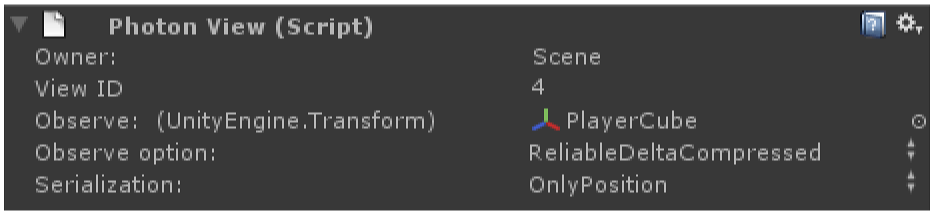
Part 2A1

Open the scene "**Tutorial 2/Tutorial 2A1**" and have a look around. The connection code from tutorial 1C has been attached to the "**Connect**" gameobject. Furthermore the "**PlayerCube**" object has the "**Tutorial 2A1.cs"** script and a "**PhotonView**" component attached. Every object that sends or receives network messages requires a PhotonView component. Using PUN, it is impossible to send or receive messages without PhotonViews. You could use one PhotonView for your entire game by referencing it from a global script, but that wouldn't make sense; in general just add a PhotonView per object that you want networked. Adding a PhotonView to every player and to your GameManager code would be useful. Adding a PhotonView to every bullet in your FPS is crazy though; you'll be running out of so called "view id's" (as there is a maximum of 1000 PhotonViews per default and besides that creating and destroying so many PhotonViews that often is just a waste of resources.

Run the 2A1 demo in the editor and in a (standalone) build. The second client will be able to look at the master client moving the cube around. All magic to this movement is thanks to the observing PhotonView and the movement code. Have a look at the `**Tutorial 2A1.cs"** script attached to the cube. This code is only being run on the master client (hence the PhotonNetwork.ismaster client check): When the server uses the movement keys; it will move the cube right away.

Now how do the other clients know about the master clients movement? Have a look at the PhotonView 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 master client to the other clients(and not the other way around) because the master client is automatically the owner of all PhotonViews that are in a scene. Later on you'll discover how clients can be owner of their own objects by instantiating PhotonViews 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 in general you should really try to gain more control over your network messages. I advise you to not use the observe property, by instead send all your messages via OnSerializePhotonView and/or RPC's. We'll get back to these two methods in the next tutorials. 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.


Photon View (Script)

Owner:	Scene
View ID	4
Observe: (UnityEngine.Transform)	PlayerCube
Observe option:	ReliableDeltaCompressed
Serialization:	OnlyPosition

Let's look at the rest of the PhotonView options to completely wrap up this subject.

Owner:
The player that owns this PhotonView. When a PhotonView is saved in a scene this is always the scene, meaning that the Master Client will be able to control this PhotonView.

View ID:
The ID of this PhotonView. For scene views this will be between 1 - 999. For player PhotonViews they will start with the player's ID. E.g. player #3 will start with PhotonViewID 3000 - 3999.

Observe:
The selected object to observe. Observing has no influence on the ability to send RPCs via this PhotonView. Supported observe targets are: MonoBehaviour(scripts), Rigidbody, Transform.

Observe option: There are three options:
Off: No synchronization via the Observe target. RPCs can still be send via this PhotonView.
ReliableDeltaCompressed: Only the changed data is sent and no more. If nothing changes, nothing is send
Unreliable: The observed targets information is send every time the PhotonViews are serialized (see: PhotonNetwork.sendRateOnSerialize, 10 times per second per default).

Serialization:
This option is only available when observing a Transform or Rigidbody. This helps you specify what you'd like to automatically synchronize. E.g. it often has no use to synchronize the scale of Transforms.

The PlayerCube's PhotonView 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 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 PhotonView. If you wonder why you'd ever set synchronization to 'off'; "**R**emote **P**rocedure **C**alls" need a PhotonView but don't use the "state synchronization" and "observed" option, both are ignored for RPCs. You can use RPCs on a PhotonView that is observing something, there are no conflicts. RPC's will be introduced in tutorial 2A3; they are basically custom defined network messages. Typically you will send player movement by observing a script (unreliable) via the PhotonView, while you will send messages like "Game started!" via RPCs. RPCs are more suited for messages that are not sent regularly.

Part 2A2

This part shows you how you can 'obersve' your scripts to synchronize custom network information. 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 PhotonView attached to the PlayerCube is now observing the "**Tutorial 2A2.cs"** script. This specifically means that the PhotonView is now looking for a "**OnSerializePhotonView**" 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. The OnSerializePhotonView function always looks as strange as this. This function is used to send and receive the data, PUN decides if you can send ("*istream.isWriting"*) by checking the PhotonView owner, otherwise you'll only be able to receive(the "*else"* bit). The master client is always the owner in this case as it owns all PhotonViews that are 'hardcoded' in the scene.

Part 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 PhotonView is no longer observing anything (and the state synchronization option has therefore been set to "off"). The mojo is in "**Tutorial 2A3.cs"**, specifically this line:
PhotonView.*RPC("SetPosition",* PhotonTargets*.Others, transform.position);*.

A RPC is called by the master client, 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
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.
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:

PhotonTargets.MasterClient	Only send to the master client (can be yourself)
PhotonTargets.Others	Send to everyone, except the caller itself
PhotonTargets.OthersBuffered	Send to everyone, master client the caller itself. Buffered
PhotonTargets.All	Send to everyone, including the caller itself.
PhotonTargets.AllBuffered	Send to everyone, including the caller itself. Buffered

Buffered means that when new players connect; they will immediately 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.

Tutorial 2B: Instantiation

We're going to get dirty with some details that could form the basis of a real multiplayer game. We want to enable multiple players, for this purpose we will be instantiating players when they connect, instead of having a hardcoded playerprefab in the scene. Remember that having a player in the scene like in the previous tutorials will only work for the master client. Open the scene "**Tutorial 2/Tutorial 2B**" and run it in an editor and in a (standalone) build. Walk around for a bit with the two players to verify the movement of both is networked properly. Each client should only be able to move it's own player-cube.

The **PlayerCube** has been removed in this scene, instead **Tutorial_2B_Spawnscript.cs** 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 groups for now). On disconnection the spawnscript will remove the instantiated objects. The one who calls a PhotonNetwork.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 PhotonView owner to decide whether one can control a player. "**Tutorial_2B_Playerscript.cs**" uses the movement code from **Tutorial 2A2** with the as difference that only the input of the object owner is captured.

Tutorial 3: Authoritative servers

The network architecture of the last few examples are what's called “non-authoritative”; There was no server 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.

In PUN, instead of the server we can use the master client to make authorative decisions. Note that this does not offer full authorization since this master client still is a normal client! If you need full authorization you can extend the Photon server to control (some) game logic. In this guide we will not cover server side game logic but will focus on utilizing the master client as the authoritative client.

Let's first sit back to think what changes the last (2B) example would need to make it authoritative. First of all; the master client needs to spawn the players, the players cannot decide when they want to be spawned and where. Secondly, the master client needs to tell everyone the correct position of all player objects, the players can't directly share their own positions. Instead, the master client 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 master client, have the master client execute it, and send the result (the new position) back to all the clients. Have a look at the Tutorial 3 scene. Again, it'll play just like before, but the background has changed. The movement will probably be even more laggy than before, but this is due to simplicity and 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.cs”. Normal clients no longer do anything here, the master client starts a spawn process when a new client connects. Furthermore the master client 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.cs”, this script is now not only run by the objects PhotonView owner. Since the master client instantiated all objects, it is always the owner of all PhotonViews. 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 master client executes all playerscripts to process the movement input and actually move the players. We now have a fully authoritative master client!

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 master client to process it and then finally receive the new master client position. While we do want the master client to have all authority, we don't want the clients waiting for the master client response too long. We can quite easily fix this lag by **also** having the client calculate the movement right away, the master client will always overwrite its calculated movement anyway and is still authoritative. This is quite easy to add.

> Adding local “prediction”:
> In “Tutorial_3_Playerscript.cs” simply also have the client execute “*SendMovementInput(HInput, Vinput);*” where you are sending the movement RPC (uncomment line 55) . Then make sure that the SendMovementInput RPC call actually affects the client by updating the (master client) movement code in the bottom of the Update() function; also run it on the local player by adding “|| *PhotonNetwork.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.
>
> Smoothing lag
> 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, first comment out the position assignment at line 109. Then check out lines 71-74, here's a snippet to “merge” the clients current position and the servers position, with the servers position having more weight. Notice that the Lerping is done in Update since that is called every frame, whereas OnPhotonSerializeView is only called 10 times per second per default. The end result is that the player cubes will slowly blend their movement instead of just static stuttering movements during network updates.

“Technically” you can make this happen

Do note that you don't always need to make your multiplayer games authoritative. It is a bit extra work, and when using the master client concept you rely on the master clients connection to the server which will add extra lag. None of our own games are fully authoritative. Yes, a player could *possibly* change its 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 what’s really vital to make authoritative. Also do not forget that even an authoritative master client itself can still cheat.

Tutorial 4: Manually instantiating PhotonViewIDs

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

1. ..use PhotonNetwork.Instantiate
2. ..save it in the scene (e.g. have a GameManager object with one PhotonView attached)
3. ..manually instantiate a PhotonView and assign it an ID. You need at least one existing PhotonView to be able to send over the new PhotonView though.

Using manual allocation instead of PhotonNetwork.Instantiate() gives you a bit more control, furthermore you do not need to have your prefabs under a "Resources" folder which is required by PhotonNetwork.Instantiate.

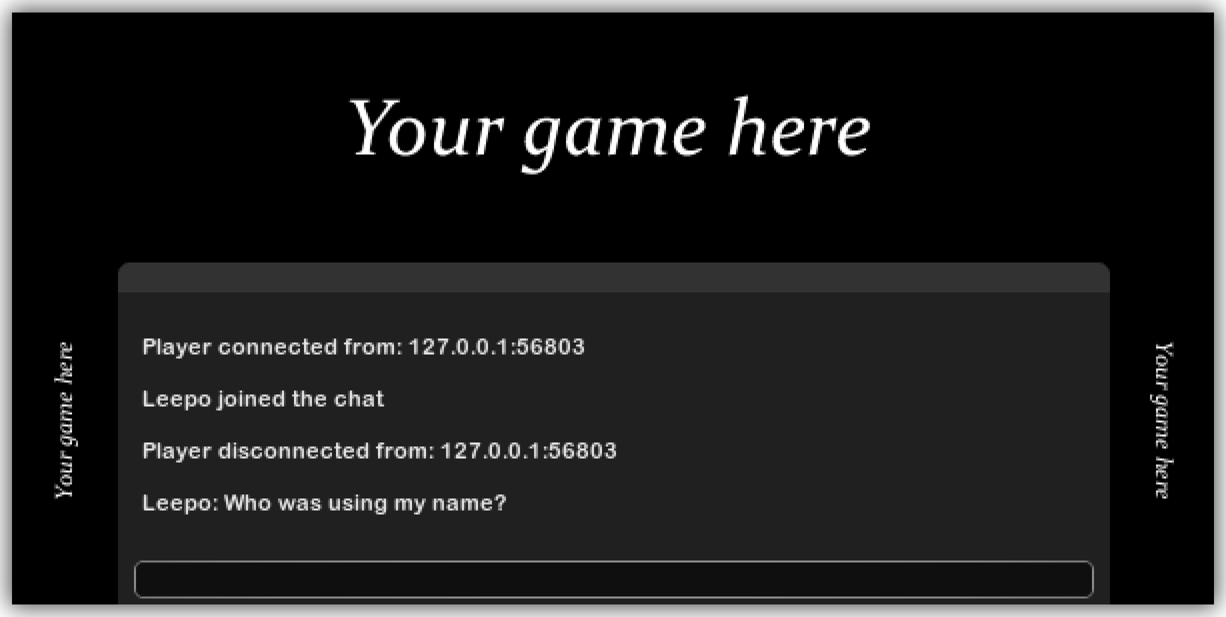
Tutorial 4 is all about the Game manager script (see GameObject **'code'** -> **Tutorial_4_GameManager.cs**). The player movement here is based on 2B and is not of interest here.

The game manager maintains a list of all players and their transforms. When a master client 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. PhotonTargets.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 PhotonTargets.Others to send the AddPlayer and SpawnOnNetwork messages only once, without buffering it. But when a new client connects the master client should then send all player data manually.

Examples

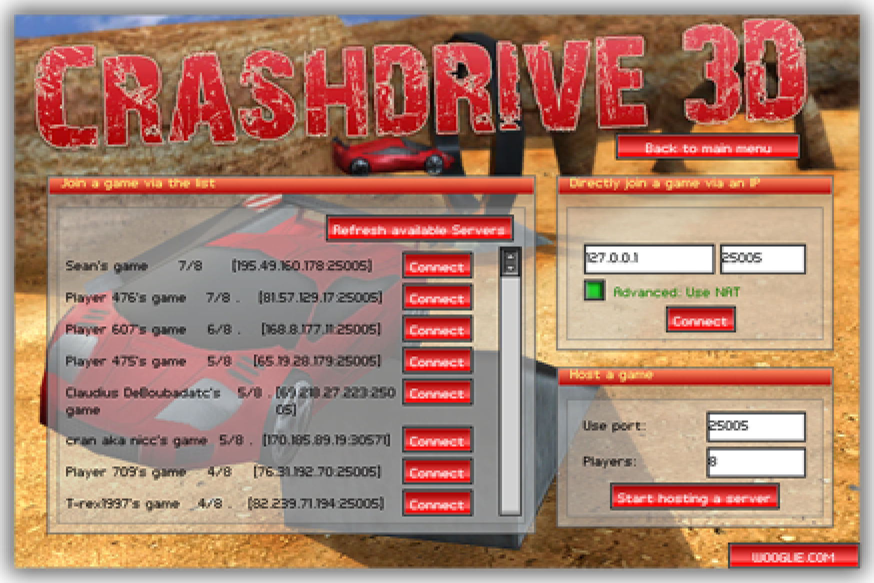
Example 1: Chatscript



```
Player connected from: 127.0.0.1:56803

Leepo joined the chat

Player disconnected from: 127.0.0.1:56803

Leepo: Who was using my name?
```

The scene “Example1/Example1_Chat” is nothing more than the connect script as seen in tutorial 1C 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. The chat currently holds at maximum 4 lines. When you modify the code to show more lines, you could use a coroutine to delete/”fade out” older messages. The player names are taken directly from PhotonPlayer.name (which is networked automatically).

Example 2: Game list



Sean's game	7/8	[195.49.160.178:25005]	Connect
Player 476's game	7/8 .	[81.57.129.17:25005]	Connect
Player 607's game	6/8 .	[168.8.177.11:25005]	Connect
Player 475's game	5/8	[65.19.28.179:25005]	Connect
Claudius DeBoubadatc's game	5/8 .	[6] 218.27.223:25005]	Connect
cran aka nicc's game	5/8 .	[170.185.89.19:30571]	Connect
Player 709's game	4/8	[76.31.192.70:25005]	Connect
T-rex1997's game	4/8 .	[82.239.71.194:25005]	Connect

127.0.0.1 25005

🟩 Advanced: Use NAT

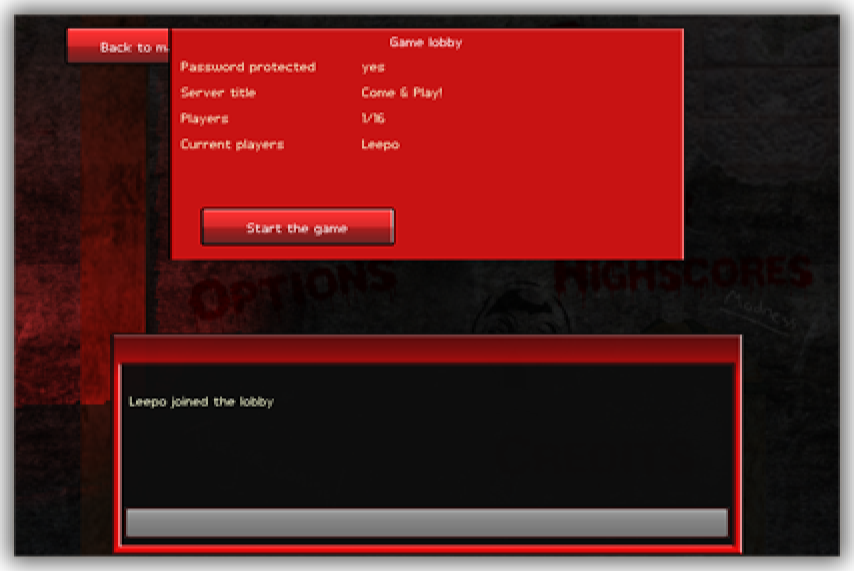
Use port: 25005

Players: 8

The multiplayer menu implementation in our game Crashdrive 3D

Open the scene "**Example2/Example2_menu**". This example showcases how you can create a multiplayer menu to show all currently running games. The GUI as shown in the example is a basic bare bone but it does feature flexible functionality: Under JOIN the player can see a list of all active hosts, join a random game or join a game by title. Under the CREATEGAME option a player can start a game. 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 remember to set "*PhotonNetwork.isMessageQueueRunning = true;*" in your game scene since we disabled it in the menu scene to prevent strange things from happening in the menu scene when a client joins a game that's in progress.

Example 3: Lobby system



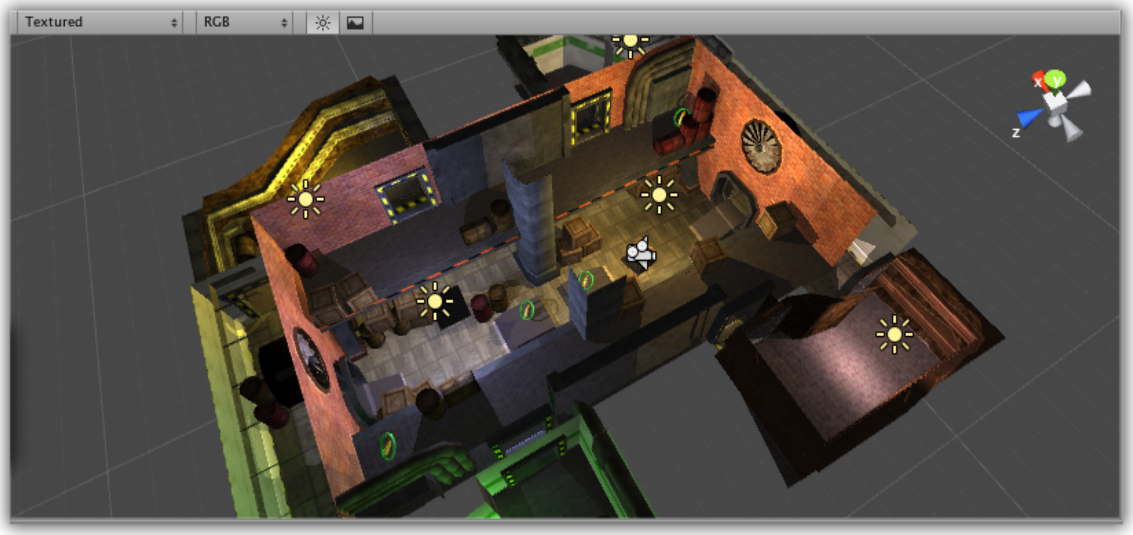
Password protected	yes
Server title	Come & Play!
Players	1/16
Current players	Leepo

```
Leepo joined the lobby
```

A simple lobby system implementation in "Surrounded by Death"

"**Example3/Example3_lobby**": This example is very much like the first example, except that it features a pre-game lobby. All rooms are shown in the room list by default, games that have started are removed from this list right away by setting Room.open and Room.visible to false. 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

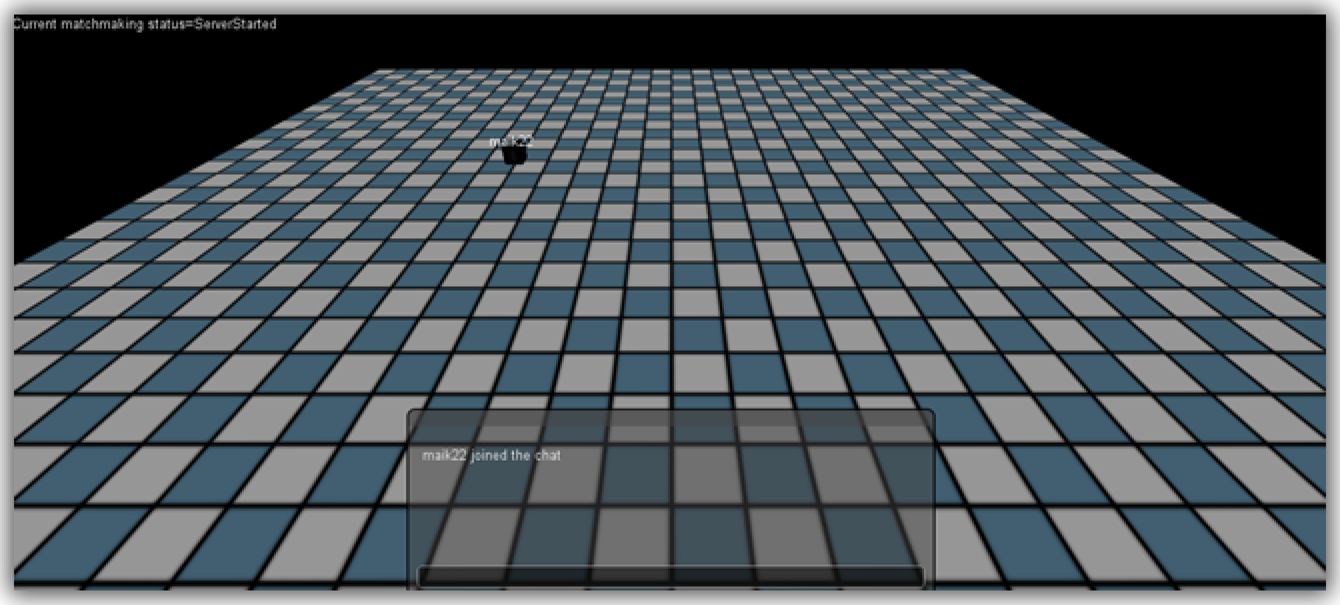


Since most people want to create a FPS game I decided to provide a FPS game base. This FPS example is non authoritative, so it's up to you to redesign the game for a secure authoritative setup ;)!

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 character 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 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 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 an available room again. This 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; in lobby or inside a room. Both states are possible at any time(!). This means that the code will have to check if you're connected to a room to send a network message *or* to execute a local message when you're disconnected. Even so, I didn't have much pain adding gameplay (events) to Crashdrive and PUN made this even easier.

Further network subjects explained

1. Best practices

For your player characters, use **unreliable observation**. While this might cost more bandwidth then using reliable, using unreliable messages saves quite some performance overhead on both client and server side. Use reliable to easily synchronize values that don't change often (like highscores if you don't want to use RPC's)

Use the **Photon cloud** if you can, this saves you time you can spend on your game instead of server management.

Unless really necessary, don't focus on making your game full **authoritative**, you can achieve far smoother network behaviour by finding the right balance between client side and authoritative code.

Your **AI** characters can be spawned easily by PhotonNetwork.**InstantiateSceneObject**. Doing so they will belong to the scene. Whoever is the active master client will be able to control these objects.

PhotonNetwork**.Instantiate** can be used to spawn every clients own player character. However, for something like firing missiles you could send an RPC on that player's weapon and then spawn the missiles manually yourself. If you would use PhotonNetwork.Instantiate and PhotonNetwork.Destroy on something like AK47 bullets you would easily run out of PhotonViewIDs and this is a waste of performance.

2. Interest management (using PhotonView.group)

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. PUN does not yet offer server based interest management. However, client side implementations are already possible.

```
PhotonNetwork.SetReceivingEnabled (int group, bool enabled);
PhotonNetwork.SetSendingEnabled (int group, bool enabled);
```

Every PhotonView has a group, by default group 0. The group can be set when using PhotonNetwork.Instantiate or manually by changing photonView.group. Note that a manual change of photonView.group is *not* networked automatically. When you instantiate a prefab via

PhotonNetwork.Instantiate the group is automatically set everywhere.

The sole purpose of groups is the use of the SetSending and SetReceiving methods quoted above. If a group is disabled using SetSending, all local outgoing network messages will be ignored for all PhotonViews using this group. When using SetReceiving, all photonviews using the disabled 'SetReceiving' group *will* receive all messages internally, but they are dropped by the PUN API. So please note that SetReceiving will not save you performance currently.

The amount of groups is only limited by the size of integers.

3. Anti cheating

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.

You can make use of the (authoritative) master client to keep an eye out for cheating. But do note that the master client might be a cheater itself. It could be an option to have clients double check and correct each other.

4. 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 have the client also calculate its predicted movement to reduce wait times. This smoothes the players experience for its 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 sent (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: You know the FROM and TO values, so you can smoothly *lerp* between these values.

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: We use the last known values and 'extrapolate' these to generate likely future values.

An example of interpolation/extrapolation is available used in the FPS example of this tutorial.

Increasing your network update rate is not a good solution to as every message will still have the same delay as with a lower update rate.

5. Manually allocate PhotonView ID's

When you want full control over all network objects, or when you don't want to rely on the use of the Resources folder for automatically spawning PhotonNetwork.Instantiate-ed objects, you'll need to manually assign PhotonViews in runtime.

To create a PhotonView on all connected clients, one owner will need to allocate a new PhotonViewID and send this over the network. It comes down to the following code:

```
public GameObject playerPrefab;

void SpawnNew(){
       PhotonViewID id1 = PhotonNetwork.AllocateViewID();
       photonView.RPC("SpawnOnNetwork", PhotonTargets.AllBuffered, pos, rot, id1);
}

[RPC]
void SpawnOnNetwork(Vector3 pos, Quaternion rot, PhotonViewID id1,)
{
        GameObject newPlayer = Instantiate(playerPrefab, pos, rot) as GameObject ;
        PhotonView[] nViews = newPlayer.GetComponentsInChildren<PhotonView>();
        nViews[0].viewID = id1;
}
```

Note that this code will need at least one existing scene PhotonView in order to send the instantiation RPC.

For a working example, refer back to tutorial 4.

6. Manual cleanup: **PhotonNetwork.autoCleanUpPlayerObjects=false**

By default, PUN will completely clean up a players buffered messages and active objects when it leaves the room. There are cases where this is undesired, i.e. think of Example5: sometimes we leave our room to check if we can join any other rooms. When leaving our own room we don't want our player object destroyed. In this case we can disable autoCleanup to gain more control over destroying objects. However, with power comes responsibility: As soon as you disable autocleanup you'll need to cleanup everything manually. This isn't as bad as it sounds thanks to some helper functions.

```
Destroy(PhotonView view)
Destroy(GameObject go)
DestroyPlayerObjects(PhotonPlayer destroyPlayer)
RemoveAllInstantiatedObjects()
RemoveAllInstantiatedObjects(PhotonPlayer targetPlayer)
RemoveRPCs()
RemoveRPCs(PhotonPlayer targetPlayer)
RemoveAllBufferedMessages()
RemoveAllBufferedMessages(PhotonPlayer targetPlayer)
RemoveRPCs(PhotonView view)
RemoveRPCsInGroup(int group)
```

Typically, after a client leaves with autoCleanup turned off, the following set up could be used.

```
    void OnPhotonPlayerDisconnected(PhotonPlayer player)
    {
        RemovePlayer(player);
    }

    void RemovePlayer(PhotonPlayer networkPlayer)
    {
        PlayerInfo5 thePlayer = GetPlayer(networkPlayer);

        if (PhotonNetwork.isMasterClient)
        {
            PhotonNetwork.DestroyPlayerObjects(networkPlayer);
            PhotonNetwork.RemoveAllBufferedMessages(networkPlayer);
        }
        if (thePlayer.transform)
        {
            Destroy(thePlayer.transform.gameObject);
        }
        playerList.Remove(thePlayer);
    }
```

7. Connectivity

With connectivity being a big problem when using Unity Networking, I can imagine you wonder what the situation is with PUN. Well, luckily there are near to no problems at all: Normal home computers have no trouble connecting and not even mobile devices need any special setup. This is of course all thanks to the fact that the Photons servers are dedicated and have their network properly configured. The only connectivity problems you can ever stumble upon are when either the clients connection drops, or when some strict (business) firewall is blocking non-standard connections. Connectivity couldn't be better.

8. Network and level loading

From a network perspective it doesn't matter what is running on every client PC as long as the network communication runs smoothly. The network only cares about the right PhotonViews being present (and specifically their viewIDs). This means that you can have a client running a game scene, whilst a new client just connected to the same room but via the mainmenu lobby scene. This is often a problem because the master client will try to send the new client all buffered RPC and Instantiation messages. For this reason you'd better shutdown the network communication on the client temporary while loading the game. This can be done by calling “PhotonNetwork.isMessageQueueRunning=false;” on the client right after the server connection was successful. Example 2 shows you how this works.

For hassle free level loading you can also call the following method that will automatically take care of the message queue:

```
PhotonNetwork.LoadLevel( levelIDorName );
```

9. Room properties

PUN exposes room properties as an easy way to save your game settings. One important use of these properties is exposing your games settings (map, game mode). This is explained below.

If a client is in the main menu and wants to join a specific room (which is in-game), it will need to know some data, such as what level it needs to load. However, you can’t simply connect and ask some data via RPCs. If you connect to an existing room it will send all buffered calls and instantiate network objects right away. If a client is still in the main menu those calls will generate errors. Instead, we need to set the map in the room properties. Room properties are “private” by default, meaning they are only available when inside that room. You can expose some properties in the lobby as well, to do so you need to specify which of the properties will be exposed in the lobby.
First, we create our room. At this stage we have to pre-define that the map property will have to be made available to other users in the lobby. First we set the map property in our customProps Hashtable. CreateRoom takes a string[] argument with a list of the exposed properties. We use the *exposedProps* list for that

```
void CreateMyRoom(){
     Hashtable customProps = new Hashtable();
     customProps["map"] = "TheCityLevel";

     string[] exposedProps = new string[1];
```

```
    exposedProps[0]= "map";

    PhotonNetwork.CreateRoom(“MyRoomName”, true, true, 10, customProps, exposedProps);
}
```

setting exposed lobby properties

To read the exposed room properties inside the lobby, we can use the following snippet.

```
foreach (RoomInfo roomInfo in PhotonNetwork.GetRoomList()){
   GUILayout.BeginHorizontal();
   GUILayout.Label(roomInfo.name + " - Players: "+roomInfo.playerCount);
   GUILayout.Label("Map: "+roomInfo.customProperties["map"]);
   if (GUILayout.Button("JOIN"))
   {
       PhotonNetwork.JoinRoom(roomInfo);
   }
   GUILayout.EndHorizontal();
}
```

reading room properties in the lobby GUI code

10. Versioning: keeping different game versions separated

PUN saves its version in PhotonNetwork.versionPUN. Also, remember that you define your game version when connecting to Photon:

```
PhotonNetwork.ConnectUsingSettings(string gameVersion);
PhotonNetwork.Connect(string serverAddress, int port, string appID, string gameversion)
```

The Photon server will automatically separate users that use different PUN or game versions. So a player will only see the rooms for which *both* the PUN and the game versions match exactly. You’ll never have to worry about incompatible versions as long as you update your game version.

11. Improve performance

Photon has been optimized for performance over the years. However, PUNs focus is also on usability, so there have been decisions in favor of usability that cost some performance. If you need more performance you can start by tweaking the following PUN settings.

- RPCs: send a byte identifier instead of method string

- PhotonViewIDs: send a short instead of an int. This does limit the amount of max network views though.

12. Statistics

In order to optimize your network messages you can make use of the provided *PhotonStatsGui.cs* file. Simply add it to your game to view real-time in-game network statistics. Next to these specific in-game statistics the following three counters are also available. These are updated as long as you are connected to the master server (in the lobby state)

```
// All players connected using this APPID
PhotonNetwork.countOfPlayers
//All players in the lobby stage (not inside a room) for this APPID
PhotonNetwork.countOfPlayersOnMaster
//The count of players currently inside a room
PhotonNetwork.countOfPlayersInRooms
// Total of rooms for this APPID
PhotonNework.countOfRooms
```

Unity tips

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 two clients from the editor, this does mean you need to apply your (code) 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/"
```

Converting a Unity Networking project to PUN

PUN has a built in converter that can convert a UN project to PUN, about 90% of the work is automated. This converter will do the following for you:

- Convert all NetworkViews -> PhotonViews (For prefabs and scenes)
- PUN requires PhotonNetwork.Instantiated prefabs to be in a resources folder, therefore it will ask your permission to move prefabs under a Resources folder.
- All scripts will be converted: Network.* methods are replaced with their PUN equivalent.

While we have never seen a project blow up, we do advise you to backup your project before running the conversion.
After the conversion there are always some loose ends that you'll have to fix manually . Mostly these are related to GUI work where Unity networking required IP addresses and ports or the obsolete masterserver hostlist polling. Luckily PUN makes this work much simpler.

See: Windows -> Photon Unity Networking -> Converter -> Start