Photon Unity Networking Guide

02 July 2013

This is the free version of the PUN guide. The *full guide* includes 6 months cloud hosting and more documentation and examples.

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

Note: This is the free version of the PUN guide. The ***full guide*** includes 6 months cloud hosting and more documentation and examples.

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).

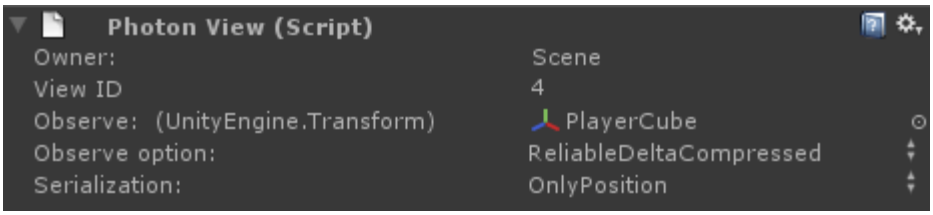
Part 2A1

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.



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 'observe' 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

Available in the full guide (includes 6 months cloud hosting).

Tutorial 4: Manually instantiating PhotonViewIDs

Available in the full guide (includes 6 months cloud hosting).

Examples

Example 1: Chatscript

Available in the full guide (includes 6 months cloud hosting).

Example 2: Game list

Available in the full guide (includes 6 months cloud hosting).

Example 3: Lobby system

Available in the full guide (includes 6 months cloud hosting).

Example 4: FPS game

Available in the full guide (includes 6 months cloud hosting).

Example 5: Multiplayer without any GUI

Available in the full guide (includes 6 months cloud hosting).

Further network subjects explained

Available in the full guide (includes 6 months cloud hosting).

1. Best practices

2. Interest management (using PhotonView.group)

3. Anti cheating

4. Combat Lag: prediction, extrapolation and interpolation

5. Manually allocate PhotonView ID's

6. Manual cleanup: PhotonNetwork.autoCleanUpPlayerObjects=false

7. Connectivity

8. Network and level loading

9. Room properties

10. Versioning: keeping different game versions separated

11. Improve performance

12. Statistics

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