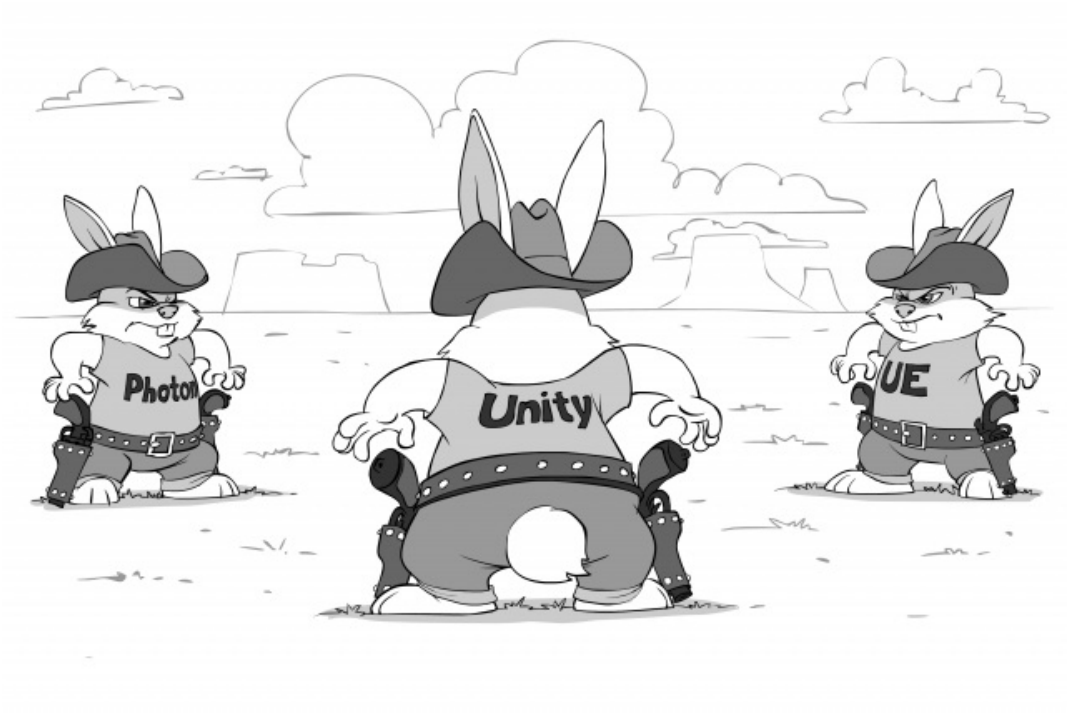
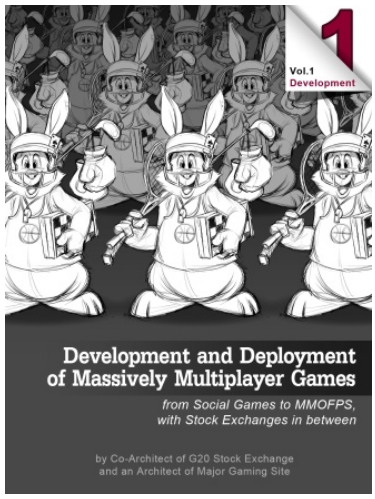
Unity 5 vs UE4 vs Photon vs DIY for MMO

posted February 22, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko



[[*This is Chapter VIII from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



By this point, we've discussed all the major parts of Modular MMOG Architecture. Now we are in a good position to take a look at some of the popular game engines and their support for MMOG, aiming to find out how they support those features which we've described for Modular Architecture.

DIY

Do it yourself, also known as DIY, is the method of building, modifying, or repairing something

There are lots of game engines out there, so we'll consider only the most popular ones: Unity 5, Unreal Engine 4, and Photon Server (which is not a game engine in a traditional sense, but does provide MMOG support on top of the existing game engines). Note that comparing graphics advantages and disadvantages of Unity vs UE, as well as performance comparisons, pricing, etc. are out of scope; if you want to find discussion on these issues, Google "Unity 5 vs UE4", you will easily find a ton of comparisons of their non-network-related features. We, however, are more interested in network-related things, and these comparisons are not that easy to find (to put it mildly). So,

Let the comparison begin!

> **without the direct aid of experts or professionals**
> — *Wikipedia* —

Unity 5

Unity 5 is a very popular (arguably *the most* popular) 3D/2D game engine. It supports tons of different platforms (HTML5 support via IL2CPP+emscripten included, though I have no idea how practical it is), uses .NET CLI/CLR as a runtime, and supports C#/JS/Boo (whatever the last one is) as a programming language. One thing about Unity is that it targets a very wide range of games, from first-person shooters to social games (i.e. "pretty much anything out there").

As usual, support of CLI on non-MS platforms requires Mono which is not exactly 100% compatible with CLR, but from what I've heard, most of the time it works (that is, as long as you adhere to the "write once – test everywhere" paradigm).

> **"As usual, support of CLI on non-MS platforms requires Mono which is not exactly 100% compatible with CLR, but from what I've heard, most of the time it works**

Another thing to keep in mind when dealing with Unity is that CLR (as a pretty much any garbage-collected VM, see discussion in Chapter VI) suffers from certain potential issues. These issues include infamous "stop-the-world"; for slower games it doesn't really matter, but for really fast ones (think MMOFPS) you'll need to make sure to read about mitigation tricks which were mentioned in Chapter VI.

Event-driven Programming/FSMs

Unity is event-driven by design. Normally the game loop is hidden from sight, but it does exist "behind the scenes", so everything basically happens in the same thread,[1] so you don't need to care about inter-thread synchronisation, phew. From our point of view, Unity is an ad-hoc FSM as defined in Chapter V.

In addition, Unity encourages co-routines. They (as co-routines should) are executed within the same thread, so no inter-thread synchronisation is necessary. For more discussion on co-routines and their relation to other asynchronous handling mechanisms, see Chapter VI. [[TODO! – add discussion on co-routines there]]

One thing Unity event-driven programs are missing (compared to our ad-hoc FSMs discussed in Chapter V) is an ability to serialise the program state; it implies that Unity (as it is written now) can't support such FSM goodies discussed in Chapter V as production post-mortem, server fault tolerance, replay-based testing, and so on. While not fatal, this is a serious disadvantage, *especially when it comes to debugging of distributed systems (see "Distributed Systems: Debugging Nightmare" section in Chapter V for relevant discussion).*

---

[1] or at least "as if" it happens in the same thread

Unity for MMOG

When using Unity for MMOG, you will notice that it deals with one single Game World, and that separation between Client and Server is quite rudimentary. In "Engine-Centric Development Flow" section below we'll see that this might be either a blessing (if your game is more on "Client-Driven Development Flow" side) or a curse (for "Server-Driven Development Flows"). On the other hand, in any case it is not a show-stopper.

Communications: HLAPI

Communication support in Unity 5 (known as UNet) is split into two separate API levels: High-Level API (HLAPI), and Transport-Level API (LLAPI). Let's take a look at HLAPI first.

"You SHOULD NOT use Command requests to allow the client to modify state of the PC on the server directly

One potential source of confusion when using HLAPI, is an HLAPI term "Local Authority" as used in [UNet]. When the game runs, HLAPI says that usually a client has an "authority" over the corresponding PC. It might sound as a bad case of violating the "authoritative server" principle (that we need to avoid cheating, see Chapter III), but in fact it isn't. In HLAPI-speak, "client authority" just means that the client can send [Command] requests to the server (more on [Command]s below), that's pretty much it, so it doesn't necessarily give any authority to the client, phew.

On the other hand, you SHOULD NOT use [Command] requests to allow the client to modify state of the PC on the server directly; doing *this* will violate server authority, widely opening a door for cheating. For example, if you're allowing a Client to send a [Command] which sets PC's coordinates directly and without any server-side checks, you're basically inviting a trivial attack when a PC controlled by a hacked client can easily teleport from one place to another one. To avoid it,

instead of making decisions on the client-side and sending *coordinates* resulting from player's inputs, you should send the *player's inputs* to the server, and let the (authoritative) server simulate the world and decide where the player goes as a result of those inputs

State Synchronization

In HLAPI, basically you have two major mechanisms – "state synchronization" and RPCs.

State synchronization is a Unity 5's incarnation of Server State -> Publishable State -> Client State process which we've discussed in Chapter VII. In Unity 5, state synchronization can be done via simple adding of [SyncVar] tag to a variable [UNetSync], it is as simple as that.

Importantly, Unity does provide support for both distance-based and custom interest management. Distance-based interest management is implemented via NetworkProximityChecker, and custom one – via RebuildObservers() (with related OnCheckObservers()/OnRebuildObservers()).

For quite a few games, you will need to implement Interest Management. Not only it helps to reduce traffic, it is also necessary to deal with "see through walls" and "lifting fog of war" cheats[2]

On top of [SyncVars], you may need to implement some (or *all*) of the Client-Side stuff discussed in Chapter VII (up to and including Client-Side Prediction); one implementation of Client-Side Prediction for Unity is described in [UnityClientPrediction].

So far so good, but the real problems will start later. In short – such synchronization is usually quite inefficient traffic-wise. While Unity seems to use per-field delta compression (or a reasonable facsimile), it cannot possibly implement most of the compression which we've discussed in "Compression" section of Chapter VII. In particular, restricting precision of Publishable State is not possible (which in turn makes bitwise streams pretty much useless), dead reckoning is out of question, etc. Of course, you can create a separate set of variables just for synchronization purposes (effectively creating a Publishable State separate from your normal Client State), but even in this case (which BTW will require quite an effort, as well as being a departure from HLAPI philosophy, even if you're formally staying within HLAPI) you won't be able to implement many of the traffic compression techniques which we've discussed in Chapter VII.

"While Unity does use per-field delta compression (or a reasonable facsimile), it cannot possibly implement most of the compression which we've discussed in 'Compression' section of Chapter VII.

These problems do not signal the end of the world for HLAPI-based development, but keep in mind that at a certain stage you may need to re-implement state sync on top of LLAPI; more on it in "HLAPI Summary" subsection below.

---

[2] see Chapter VII for discussion on Interest Management

RPCs (a.k.a. "Remote Actions")

In Unity 5, RPCs were renamed into "Remote Actions". However, not much has changed in reality (except that now there is a [Command] tag for Client-to-Server RPC, and [ClientRpc] tag for Server-to-Client RPC). In any case, Unity RPCs still MUST be void. As it was discussed in Chapter VII, this implies quite a few complications when you're writing your code. For example, if you need to query a server to get some value, then you need to have an RPC call going from client to server ([Command] in Unity), and then you'll need to use something like Networking.NetworkConnection.Send() to send the reply back (not to mention that all the matching between requests and responses needs to be done manually). In my books[3] it qualifies as "damn inconvenient" (though you certainly *can* do things this way).

In addition, Unity HLAPI seems to ignore server-to-server communications completely.**[[PLEASE CORRECT ME IF I'M WRONG HERE]]**

---

[3] pun intended

HLAPI summary

"You can still use HLAPI despite its shortcomings

As noted above, for quite a few simulation games, HLAPI's [SyncVar] won't provide "good enough" traffic optimization. But does it make HLAPI hopeless? IMHO the answer is "no, you can still use HLAPI despite its shortcomings". HLAPI's [SyncVar] will work reasonably good for early stages of development (that includes testing, and probably even over-the-Internet small-scale testing), speeding development up. And then, when/if your game is almost-ready to launch (and *if* you're not satisfied with your traffic measurements), you will be able to rewrite [SyncVars] into something more efficient using LLAPI. It is not going to be a picnic, and you'll need to allocate enough time for this task, but it can be done.

As for RPCs calls (and network events) – due to their only-void nature, they're not exactly convenient to use (to put it mildly), but if you have nothing better (and you won't as long as you're staying within Unity's network model) – you'll have to deal with it yourself, and will be able to do it too.**[[IF YOU KNOW SOME WORKABLE LIBRARIES PROVIDING non-void RPCs**

in Unity, PLEASE LET ME KNOW]]

In addition, I need to note that the absence of support for Server-to-Server communications is very limiting for quite a few games out there. Having your server side split into some kind of micro-services (or even better, Node.js-style nodes) is a must for any sizeable server-side development, and having your network/game engine to support interactions between these nodes/micro-services is extremely important. While manual workarounds to implement Server-to-Server communications in Unity are possible, doing it is a headache, and integrating it with game logic is a headache even more ☹. This is probably one of the Biggest Issues you will face when using Unity for a serious MMOG development.

Communications: LLAPI

> "Just as advertised, Unity Transport Layer API (a.k.a. LLAPI), is an extremely thin layer on top of UDP.

Just as advertised, Unity Transport Layer API (also known as LLAPI[4]), is an extremely thin layer on top of UDP. There is no RPC support, no authentication, no even IDL or marshalling (for this purpose you can use .NET BinaryFormatter, Google Protocol Buffers, or write something yourself).

For me, the biggest problem with LLAPI lies with its IP:Port addressing model. Having to keep track at application level all those IP/port numbers is a significant headache, especially as they can (and will) change. Other issues include lack of IDL (which means manual marshalling for any not-so-trivial case, and discrepancies between marshalling of different communication parties tend to cause a lot of unnecessary trouble), lack of explicit support for state synchronization, and lack of RPCs (even void RPCs are better than nothing from development speed point of view).

On the positive side, LLAPI provides you with all capabilities in the world – that is, as long as you do it yourself. Still, it is *that* cumbersome that I'd normally suggest to avoid it at earlier stages of development, and introduce only when/if you have problems with HLAPI.

---

[4] don't ask why it is named LLAPI and not TLAPI

Unity 5/UNet Summary

> "All-in-all, Unity 5/UNet does a decent job if you want to try converting your existing single-player game into a low-player-number multi-player one.

All-in-all, Unity 5/UNet does a decent job if you want to try converting your existing single-player game into a low-player-number multi-player one. On the other hand, if you're into serious MMO development (with thousands of simultaneous players), you're going to face quite a few significant issues; while not show-stoppers, they're going to take a lot of your time to deal with (and if you don't understand what they're about, you can easily bring your whole game to the knees).

If going Unity way, I would suggest to start with HLAPI to get your game running as a MMO. Most likely, when using HLAPI for a serious MMOG, you'll face traffic problems with replicated states (and/or cheating) when number of players goes up, but to have your prototype running HLAPI is pretty good. At this stage you'll probably need to rewrite the handling of your Publishable State, most likely on top of LLAPI. This rewrite can include all those optimizations we've spoke about, and is going to be quite an effort. On the positive side, it can usually be done without affecting the essence of your game logic, so with some luck and experience, it is not going to be *too bad*.

Additionally, you'll also have issues with server-to-server communications (which are necessary to split your servers into manageable portions). You can either

implement these on top of LLAPI, or to use good old TCP sockets, but in any case you will stay even without RPCs, just with bare messages. While I've seen such message-based architectures to work for quite large projects, they are a substantial headache in practice ☹ .

At this point you might think that your problems are over, but actually the next problem you're going to face, is likely to be at least as bad as the previous ones. As soon as a number of your players goes above a few hundred, you'll almost certainly need to deal with load balancing (see Chapter VI for discussion on different ways of dealing with load balancing). And Unity as such won't help you for this task, so you'll need to do it yourself. Once again, it is doable, but load balancing is going to take a lot of efforts to do it right ☹ .

Unreal Engine 4

Unreal Engine 4 is a direct competitor of Unity, though it has somewhat different positioning. Unlike Unity (which tries to be a jack of all trades), Unreal Engine is more oriented towards first-person games, and (arguably) does it better. Just as Unity, UE also supports a wide range of platforms (with differences from Unity being of marginal nature), and does have support for HTML (also using emscripten, and once again I have no idea whether it really works[5]).

"Unreal Engine is more oriented towards first-person games, and (arguably) does it better.

As of UE4, supported programming languages are C++ and UE's own Blueprints. At some point, Mono team has tried to add support for C# to UE4, but dropped the effort shortly afterwards ☹ .

It should be noted that UE4's variation of C++ has its own garbage collector (see, for example, [UnrealGC]). Honestly, I don't really like hybrid systems which are intermixing manual memory management with GC (they introduce too many concepts which need to be taken care of, and tend to be rather fragile as a result), but Unreal's one is reported to work pretty well.

---

[5] as of beginning of 2016, support for HTML5 in UE4 is tagged Experimental

Event-driven Programming/FSMs

Unreal Engine is event-driven by design. As with Unity, normally game loop is hidden from sight, but you can override and extend it if necessary. And exactly as with Unity or our FSMs, everything happens within the same thread, so (unless you're creating threads explicitly) there is no need for thread synchronization.

On the negative side of things, and also same as Unity, UE's event-driven programs don't have an ability to serialize the program state, and (same as with Unity), it rules out certain FSM goodies.

UE for MMOG

Just like Unity, UE doesn't really provide a way to implement a clean separation between the client and the server code (while there is a WITH_SERVER macro for C++ code, it is far from being really cleanly separated). More on advantages and disadvantages of such "single-Game-World" approach in "Engine-Centric Development Flow" section below.

UE Communications: very close to Unity 5 HLAPI

"There is not much to discuss here, as both replication and RPCs are very close to Unity counterparts which were discussed above.

Just like Unity, UE4 has two primary communication mechanisms: state synchronization ("Replication" in UE-speak), and RPCs. There is not much to discuss here, as both replication and RPCs are very close to Unity counterparts which were discussed above.

In particular, replication in UE4 is very similar to Unity's [SyncVars] (with a different syntax of UPROPERTY(Replicated) and DOREPLIFETIME()). UE4's RPCs (again having a different syntax of UFUNCTION(Client)/UFUNCTION(Server)) are again very similar to that of Unity HLAPI (with the only-void restriction, no support for addressing and for server-to-server communications, and so on).

Interest management in UE4 is based on the concept of being "network relevant" and is dealt with via AActor::NetCullDistanceSquared() and AActor::IsNetRelevantFor() functions (ideologically similar to Unity's NetworkProximityChecker and RebuildObservers respectively).

Being so close to Unity 5 means that UE4 also shares all the drawbacks described for Unity HLAPI above; it includes sub-optimal traffic optimization for replicated variables, void-only RPCs, and lack of support for server-to-server communications; see "HLAPI summary" section above for further discussion.

On the minus side compared to Unity 5, UE4 doesn't provide LLAPI, so bypassing these drawbacks as it was suggested for Unity, is more difficult. On the other hand, UE4 does provide classes to work directly over sockets (look for FTcpSocketBuilder/FUdpSocketBuilder), and implementing a (very thin) analogue of LLAPI is not *that much* of a headache. So, even in this regard the engines are very close to each other. As a result, for UE4 MMO development I still suggest about-the-same development path as discussed in "Unity 5/UNet Summary" section for Unity, starting from Replication-based game, and moving towards manually controlled replication (implemented over plain sockets) when/if the need arises.

Photon Server

Photon Server is quite a different beast from Unity and Unreal Engine: unlike Unity/UE, Photon isn't an engine by itself, but is rather a way to extend a game developed using an existing engine (such as Unity or Unreal) into an MMO. It is positioned as an "independent network engine", and does as advertised – adds its own network layer to Unity or to Unreal. As a result, it doesn't need to care about graphics etc., and can spend more effort of MMO-specific tasks such as load balancing and matchmaking service.

"Photon Server is quite a different beast from Unity and Unreal Engine

As Photon is always used on top of existing game engine,[6] it is bound to inherit quite a few of its properties; this includes using game engine graphics and most of scripting. One restriction of Photon Server is that server-side always runs on top of Windows .NET and APIs are written with C# in mind (I have no idea how it feels to use other .NET languages with Photon, and whether Photon will run reasonably good on top of Mono). For the client-side, however, Photon supports pretty much every platform you may want, so as long as you're ok with your *servers* being Windows/.NET – you should generally be fine.

Functionally, Photon Server is all about simulated worlds consisting of multiple rooms; while it can be considered a restriction, this is actually how most of MMOs out there are built anyway, so this is not as limiting as it may sound. In short – as we've discussed it briefly in Chapter VII [[TODO! – add discussion on Big Fat World there]], if your MMO needs to have one Big Fat World, you'll need to split it into multiple zones anyway to be able to cope with the load.

Within Photon Server, there are two quite different flavours for networked game development: Photon

Server SDK and Photon Cloud / PUN.[7]

---

[6] or should I rather say *underneath* existing game engine?
[7] Exit Games also provide Photon/Realtime and Photon/Turnbased cloud products, but I know too little about them to cover them here [[TODO! – try to learn more about them]]

Photon Server SDK: Communications

IMPORTANT: Photon Server SDK is not to be confused with Photon Cloud/PUN, which will be discussed below.

Unfortunately, personally I didn't see any real-world projects implemented over Photon Server SDK, and documentation on Photon Server SDK is much less obvious than on Photon Cloud and PUN, so I can be missing a few things here and there, but I will try my best to describe it. **[[PLEASE CORRECT ME IF I'M MISSING SOMETHING HERE]]**

"Photon Server SDK doesn't explicitly support a concept of synchronized state

First of all, let's note that Photon Server SDK doesn't explicitly support a concept of synchronized state. Instead, you can BroadcastEvent() to all connected peers, and handle this broadcast on all the clients to implement state synchronization. While BroadcastEvent() can be used to implement synchronized state, there is substantial amount of work involved in making your synchronization work reliably (I would estimate the amount of work required to be of the same order of magnitude as implementing synchronised states on top of Unity's LLAPI). In addition, keep in mind that when relying on BroadcastEvent(), quite a few traffic optimizations won't work, so you may need to send events to individual clients (via SendEvent()).

From RPC point of view, Photon Server does have *kinda*-RPC. Actually, while it is named Photon.SocketServer.Rpc, it is more like message-based request-response than really a remote procedure call as we usually understand it. In other words, within Photon Server (*I'm not speaking about PUN now)* I didn't find a way to declare a function as an RPC, and then to call it, with all the stubs being automagically generated for you. Instead, you need to create a *peer,* to send an *operation request* over the peer-to-peer connection, and while you're at it, to register an *operation handler* to manage *operation response.*

This approach is more or less functionally equivalent to Take 1 from "Take 1. Naïve Approach: Plain Events (will work, but is Plain Ugly)" section of Chapter VI; as Take 1 is not the most convenient thing to use (this it to put it very mildly), it will become quite a hassle to work with it directly. In addition, I have my concerns about Peer.SetCurrentOperationHandler() function, which seems to restrict us to one outstanding RPC request per peer, which creates additional (and IMHO unnecessary) hassles.

On the positive side (and unlike all the network engines described before), Photon Server does support such all-important-for-any-serious-MMO-development features as Server-to-Server communication and Load Balancing.

"On the positive side (and unlike all the network engines described

Photon Cloud / PUN: Communications

IMPORTANT: Photon Cloud / PUN is not to be confused with Photon Server SDK, which is discussed above.

The second flavour of Photon-based development is Photon Cloud with Photon Unity Networking (PUN). While Photon Cloud/PUN is implemented on top of Photon Server which was discussed above, the way Photon Server is deployed for Photon Cloud/PUN, is very different from the way you would develop your own game on top of Photon Server SDK 🙁 .

The key problem with Photon Cloud is that basically you're not allowed to run your own code on the server. While there is an exception for so-called "Photon Plugins", they're relatively limited in their abilities, and what's even even worse, they require an "Enterprise Plan" for your Photon Cloud (which as of beginning of 2016 doesn't even have pricing published saying "contact us" instead, ouch).

before), Photon Server does support such features as Server-to-Server communication and Load Balancing.

And as long as you're not allowed to run your own code on the server-side, you cannot make your server authoritative, which makes dealing with cheaters next-to-impossible. That's the reason why I cannot recommend PUN for any serious MMO development, at least until you (a) realize how to deal with cheaters given limited functionality of Photon Plugins, and (b) get a firm quote from Exit Games regarding their "Enterprise Plan" (as noted above, lack of publicly available quote is usually a pretty bad sign of it being damn expensive 🙁).[8]

This restriction is a pity, as the rest of PUN is quite easy to use (more or less in the same range as Unity HLAPI, but with manual serialization of synchronization states, what is IMHO more of a blessing rather than a curse, as it allows for more optimizations than [SyncVars]). Still, unless you managed to figure out how to implement an authoritative server over PUN (and how to pay for it too), I'd rather stay away from it, because any game without an authoritative server carries too much risk of becoming a cheaterfest.

---

[8] BTW, I do sympathize Chris Wegmann in this regard and do realize that allowing foreign code on servers opens more than just one can of worms, but still having an authoritative server is *that* important, that I cannot really recommend Photon Cloud for any serious MMO

"I want YOU to read page 2!

Keep reading for a Huge Table comparing 40+ different network-related parameters of Unity 5, UE4, and Photon

Summary

The discussion above (with some subtle details added too) is summarized in the table below.

In this table, the rightmost column represents what I would like to see from my own DIY game network engine. In this case, while the network engine itself is DIY, there is a big advantage of pushing all these things into the network engine and to separate them from the game logic. The more things are separated via well-defined interfaces, the less cluttered your game logic code becomes, and the more time you have for really important things such as gameplay; in the extreme case, this difference can even mean the difference between life and death of your project. Also keep in mind that if going a DIY route, for any given game you won't need to implement *all* the stuff in the table; think what is important for *your* game, and concentrate only on those features which you really need. For example, UDP support and dead reckoning are not likely to be important for a non-simulation game, and HTTP polling isn't likely to work for an MMOFPS.

[[PLEASE CORRECT ME IF SOMETHING LOOKS WRONG HERE!]]

Features (those IMO most important ones are **in bold**)	**Unity 5 (HLAPI)**	**Unity 5 (LLAPI)**	**Unreal Engine 4**	**Photon Server SDK**	**Photon Cloud / PUN**	**My ideal DIY network engine (along the lines of this book)**
Platforms						
Desktop	Win / MacOS / SteamOS		Win / MacOS / SteamOS	Win / MacOS		Whatever tickles your fancy
Consoles	PS / Xbox / Wii		PS / XBox	PS / Xbox / Wii		Whatever floats your boat
Mobile	IOS / Android / WinPhone		iOS / Android	iOS / Android / WinPhone		Whatever butters your biscuit
HTML 5	Yes / Websockets		Experimental	Yes / Websockets		Yes
Server	Windows / Linux		Windows / Linux	Windows Only		Windows / Linux
Languages						
C/C++	Sort Of[9]		Yes[10]	Client Only[11]		Yes
Garbage-Collected	C#/CLI		No[12]	C#/CLI		C#/Any, Java/Any, etc.
Scripting	JS/CLI, Boo/CLI		"Blueprints"	Client Only		JS/Any (incl JS/V8 and Node.js), Python/Any, etc.
	Unity 5 (HLAPI)	**Unity 5 (LLAPI)**	**Unreal Engine 4**	**Photon Server SDK**	**Photon Cloud / PUN**	**My ideal DIY**

Programming						
Event-driven	Yes	Yes	Yes	Yes	Yes	Yes
Deterministic Goodies[13]	No	No	No	No	No	Yes[14]
void non-blocking RPCs	Yes	No	Yes	No	Yes	Yes
non-void non-blocking RPCs	No	No	No	No	No	Yes
Futures for RPCs	No	No	No	No	No	Yes[15]
Co-routines	Yes	Yes	No	Yes	Yes	Yes[16]
Clear Client-Server Separation	No (favors Client-Driven Development Flow)	No (favors Client-Driven Development Flow)	No (favors Client-Driven Development Flow)	Yes (favors Server-Driven Development Flow)	No (favors Client-Driven Development Flow)	Whatever you prefer
Graphics [17]						
3D		Yes	Yes	External: Unity, Unreal Engine		External[18]
2D		Yes	Yes	External: Cocos2X		External[19]
Model-View-Controller		DIY	DIY	DIY	No	Yes
2D+3D Views on the same game		No	No	DIY	No	Yes
	Unity 5 (HLAPI)	**Unity 5 (LLAPI)**	**Unreal Engine 4**	**Photon Server SDK**	**Photon Cloud / PUN**	**My ideal DIY**
Networking – General						
Support for Authoritative Server	**Yes**	**Yes**	**Yes**	**Yes**	**No**[20]	**Yes**
Networking – Marshalling/IDL						
					In-	

IDL	In-Language	No	In-Language	No	Language[21]	External
State Synchronization	Yes	DIY	Yes	DIY	DIY	Yes
Clear Server-State – Publishable State – Client State separation	No[22]	N/A	No[22]	N/A	No[22]	Yes
Cross-language IDL	No	N/A	No	No	N/A	Yes
IDL Encodings	No	N/A	No	No	N/A	Yes
IDL Mappings	No	N/A	No	No	N/A	Yes
Interest Management	Yes	DIY	Yes	DIY	DIY	Yes
Client-Side Interpolation	DIY	DIY	DIY	DIY	DIY	DIY
Client-Side Extrapolation	DIY	DIY	DIY	DIY	DIY	DIY
Client-Side Prediction	DIY	DIY	DIY	DIY	DIY	DIY
Delta Compression (whole fields)	Automatic	DIY	Automatic	DIY	DIY	Controlled
Delta Compression (field increments)	No	DIY	No	DIY	DIY	Yes
Variable Ranges, Rounding-when-Transferring, and Bit-Oriented Encodings	No	DIY	No	DIY	DIY	Yes
Dead Reckoning	No	DIY	No	DIY	DIY	Yes

Revision-Based Large Objects Sync [[TODO! Add to Chapter VII]]	No	DIY	No	DIY	DIY	Yes
VLQ	No	DIY	No	DIY	DIY	Yes
Huffman coding	No	DIY	No	DIY	DIY	Yes
IDL Backward Compatibility Support	No	N/A	No	No	N/A	Yes
	Unity 5 (HLAPI)	**Unity 5 (LLAPI)**	**Unreal Engine 4**	**Photon Server SDK**	**Photon Cloud / PUN**	**My ideal DIY**
Networking – Addressing/Authentication						
Addressing Model	"Client" / "Server"[23]	IP:Port[24]	"Client" / "Server"[23]	IP:Port[24]	"Client" / "Server"[23]	By server name for servers, player ID / "connected client" for players
Player Authentication	DIY	DIY	DIY	DIY	DIY	Yes
Server-to-Server Communications	No	DIY	No	Yes	No	Yes
Networking – Supported Protocols						
UDP	Yes	Yes	Yes	Yes	Yes	Yes
TCP	No[25]	No[25]	No	Yes	Yes	Yes
WebSockets	Yes (only for WebGL apps?)		?	Yes	Yes	Yes
HTTP	No	No	No	Yes	Yes	Yes
Scalability/Deployment Features						
				Inter-World Only [[TODO!: describe	Inter-World Only [[TODO!: describe	Both Inter-

Load Balancing	No	No	No	inter-world / intra-world balancing in Chapter VI]]	inter-world / intra-world balancing in Chapter VI]]	world and Intra-world
Front-End Servers	No	No	No	No	No	Optional

[9] Unmanaged code is possible, but cumbersome
[10] actually, UE4 is using a somewhat-garbage-collected dialect of C++
[11] on server side unmanaged C++ may work
[12] Mono tried to add support for C#, but this effort looks abandoned
[13] replay testing, production post-mortem, server failure handling
[14] to enable deterministic goodies while using either futures or co-routines, a source pre-processor will be necessary
[15] to use futures with deterministic goodies enabled, a source pre-processor will be necessary
[16] to use co-routines with deterministic goodies enabled, or for a language which doesn't support them explicitly, a source pre-processor will be necessary
[17] yes, graphics comparison is intentionally VERY sketchy here
[18] in particular, can use Unity or Unreal Engine for rendering
[19] in particular, can use Cocos2X or a homegrown 2D library for rendering
[20] Photon Plugins MAY allow for a way out, but this needs separate analysis
[21] Last time I've checked, Photon has had only RPC part as declarative IDL; Publishable State was via manual serialization
[22] it is possible to separate them, but it requires efforts
[23] i.e. there is no way to address anything except for "Client" on Server, and "Server" on Client; this addressing model is too restrictive, and effectively excludes server-to-server communication
[24] quite cumbersome in practice
[25] support reportedly planned

Engine-Centric Development Flow

Ok, so we've got that nice table with lots of different things to compare. Still, the Big Question of "What should I use for my game?" remains unanswered. And to answer it, we'll need to speak a bit about different development flows (which are not to be confused with data flows(!)).

In general, for a pretty much any game being developed, there are two possible development scenarios which heavily depend on the nature of your MMO game.

Server-Driven Development Flow

"The first development

The first development scenario occurs when the logic of your MMOG does not require access to game assets. In other words, it happens when the gameplay is defined by some internal rules, and not by object geometry or levels. Examples of such games include stock exchanges, social games, casino-like games, some of simpler simulators (maybe snooker simulator), and so on.

What is important for us in this case, is that you can easily write your game logic (for your authoritative server) without any 3D models, and without any involvement of graphics artist folks. It means that for such development server-

scenario occurs when the logic of your MMOG does not require access to game assets.

side has no dependencies whatsoever, and that server-side becomes a main driver of the things, plain and simple. And all the graphical stuff acts as a mere rendering of the server world, without any ability to affect it.[26]

In this kind of Server-Driven development workflow game designers are working on server logic, and can express their ideas without referring to essentially-3D or essentially-graphical things such as game levels, character geometry, or similar.

If your game allows it, Server-Driven development is a Good Thing(tm). It is generally simpler and more straightforward than a Client-Driven one. Developing, say, a social game the other way around is usually a Pretty Bad Idea. However, not all the MMOGs are suitable for such Server-Driven development, and quite a few require a different development workflow.

---

[26] as discussed above in Chapter VII, from data flow point of view it will happen anyway when the game is running, but from game designer point of view it might be different, see "Client-Driven Development" section below

Server-Driven Development Flow: Personal Suggestions

From what I've seen and heard, if you're using one of the engines above (and not your own one), and your game is suitable for Server-Driven Development Flow, starting with Unity 5 HLAPI (and rewriting necessary portions into LLAPI when/if it becomes necessary) is probably your best bet. UE4, as it is even more simulation-world-oriented, is less likely to be suitable for the games which fit Server-Driven Development Flow, but if it is – you can do it with UE4 too.

"From what I've seen and heard, if you're using a 3rd-party game engine, and your game is suitable for Server-Driven Development Flow, starting with Unity 5 HLAPI is probably your best bet.

Photon Server SDK might work for Server-Driven development flow too, though you IMO should stay away from Photon Cloud and PUN at least until you realize how Photon Plugins will help to deal with cheaters, and figure out Photon Cloud Enterprise pricing (as noted above, Enterprise plan is necessary to run Photon Plugins).

And of course, a DIY engine can really shine for such development scenarios (using some game engine or 2D/3D engine for client-side rendering purposes).

Client-Driven Development Flow

For those games where your game designers are not only laying out the game rules, but are also involved in developing graphical things such as game levels, Server-Driven development flow described above, tends to fall apart fairly quickly. The problem here lies with the fact that game designers shouldn't (and usually couldn't) think in terms of servers and clients. When thinking in terms of "whenever character comes to city X and doesn't have level 19, he is struck into his face", there is no way to map this kind of the world picture into servers and clients. In such cases, from Game Designer perspective there is usually a single Game World which "lives" its own life, and introducing separation between client and server into the picture will make their job so much more difficult that their performance will be affected badly, quite often beyond any repair 🙁 .

Games which almost universally won't work well with Server-Driven development flow and will require a Client-Driven approach described below, are MMORPGs and MMOFPS.

For such games, the following approach is used pretty often (with varying degrees of success):

- develop a game using existing game engine “as if” it is a single-player game.
  - There is only one Game World, and both game designers and 3D artists who can work within a familiar environment, are able to test things right away, and so on
  - at this stage there is no need to deal with network at all: there is no [SyncVars], no RPCs, nothing
- at certain point (when the engine as such is more or less stable), start a project to separate server from the client. This may include one or more of the following:
  - dropping all the textures from the server side
  - using much less detailed meshes for the server side; in the extreme cases, your PCs/NPCs can become prisms or even rectangular boxes/parallelepipeds.
  - taking existing Game World State as a Client-State, figuring out how it can be reduced to get Server-State
  - working on further reducing Server-State for transfer purposes, obtaining Published-State
    - this process is likely to involve certain visually observable trade-offs and degradations, and is going to take a while
  - at the same time, work of game designers on high-level scripts etc., and of 3D artists on further improvements, may continue

> “Games which almost universally won't work well with Server-Driven development flow and will require a Client-Driven approach described below, are MMORPGs and MMOFPS.

[[TODO! – vigilance]]

While this Client-Driven development process is not a picnic, it is IMHO the best you can do for such games given the tools currently available. Most importantly, it allows game designers to avoid thinking too much about complexities related to state synchronization; while certain network-related issues such as “what should happen with a player when she got disconnected” will still appear in the game designer space, it is still much better than making them think about clients and servers all the time.

Client-Driven Development Flow: Personal Suggestions

If you’re using one of the engines above (and not your own one), and your game requires Client-Driven Development Flow, you may want to start with a single-player Unity 5, or with a single-player UE4. Then (as a part of “client-server separation project” described above), you will be able to proceed either to Unity 5 HLAPI, or to UE4 Replication/RPCs. And as a further step, as discussed above, you may need to rewrite state sync into LLAPI or on top of plain sockets respectively.

While Photon Server SDK might work for Client-Driven Development too, I expect it to be too cumbersome here. As for Photon Cloud and PUN – just as with Server-Driven Development workflow, you IMO still should keep away from them at least until you realize how Photon Plugins will help to deal with cheaters, and figure out Photon Cloud Enterprise pricing.

As for DIY network engine, you can certainly use it for “client-server separation” too (and that’s what I would personally suggest if you have reasonably good network developers).

> “If you're using one of the engines above (and not your own one), and your game requires Client-Driven Development Flow, you may want to start with a single-player Unity 5,

Important Clarification: Development Flow vs Data Flow

One important thing to note that regardless of game development flow being Server-Driven or Client-Driven, from the technical point of view the completed

game will always be server-driven: as our server needs to be authoritative, all decisions are always made by the server and are propagated to the clients, which merely render things as prescribed by the server (see more discussion on data flows in Chapter VII). What we're speaking about here, is only Development Flow (and yes, having development flow different from program data flow is a major source of confusion among multi-player game developers).

or with a single-player UE4.

[[To Be Continued…

This concludes beta Chapter VIII from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter IX, "Pre-Development Checklist: Things everybody hates but everybody needs too"]]

[–] References

[UNet] "Unity 5 Network System Concepts", Unity
[UNetSync] "Unity 5 State Synchronization", Unity
[UnityClientPrediction] Christian Arellano, "UNET Unity 5 Networking Tutorial Part 2 of 3 - Client Side Prediction and Server Reconciliation", Gamasutra
[UnrealGC] https://wiki.unrealengine.com/Garbage_Collection_Overview

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *IDL: Encodings, Mappings, and Backward Compatibility*

Pre-Coding Checklist: Things Everybody Hates, but Everybody Needs Them T… »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: game, multi-player, Photon, UE, unity