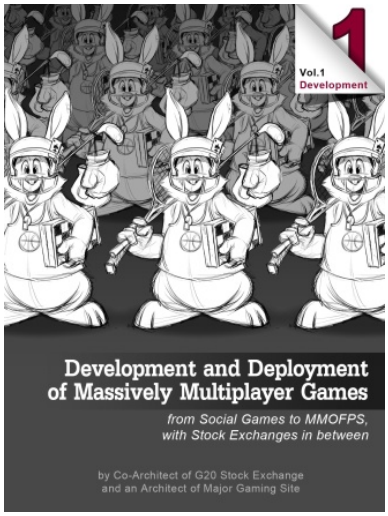
Chapter V(b). Modular Architecture: Client-Side. Programming Languages for Games, including Resilience to Reverse Engineering and Portability

posted November 30, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter V(b) 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.]]

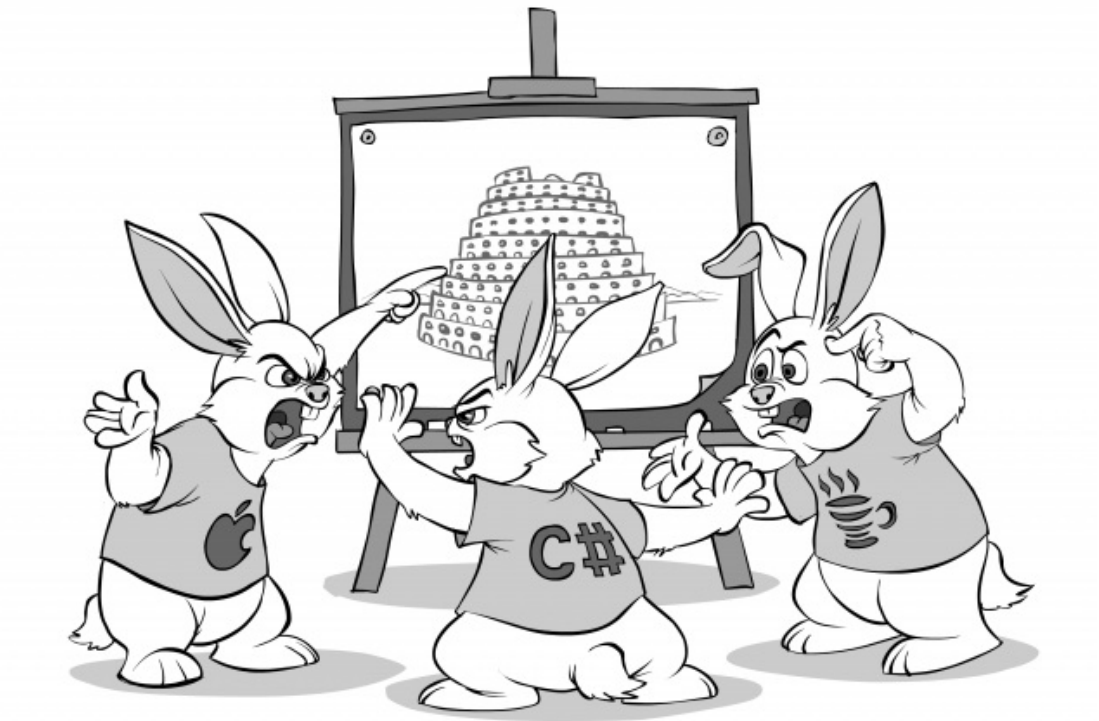


Programming Language for Game Client

Some of you may ask: "What is the Big Fat Hairy Difference between programming languages for the game client, and programming language for any other programming project?" Fortunately or not, in addition to all the usual language holy wars[1], there are some subtle differences which make programming language choice for the game client different. Some of these peculiarities are described below.

---

[1] between strongly typed and weakly typed programming languages, between compiled and scripted ones, and between imperative and functional languages, just to name a few



One Language for Programmers, Another for Game Designers

(MMORPG/MMOFPS etc.)

First of all, let's note that in quite a few (or maybe even "most") development environments, there is a practice of separating game designers from programmers (see "On Developers, Game Designers, and Artists" section in this chapter). This practice is pretty much universal for MMORPG/MMOFPS, but can be applicable to other genres too (especially if your game includes levels and/or quests designed-by-hand).

"It is quite common to have two different programming languages: one (roughly) intended for programmers, and another one (even more roughly) intended for game designers.

In such cases, it is quite common to have two different programming languages: one (roughly) intended for programmers, and another one (even more roughly) intended for game designers. For example, Unreal Engine 4 positions C++ for developers, and Blueprint language for game designers. CryEngine goes further and has three (!) languages: C++, Lua, and Flowgraph. It is worth noting that while Unity 3D does support different languages, it doesn't really suggest using more than one for your game, so with Unity you can get away with only, say, C# for your game client.

While having two programming languages in your game client is not fatal, it has some important ramifications. In particular, you need to keep in mind that whenever you have two programming languages, the attacker (for example, bot writer/reverse engineer as discussed in "Different Languages Provide Different Protection from Bot Writers" section below) will usually go through the weakest one. In other words, if you have C++ and JavaScript, it is JavaScript which will be reverse-engineered (that is, if JavaScript allows to manipulate those things which are needed for the bot writer – and usually it does).

A word on CUDA and OpenCL

I wanna show you something. Look, Timon. Go on, look. Look out to the horizon, past the trees, over the grasslands. Everything the light touches… [sharply] belongs to someone else!
— Timon's Mom, Lion King 1 1/2 —

If your game is an inherently 3D one, it normally means that you have a really powerful GPU at your disposal on each and every client. As a result, it can be tempting to try using this GPU as a GPGPU, utilizing all this computing power for your purposes (for example, for physics simulation or for AI).

GPGPU
General-purpose computing on graphics processing units (GPGPU, rarely GPGP or GP²U) is the use of a graphics processing unit (GPU), which typically handles computation only for computer

Unfortunately, on the client side, player's GPU is usually already pushed to its limits (and often beyond), just for rendering. This means that if you try using GPU for other purposes, you're likely to sacrifice FPS, and this is usually a Big No-No in 3D game development. This is pretty much why while in theory CUDA (and/or OpenCL) is a great thing to use on the game client, it is rarely used for games (beyond 3D rendering) in practice. In short – don't hold your breath about available GPU power to use it as a GPGPU; not because this power is small (it is not), but because it is already used.

On the other hand, for certain types of simulations, server-side CUDA/OpenCL in an authoritative server environment might make sense; we'll discuss it in a bit more detail in Chapter [[TODO]].

Different Languages Provide Different Protection from

Bot Writers

Game Bot is a type of weak AI expert system software which for each instance of the program controls a player
— Wikipedia —

graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).
— Wikipedia —

As it was discussed in Chapter III, as soon as your MMO game becomes successful, it becomes a target for cheaters. And two common type of the cheaters are bot writers and closely related bot users. For example, for an MMORPG you can be pretty much sure that there will be people writing bots; these bots will "grind" through your RPG, will collect some goodies you're giving for this "grinding", and will sell these goodies, say, on the eBay. And as soon as there is a financial incentive for cheating, cheaters will be abundant. For other genres, such as MMOFPS or casino multiplayer games, bots are even more popular. And if cheaters are abundant, and cheaters have significant advantage over non-cheating players, your game is at risk (in the ultimate case, your non-cheating players will become so frustrated that your game is abandoned). As a result, you will find yourself in an unpleasant, but necessary role of a policeman, who needs to pursue cheaters so that regular non-cheating users are not in a significant disadvantage.

The problem of bot fighting is extremely common and well-known for MMOs; unfortunately, there is no "once and for all" solution for it. Bot fighting is always a two-way battle with bot writers inventing a way around the MMO defences, and then MMO developers striking back with a new defence against the most recent attack; rinse and repeat.

"Bot fighting is always a two-way battle with bot writers inventing a way around the MMO defences, and then MMO developers striking back with a new defence against the most recent attack; rinse and repeat.

We'll discuss bot fighting in more detail in Chapter [[TODO]], but at the moment, we won't delve into the details of this process; all we need at this point is two observations:

- for bot fighting, every bit of protection counts (this can be seen as a direct consequence of the battle going back-and-forth between bot writers and MMO developers)
- reverse engineering is a cornerstone of bot writing

From these, we can easily deduce that

for the game client, the more resilient the programming language against reverse engineering – the better

Resilience to Reverse Engineering of Different Programming Languages

Now let's take a look at different programming languages, and their resilience to reverse engineering. In this regard, most of practical programming languages can be divided into three broad categories.

Compiled Languages.[2] Whether you like compiled languages or not as a developer, they clearly provide the best protection from reverse engineering.

"from all the popular compiled languages, C++ tends to produce the binary code which is the most difficult-to-reverse-engineer (*that is, provided that you have turned all the optimizations on, disabled debug info, and are not using DLLs)*

And from all the popular compiled languages, C++ tends to produce the binary code which is the most difficult-to-reverse-engineer (*that is, provided that you have turned all the optimizations on, disabled debug info, and are not using DLLs).* If you have ever tried to debug at assembly level your "release" (or "-O3") C++ code, compiled with a modern compiler, you've certainly had a hard time to understand what is going on there, *this is even with you being the author of the source code!* C++ compilers are using tons of optimizations which make machine code less readable; while these optimizations were not intended to obfuscate, in practice they're doing a wonderful job in this regard. Throw in heavy use of allocations typical for C++,[3] and you've produced a binary code which is among the most obfuscated ones out there.

One additional phenomenon which helps C++ code to be rather difficult to reverse engineer, is that even a single-line change in C++ source code can easily lead to vastly different executable; this is especially true when the change is made within an inlined function, or within a template.

Compiled languages other than C++, tend to provide good protection too, though the following observation usually stands. The less development time has been spent on the compiler, the less optimizations there are in generated binary code, and the more readable and more easy-to-reverse-engineer the binary code is.

One last thing to mention with respect to compiled languages, is that while C++ usually provides the best protection from reverse-engineering from programming language side, it doesn't mean that your code won't be cracked. Anything which resides on the client-side, can be cracked, the only question is how long it will take them to do it (and there is a Big Difference between being cracked in two days, and being cracked in two years). Therefore, making all the other precautions against bot writers, mentioned in Chapter [[TODO]], is still necessary even if you're using C++. Moreover, even if you do everything that I've mentioned in this book to defend yourself from bot writers – most likely there still will be bot writers able to make reverse engineering of your client (or at least to simulate user behaviour on top of it); however, with bots it is not the mere fact of their existence, but their numbers which count, so every bit of additional protection *does* make a difference (fo further discussion on it, see Chapter [[TODO]]).

Languages which compile to Byte-Code. Compiling to a byte-code (with the runtime interpreting of this byte-code in some kind of VM) is generally a very useful and neat technique. However, the byte-code tends to be reverse engineered significantly more easily than a compiled binary code. There are many subtle reasons for this; for example, function boundaries tend to be better visible within the byte-code than with compiled languages, and in general byte-code operations tend to have higher-level semantics than "bare" assembler commands, which makes reverse engineering substantially easier. In addition, some of byte-code-executing VMs (notably JVM) need to verify the code, which makes the byte code much more formalized and restricted (which in turn limits options available for obfuscation).

JIT
just-in-time (JIT) compilation, also known as dynamic

It should be noted that JIT compilers don't help to protect from the reverse-engineering; however, so-called Ahead-of-Time Compilers (such as gcj or Excelsior JET), which compile Java to binary instructions, do help against reverse engineering. What really matters here is what you ship with your client – machine binary code or byte-code; if you're shipping machine code – you're better than if you're shipping byte code. This also means that "compile to .exe" techniques (such as "jar2exe") which essentially produce .exe consisting of JVM

and byte-code, do not provide that much protection. Moreover, "byte-code encryption" feature in such .exes is still a Security-by-Obscurity feature[4], and (while being useful to scare some of bot writers) won't withstand an attack by a dedicated attacker (in short: as decryption key needs to be within the .exe, it can be extracted, and as soon as it is extracted, all the protection falls apart).

Still, I would say that with an "encrypted"/"scrambled" byte-code within your client, you do have a fighting chance against bot writers, though IMHO it is going to be an uphill battle.[5]

> **translation, is compilation done during execution of a program – at run time – rather than prior to execution.**
>
> — *Wikipedia* —

Interpreted Languages. From the reverse engineering point of view, interpreted programming languages provide almost-zero protection. The attacker essentially has your source code, and understanding what you've meant, is only a matter of (quite little) time. Obfuscators, while improving protection a little bit against a casual observer, are no match against dedicated attackers. Bummer. As a rule of thumb, if you have interpreted language in your client, you shall assume that whatever interpreted code is there, will be reverse engineered, and modified to the bot writer's taste. Oh, and don't think that "we will sign/encrypt the interpreted code, so we won't need to worry about somebody modifying it" – exactly as with "byte-code encryption", it doesn't really provide more than a scrambling (and to make things worse, this scrambling can be broken in one single point).

On Compilers-with-Unusual-Targets. In recent years, several interesting projects have arisen (such as Emscripten, GWT, JSIL/Santarelle, and FlasCC), which allow to compile C++ into JS or into Flash bytecode (a.k.a. "ABC"="ActionScript Byte Code"). From resilience-to-reverse-engineering point of view, a few things need to be kept in mind with regards of these compilers:

- those compilers which are based on LLVM front-end (and just provide back-end), will generate quite difficult-to-break code even for JS
  - this include at least Emscripten and FlasCC (I have no idea about the others)
- on the other hand, as all the communication with the rest of the system will need to be kept in JS (or in ActionScript), overall protection will suffer significantly compared to "pure" generated code
- if you have encrypted traffic (which itself serves as a quite strong protection from bot writers, see discussion in Chapter [[TODO]]), you will face a dilemma: either to use system-provided TLS (which will weaken your protection greatly), or to try compiling OpenSSL with these compilers (no idea if it will work, and also performance penalties, especially on connecting/reconnecting, can be Really Bad).

---

[2] technically, we're speaking not about languages as such, but about compilers/interpreters. Still, for the sake of keeping things readable, let's use the term "language" for our purposes (with an understanding that there is compiled-to-binary Java, and there is compiled-to-bytecode-Java, etc.)

[3] in practice, it may be a good idea to throw in a randomized allocator, so that memory locations differ from one run to another , more on this in Chapter [[TODO]]

[4] in fact, "scrambling" would be more fair name for such features

[5] I didn't have a chance to test this theory myself, so take it as just my yet another educated guess

Summary. Observations above can be summarized in the following Table V.1 (numbers are subjective and not to scale, just to give an idea some relations between different programming languages):

Programming Language	Resilience to Reverse Engineering (Subjective Guesstimate[6])
C++ (high-level optimization, no debug info, no DLLs[7])	7.5/10
C (high-level optimization, no debug info, no DLLs)	7/10
Java or C# (compiled to binary, no DLLs)	6.5/10
Java or C# (compiled to byte code, obfuscated, and scrambled)	5.5/10
Java or C# or ActionScript (compiled to byte code)	5/10
JavaScript (obfuscated)	2/10
JavaScript	1/10

Note that here I'm not discussing other advantages/disadvantages of these programming languages; the point of this exercise is to emphasize one aspect which is very important for games, but is overlooked way too often. Also note that I'm not saying that you MUST write in C++ no-matter-what; what you should do, however, is to take this table into account when making your choice.

---

[6] while it is supported by anecdotal evidence, gathering reliable statistics is next-to-impossible in this field

[7] as discussed in Chapter [[TODO]], DLLs represent a weak point for reverse engineering

Language Availability for Game Client-Side Platforms

The next very important consideration when choosing programming language, is "whether it will run on *all* the platforms you need". While this requirement is very common not only for games, it still has specifics in the game development world. In particular, list of the client platforms is not that usual.

In the Table V.2 below, I've tried to gather as much information as possible about support of different programming languages for different client game platforms. **[[NOTE TO BETA TESTERS: PLEASE POINT OUT IF YOU SEE SOMETHING WRONG WITH THIS TABLE]]**

	Windows	Mac OS X	PS4[8]	XBox One[8]	iOS[8]	Android	Facebook etc.

C/C++[9]	Native	Native	Native	Native	Native[10]	Native[10]	Emscripten, Chrome Native Client (Chrome Only), FlasCC
Objective C	GNUStep	Native	No	No	Native	No	No
Java	Oracle, can be distributed with the game	Oracle, can be distributed with the game	Not really[11]	Not really[11]	Oracle MAF, Robo VM	Native, Oracle MAF	Oracle, usually requires separate install, or GWT(?)[12]
C#	Native	Mono	Not yet	Native	Xamarin	Xamarin	JSIL(?) or Saltarelle(?)[12]
ActionScript (a.k.a. "Flash")[13]	Adobe AIR SDK	Adobe AIR SDK	No	No	Adobe AIR SDK	Adobe AIR SDK	Adobe, most of the time already installed
HTML 5/JavaScript[14]	Native	Native	Native	Native	Native	Native	Native

[8] not accounting for jail-broken devices
[9] Caution required, see Chapter [[TODO]]
[10] see Chapter [[TODO]]
[11] well, you can write your own JVM and push it there, but…
[12] see "Big Fat Browser Problem" section below
[13] Given developments in the 2H'2015 (see, for example, [TheVerge]), ActionScript's future looks very grim
[14] Compatibility and capabilities are still rather poor

Sprinkle with All The Usual Considerations

We've discussed several peculiarities of the programming languages when it comes to games. In addition to these not-so-usual things to be taken into account, all the usual considerations still apply. In particular, you need to think about the following:

- is your-language-of-choice used long enough to be reasonably mature (so you won't find yourself with fixing compiler bugs – believe me, this is not a task which you're willing to do while developing a game)?
- are available tools/libraries/engines sufficient for your game?
- is it readable? More specifically: "is it easily readable to the common developer out there?" (the latter is necessary so that those developers you will hire later, won't have too much trouble jumping in)
- how comfortable your team feels about it?
- how difficult is to find developers *willing* to write in it? Note that I'm not speaking about "finding somebody with 5 years of experience in the language"; I'm sure from my own 15+ years experience as an architect and a team lead,[15] that any [half-]decent programmer with *any* real-world experience in more than one programming language can start writing in a new one in a few weeks without much problems.[16] It is *frameworks* which usually require more knowledge than *languages*, but chances of finding somebody who is versed in your specific framework are usually small enough to avoid counting on such miracles. On the other hand, if your programming language of choice is COBOL, Perl, FORTRAN, or assembler, you may have difficulties with finding developers willing to use it.

"Any (half-)decent programmer with *any* real-world experience in more than one programming language can start writing in a new one in a few weeks without much problems.

- do you have at least one person on the team with substantial real-world experience in the language, with this person developing a comparable-size projects in it? Right above I was arguing that in general language experience is not really necessary, but this argument applies only when developer comes to a well-established environment. And to build this well-established environment, you need "at least one person" with an intimate knowledge of the language, environments, their peculiarities, and so on.
- is it fast enough for your purposes? Here it should be noted that performance-wise, there are only a very few tasks which are time-critical on the client side. Traditionally, with games time-critical stuff is pretty much restricted to graphics, physics, and AI. With MMO, however, most of physics and AI normally need to be moved to the authoritative server, leaving graphics pretty much the only client-side time critical thing.[17] Therefore, it might (or might not) happen that all of your game logic is not time-critical; if it isn't – you can pretty much forget about performance of your programming language (though you still need to remember not to do crazy things like using O(N^3) algorithms on million-item containers).

Just for the sake of completeness, here is the list of questions which are NOT to be taken into account when choosing your programming language:

- is it "cool"?
- how will it look on my resume after we fail this project?[18]
- is it #1 language in popularity ratings? (while popularity has some impact on those valid questions listed above, popularity as such is still very much irrelevant, and choosing programming language #6 over language #7 just because of the number in ratings is outright ridiculous)
- is the code short? As code is read much more often than it is written, it is "readability" that needs to be taken into account, not "amount of stuff which can be fit into 10 lines of code". Also note that way too often

"how will it look on my

"brevity" is interpreted as "expressiveness" (and no, they're not the same).

resume after we fail this project?

---

[15] BTW, feel free to pass this message on to your hiring manager; while they might not trust you that easily, in certain not-so-bad cases a quote from a book might help
[16] that is, if it is not an exotic one such as LISP, PROLOG, or Haskell
[17] in case of client-side prediction, however, you may need to duplicate some or even most of physics/AI on the client side, see Chapter [[TODO]] for details
[18] if you succeed with the MMO project, the project itself will be much more important for your resume than the language you've used, so the only scenario when you should care about "language looking good on resume" is when you're planning for failure

C++ as a Default Game Programming Language

Given our analysis above, it is not at all surprising that C++ is frequently used for games. Just a few years ago, it was pretty much the only programming language used for serious game development (with some other language usually used at the game designer level). These days, there is a tendency towards introducing other programming languages into game development; in particular, Unity is pushing C#.

However, we should note that while C# may[19] speed up your development, it comes with several significant (albeit non-fatal) caveats. First, as noted above, C# apps (at least when they are shipped as byte code) has lower resilience to bot writers. Second, you need to keep an eye on the platforms supported by C#/Mono. Third, with automated memory management, And last but not least, many of C# implementations out there are known to use so-called "stop-the-world" garbage collection; in short – from time to time the whole runtime needs to be stopped for some milliseconds, causing "micro-freezes". While this is certainly not a problem for games such as chess or farming, it can easily kill your MMOFPS or MMORPG. There are quite a few tricks to mitigate "stop-the-world" issues, so you might be able to get away with it, but honestly, I don't think that it is worth the trouble for FPS-critical games.

Bottom line: C++ is indeed a default programming for games, and for a good reason. While your team might benefit from using alternative languages such as C#, take a look at issues above to make sure that they won't kill your specific game.

On C++ and Cross-Platform Libraries

One common approach in cross-platform C++ development world is to find and use one single cross-platform library to cover all your platforms; with this one-library-for-all-platforms approach, you can have different libraries for different functionality (for example, one for graphics and another for networking), but each of these libraries is very often chosen with an intention to cover the whole spectrum of your target platforms; anything less than that is thrown away as unacceptable. I am arguing (alongside with quite a few developers out there) that such an approach is not necessary, and moreover, is usually detrimental, especially for Games with Undefined Life Span. More precisely, it is not the libraries which are detrimental, it is *dependency* on the library which is detrimental.

First of all, let's show that relying on one single library is not necessary. To avoid relying on one single library, there is one well-known tried-and-tested way: you can (and should) make an isolation layer with *your own* API, which isolates your code from all the 3rd-party libraries. If your own API is indeed about your own needs (and not just a dumb wrapper around 3rd-party library), you will be able, when/if it becomes necessary, to write another isolation layer and to start using a

completely different library on a different platform. One example of such an approach was described in "Logic-to-Graphics Layer" section above. [[TODO: elaborate?]]

Now, to the question *why* it is a Bad Idea to use API of a single library directly. This is because of the same good old vendor lock-in, the very same which has caused us to write cross-platform programs. The thing is that, using *any* API all over your code means that you won't be able to switch from it, hence whenever such using-some-API-all-over-your-code happens, you *are* locked-in. And being locked-in to a cross-platform library is not necessarily any better than being locked in to a single platform; not only nobody knows whether the library will be alive and kicking in the long run, but also nobody knows whether they are *the best* for every target platform, and whether they will support that new platform which everybody will be using in 5 years from now, soon enough after it appears.

I certainly don't mean that cross-platform libraries are in any way "*evil*"; what I mean is that you should (whenever possible) to keep your own isolation layer (which is more than just a *"dumb wrapper",* and provides you with an API tailored to *your* needs), to avoid vendor lock-in on a cross-platform library. Behind this isolation layer – feel free to use anything which you want, cross-platform or platform-specific. This approach is good for many reasons; in particular, it allows to resolve a dilemma "whether to use one single cross-platform library which is imperfect, or to use different libraries which are better but time consuming"; with this isolation layer in place, you can start with a single cross-platform library (hiding behind your isolation layer), and to rewrite isolation layer (not touching anything else) for those platforms which are of particular importance for you.

---

[19] and usually will, though many of C++ negatives can be avoided if you're careful enough, see Chapter [[TODO]] for details

Big Fat Browser Problem

As we can see from the Table V.2 above, if you need to have your game *both* for Facebook (read: "browser"), *and* for some other platform, you'll have quite a problem at your hands. As of now, I don't see any "fit-for-all" solution, so let's just describe more-or-less viable options available in this case.

Option 1. Drop Facebook as a platform. While very tempting technically ("hey, we can stay with C++/C#/… then!"), business-wise it might be unacceptable. Bummer.

"Option 1. Drop Facebook as a platform. While very tempting technically, business-wise it might be unacceptable.

Option 1a. Drop Everything-Except-Facebook as a platform. Also very tempting technically, and also likely to be unacceptable business-wise.

Option 2. Use Adobe AIR SDK with or without [Starling]/Citrus. One Big Obstacle on the way of this (otherwise very decent) option is that whole future of the ActionScript currently looks very grim; with even Adobe pushing its own users towards HTML5+JS [TheVerge], chances of ActionScript being developed further in 5 years from now, look negligible. Another problem with using ActionScript (as if the first one is not enough) is that resilience of your code to hacking will be not-so-good (see Table V.1 above for "ActionScript"); while not fatal, this is one thing to remember about.

Option 3. Other-Language plus ActionScript (2 code bases). This will

require to keep two separate code bases for "Other-Language" and ActionScript. And clients with two code bases are known to fail pretty badly. You may still try it, but don't tell that I didn't warn you. Also, keep in mind that as with Option 2 above, resilience of your code to reverse engineering will be that of ActionScript (according to "the weakest link" security principle).

Option 3a. Other-Language plus Line-by-Line manual translation to ActionScript (1.5 code bases). Details of line-by-line conversion will be described in [[TODO]] section below. For now, let's take it as granted that such a thing *might* work, and results in "1.5 code bases" to be maintained. Maintaining of these 1.5 code bases tends to be much easier than maintaining 2 code bases, and it *might* work for you. This option will represent a significant headache maintenance-wise, but at least it won't hurt performance on mobiles, which might make it viable, especially if mobile is more important for your game business-wise than Facebook.

Option 4. so-called "HTML5" (actually, JS). This is the option which I'd try to avoid even for a game as simple as AngryBirds (and anything beyond it would only make things worse). Despite all the improvements in this field, JS is still one big can of worms with lots of programming problems trying to get out of the can right in the face of your unfortunate player. While low-weight games along this way may be viable (see, for example, [Bergström]), as the complexity of your game grows, problems will mount exponentially. While HTML5 *might* become a viable technology for larger games at some point, right now it is not there, by far. And even when it does – you'll need to keep in mind that protection of JS from being hacked tends to be very low (see Table V.1 above).

"Despite all the improvements in this field, JS is still one big can of worms with lots of programming problems trying to get out of the can right in the face of your unfortunate player.

Option 5. Other-Language with a "Client-on-Server" trick and Flash front-end (1.5 code bases). Details of this approach will be discussed in [[TODO]] section below. Disadvantages of this approach are mostly related to scalability (and these issues MUST NOT be taken lightly, as described below); however, on the plus side – you can stay with single-code-based Other-Language for your game logic, and you can keep your Other-Language reasonably protected from bot writers (that is, if you are not too concerned about bots coming from Flash clients, which may happen if player capabilities for Facebook and for non-Facebook versions are different, so that Facebook version is actually just a "teaser" for the main one).

Option 5a. Other-Language with a "Client-on-Server" trick and HTML5/JS front-end. A variation of Option 4, replacing Flash front-end with HTML5/JS one. *Might* work even for larger games, but no warranties of any kind (and the issue with JS being easily hackable, is still present). For further discussion, see [[TODO]] section below.

Option 6. Compile-to-JS: Emscripten, Java with GWT, or C# with JSIL/Santarelle. It seems to be possible to compile game logic from C++ to JS using Emscripten, (or from Java to JS using GWT, or from C# to JS using JSIL or Santarelle), and then to have Logic-to-Graphics Layer, as well as graphics engine, in JS/HTML5. Performance-wise, LLVM-based Emscripten claims performance which is merely 3-to-10x worse compared to native C++,[GDC2013] which is not too bad for at least 95% of the client-side code. On the other hand, I have no experience with these technologies, and have no idea whether they work in practice (even less idea if they work for games); if you have any experience about this route (either positive or negative) – please let me know. IMHO, this option is one of the most promising ones in the long run, but I am not sure if it is production-ready yet.

Option 7. Chrome Native Client. This thing will work only for Chrome browsers, but given the growing market share of Chrome[20], you might be able to get away business-wise with supporting only Chrome for Facebook-oriented games. If it is the case, and if your primary language of choice

is C/C++, you can try to run C++ game logic within Chrome's "sandboxed" [GoogleNativeClient]. I have no idea if you succeed on this way, but IMHO it looks quite promising (that is, if Google will keep supporting it, which in turn depends on the number of developers using it).

Option 8. FlasCC/Crossbridge. As LLVM guys were able to compile C++ into JS, I am not surprised that they were also able to compile it into ABC (ActionScript Byte Code). Originally, FlasCC was an Adobe project, and then they released it as an open-source project known as Crossbridge. Unfortunately, as of the end of 2015, neither FlasCC nor Crossbridge seem to be actively maintained. A pity.

Which of the options above suits your game better – is your decision, and it *heavily* depends on specifics of your game. A few hints though (no warranties of any kind, batteries not included): if Facebook is your primary platform – take a look at Option 3a (most reliable, but with lots of extra maintenance and with only limited protection from bot writers), and Option 6 with Emscripten (the most promising in the long run, but probably a bit too immature now); if other platforms are of more interest than Facebook – take a look at Option 3a, Option 5/5a (ugly, but might work for you), Option 6, and Option 7 (the easiest one and the best protection, but Chrome-only).

---

[20] As of the end of 2015, Chrome market share is about 50% and is still growing [UsageShareOfWebBrowsers]

[[To Be Continued…

This concludes beta Chapter V(b) 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 V(c), "Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines"]]

[–] References

[Bergström] Sven Bergström, "Real Time Multiplayer in HTML5"
[Starling] "Starling, The Cross Platform Game Engine"
[GoogleNativeClient] Wikipedia, "Google Native Client"
[GDC2013] "Fast C++ on the Web using Emscripten and asm.js"
[TheVerge] Jacob Kastrenakes, "Adobe is telling people to stop using Flash"
[UsageShareOfWebBrowsers] Wikipedia, "Usage share of web browsers"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter V(a). Modular Architecture: Client-Side. Graphics from "D&D of M*...

Chapter V(c). Modular Architecture: Client-Side. On Debugging Distribute... »

Filed Under: Distributed Systems, Programming, System Architecture
Tagged With: client, game, multi-player