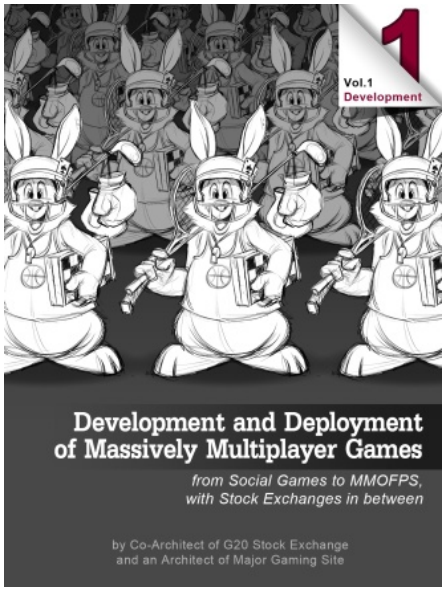
Chapter I: "Business Requirements" from upcoming book "Development and Deployment of MMOG"

posted October 26, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter I 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.*

Please note that this Chapter I may look boring for some of the developers; don't worry, there will be a lot of code-related stuff starting from Part B, but at the moment we need to describe what we're dealing with.



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

Preface

So, you have got a Great Idea for your Next Big Thing massively multi-player game, and know every tiny detail about gameplay and graphics which you want your game to have. Now the only tiny thing you need to do is to program it. Unfortunately for you (and fortunately for me as an architect and the author of this book 😉) game development and subsequent deployment is not that simple. There are lots of details you need to take into account to have your game released, to be able to cope with millions of simultaneous players having very different last-mile connections, and to make the game work with 0.01% of unplanned downtime while being able to add new game features twice a month.

Part A. Conception: Before the Very Beginning

You don't "make" a violin. It is barrels and benches which are "made". And violins, just like bread, grapes, and children – are born and raised.
— Nicola Amati character from "Visit to Minotaur" movie —

"A game being developed is pretty much like your baby.

A game being developed is pretty much like your baby. It will go through all the stages which are typical for baby development, from conception to a newborn and then to a toddler. While development of your game certainly doesn't stop at that point, in this book we won't discuss how to raise your game beyond toddler; child and teen issues (both with games and with real children) are too often of psychological nature, and are beyond mostly-physical issues which we're about to discuss.

"You" as used throughout this book, actually means "parents of your game baby"; it will usually be a small team, but can be a 100-developer team on one side of the spectrum, or a single developer on another one. What is really important is not the size of the team, but how the team feels about the project.

If you (as a future parent) don't feel that your future game is *your baby* – think twice before conceiving it. Doing such a challenging development with only money in mind might not be the best decision in your life. If you're starting to develop only for money without any feelings for the project – then there are two possible outcomes. In the first case you will gradually become attached to the project, and eventually will have certain positive feelings about its development, greatly increasing the chances for success. In the second case, you keep doing it for money; while making a great game is still possible this way, it is much much more difficult to achieve. Success if doing-it-for-money-only becomes even more elusive if this is your first massively multiplayer game project, and you need to keep this in mind.

In Part A, we will discuss activities which need to be performed even before the coding can be started; let's name this stage project conception. It includes many things which need to be done, from formulating business requirements to setting up your source control and issue tracking systems, with lots of critical decisions in between.

Chapter I. Understanding Business Requirements

As mentioned above, we're working under assumption that you've got a Great Game Idea (with as full understanding of planned user experience as it is possible at this time), you're really passionate about it, and are really eager to start development.

What should be your very first step on this way? Start coding? Nope. Choose the programming language? By the tiniest of the margins closer, but still no. The very first step should be to understand what exactly you're going to achieve.

> "The very first step should be to understand what exactly you're going to achieve.

With any game, there are quite a few things which are dictated by your future players (and other project stakeholders). Even if your project is entirely non-commercial, just for the sake of being consistent with the rest of the world, let's name these things "Business Requirements".

Project Stakeholders

Every project has project stakeholders. A stakeholder can be an investor, a manager, and/or a customer. For games, it is often translated into game designers, producers, marketing/monetizing guys, customer service representatives (CSRs), and, of course, players. If you're developing the game in your spare time, it can even be yourself. In any case, every project out there has project stakeholders. For games, one extremely important type of the stakeholder is the future player (usually as a "focus group").

One thing which is paramount for the game to be successful, is to

Have Project Stakeholders, including Future Players, Take Part in Development Process

If your project stakeholders don't participate in your development process (this should apply to all the development stages, from specifying requirements to alpha/beta testing) – the project is doomed almost for sure. And for games, project stakeholders MUST include future players of your game.

On the other hand, having only future players as project stakeholders is not enough. For your game to survive, most likely you will need some kind of monetization. And those people who're responsible for monetization (marketing etc.) are also very important project stakeholders, and MUST be involved in game development.

> "For your game to survive, most likely you will need some kind of monetization.

The reason behind can be roughly described as follows. Each game tends to create a separate world, with its own rules, which are not obvious to the outsiders (and developers are outsiders for the game world despite intimate involvement with game mechanics). While we as developers can try to guess what is the best from the stakeholder's point of view – these guesses are usuallyway off, that makes the game unplayable (if players' opinions are not asked for), or non-monetizable (if other stakeholders are not asked). For the project to be successful, we DO need to have a stakeholder available during all the stages of the game development process. In other words, if we (as developers) have any doubts on any issue related to business requirements – we SHOULD have somebody on hand to ask for their authoritative opinion.

BTW, don't think that if you're going to play the game, your opinion as a future player's will be sufficient. Unfortunately, when we (as developers) are writing code, it affects our judgments about the game a lot; in other words, we know too much about the game internals (and on efforts we need to spend to develop this or that particular feature) to represent opinion of "an average player out there". While our suggestions (based on this knowledge) can be very valuable, all the decisions about gameplay SHOULD be made by those future players who are not developers.

To summarize:

Participation of both future-players and other stakeholders (such as people responsible for monetization) in developing (and later in amending) Business Requirements is absolutely necessary.

No stakeholders – no Business Requirements – no development, it is that simple. Doing it any other way is a foolproof way right into disaster.

Stakeholders and Business Requirements

In some cases your stakeholders will give you a specification which says what you need to do. More often than not, however, you will just get a vague description of the Great Game Idea. It's fine as a first step, but to get a clearer understanding of what is needed, you have to get your project stakeholders to sit down together with you and to write down your real Business Requirements.

This will involve at least one session dedicated just to this purpose (and probably much more than just one such session); ideally, these sessions should be in person rather than some kind of a conference (video)-call. I do know that in the XXI century there are ultra-cheap conference calls and video conferences available, but they still fall short compared to in-person meetings. While most of ongoing communication can be made over the phone/Skype/chat/email/..., at a few important points during game development process, such IRL meetings are necessary, and one of these points is certainly those Business-Requirements-writing sessions.

Much more important, however, is to make sure that

Business Requirements are written by Project Stakeholders (and not by Developers)

During the session, by all means, note down all the things-you-think-are-stupid and raise concerns (preferably in a bit more polite form than "are you guys crazy or what?"), but be ready to accept decisions by stakeholders when they insist (as long as they're staying away from implementation details, see below).

During these business-requirement-writing sessions, our role as developers is generally not to suggest business requirements, but to make sure that all our questions to project stakeholders are answered. Also it is very important to remember that Business Requirements is *not* about "*how* we will do it", and to concentrate on "*what* is the thing which we will do". While it is perfectly ok to say "implementing this feature will take us extra 3 months" (which in turn does need you to understand – but not explain – how to do it), a decision "if having this feature worth these extra 3 months", lies entirely in stakeholder's domain.

"During these business-requirement-writing sessions, our role as developers is

It should be also understood that (exactly because the session is about *what?* and not about *how?*) it is entirely possible that at later stages (but still before the coding is started) it may happen

that Business Requirements cannot be satisfied. It is even more important to emphasize that this is a part of normal iterative development process, and in this case another Business Requirement session may be necessary (though the second and subsequent sessions usually simple enough and don't normally need to be in-person; that is, if you're not too unlucky and didn't skip too much of this book 😉).

generally not to suggest business requirements, but to make sure that all our questions to project stakeholders are answered.

Requirements vs Implementation Details

What are these "Business Requirements" for a typical game? Basically, they include everything your players will be able to observe. However, we need to distinguish between the things that the player cares about, and their respective implementation details.

For example, players do care about the platforms where they will be able to run your game, so "which platforms are to be supported?" is certainly one of your business requirements, but on the other hand players don't care about the programming language you will be using (as long as it can run on all those platforms). In another example players do care about response times and may care about how-your-app-works-over-firewalls, but they don't care if you achieve those response times and working over firewalls via TCP or via UDP, as long as the whole thing does work.

It can be summarized as follows:

Business Requirements SHOULD be expressed exclusively in Player's Terms

Or the other way around, using terms which are not familiar to the majority of players (or monetization people) SHOULD be prohibited for your business requirements document.

"If you write down a Bad business requirement "We MUST write our app in Java"

Why is this so important? Because writing requirements down in implementation terms rather than in player terms may severely hurt your ability to choose an optimal way to implement your game. Just as an example, if you write down a Bad business requirement "We MUST write our app in Java" (instead of the Good one "Our app MUST run on Windows, iPhone, and Android"), you won't even start to think about writing your app in C++ and porting it to Android using NDK (with a rather minimal Java UI, as described in Chapter [[TODO]]).

In another example, if you write a Bad business requirement "We MUST use UDP" (instead of Good one "In 99.99% of cases, we need at most 3sec delay between the user pressing a button and it showing up to the other users"), you won't even start to learn

(instead of the Good one "Our app MUST run on Windows, iPhone, and Android"), you won't even start to think about writing your app in C++ and porting it to Android via NDK.

about the ways to improve TCP interactivity (described in Chapter [[TODO]]), and may miss on an opportunity to make your app more firewall-friendly and to simplify your development by using TCP. Or the other way around, you may write a Bad business requirement "We MUST use TCP" (instead of a Good one "We MUST have TLS-class security"), and may miss on an opportunity to make your app more responsive via implementing it over UDP (using DTLS and/or TLS-over-reliable-UDP for security purposes, as described in Chapter [[TODO]]).

In short:

Writing Business Requirements in Player Terms allows you to Keep your Options Open

and keeping your options open is a Good Thing in general.

This separation between business requirements and implementation details means that if your project stakeholder (future player, marketing guy, manager, investor, etc.) says "we have a business requirement to write it in Java" (or "to use TCP" etc.) – you need to explain that this is an implementation detail, and to ask for a definition in terms which are obvious to the player.

Moreover, if the stakeholder is a manager and after all the explanations he is still insisting that using UDP is a business requirement – you really need to think if you want to work on this project, as such a deep misunderstanding is often a symptom of super-micro-management and upcoming deep conflicts with this specific manager.[1]

---

[1] While "we need to use UDP" (or TCP for that matter) may be a valid business requirement in some cases (for example, when you're writing a communication library, and your user is a programmer, so she knows about UDP), it doesn't apply to games. You MAY need to use UDP for your game – it is just not a business requirement, but a technical decision on "how to implement these business requirements"

Subject to Change, Seven Days a Week

It is to be understood that in the real world Business Requirements tend to change very often, and are certainly not carved in stone. This is to be expected for most software projects out there, and applies to game development in spades. Therefore:

Expect Business Requirements to Change and Leave

Lots of Room for These Changes

Even if you're told that a certain thing will "never ever" change, keep in mind that "never ever" can come up much earlier than you expect. This is not to tell that you should write an "absolutely universal" system able to deal with *any* change (see about dangers of being over-generic below); this is to tell not to be too upset when you're forced to rewrite 50% of the system when a thing-that-you-were-told-will-never-change does change overnight. Oh, and do keep records of these assurances, so when the requirement changes, you can explain why such a simple thing (from the point of view of stakeholder) requires rewriting half the system.

One important thing to understand is that business requirement being agile doesn't imply that you don't need to write them down. While each of requirements may change later, at every point it should be very clear (and agreed by both stakeholders and developers) what you're trying to achieve *right now.* When (not "if"!) business requirements change – fine, you will update them.

Treat business requirements as one of the documents under your source control system (whether you really put business requirements document under source control – is up to you, but IMHO it is a good thing to do). In any case, business requirements tend to have effects similar to those of an extremely high-level header file in C/C++: as with changing a high-level header file, changing business requirements can be very expensive, but in a majority of cases it doesn't mean rewriting everything out there – *especially if you have prepared for it (see Chapter [[TODO]] for discussion on how to do it).*

The Over-Generic Fallacy

Sculpting is Easy. You just chip away the stone that doesn't look like David.

— (Mis)attributed to Michelangelo —

When speaking about agility and taking "be ready to changing requirements" adage to the extreme, there is often a temptation to write a system-which-is-able-to-handle-everything and which therefore will never change (and handling "everything" will be achieved by some kind of configuration/script/...). While as a developer, I perfectly understand the inclination to "writing Good Code once so we won't need to change it later", unfortunately, it doesn't work this way. The issues with this over-generic approach start with the time it takes to implement, but the real problems start later, when your over-generic framework is ready. When your over-generic code is finally completed, it turns out that either that (a) "everything" as it was implemented by this system, is too narrow for practical purposes (i.e. it cannot be really used, and often needs to be started from scratch), or

"There is often a temptation to write a system-which-is-able-to-handle-everything and which

that (b) the configuration file/script are at best barely usable (insufficient, overcomplicated, cumbersome, etc.). In the extreme case of an over-generic software, its configuration file/script is a fully fledged programming language in itself, so after doing all that work on the over-generic system we need to learn how to program it, and then to program our game, so we're essentially back to the square one.[2]

therefore will never change

In fact, systems-which-can-handle-everything already exist; any Turing-complete programming language can indeed handle absolutely everything; in a sense, Turing-complete programming language, [3] represents an absolute freedom. However, as writing a Turing-complete programming language is normally not in the game development scope, our role as game developers should be somewhat different from just copying compiler executable from one place to another and saying that our job is done.

What we as developers are essentially doing, is restricting the absolute freedom provided by our original Turing-complete language (just like a sculptor restricts the absolute freedom provided to him by the original slab of stone), and saying that "our system will be able to do this, at the cost of not being able to do that". Just as the art of sculpting is all about knowing when to stop chipping away the stone, the art of the software design is all about feeling when to stop taking away the freedoms inherent to programming languages.

Coming back to the Earth from the philosophical clouds:

When developing a game, it is important to strike the Right Balance between being over-generic and being over-specific

---

[2] Creating domain-specific programming language optimized for a game, may make perfect sense; the point here is not aimed against developing scripting languages where they make sense and provide additional value specific to the game domain, but against being over-generic just for the sake of writing-it-once-and-forgetting-about-it

[3] and I don't know of any practical programming language which is not Turing-complete,

Keeping Quality under Time-To-Market Pressure: Priorities, MVP, and Planning

When developing a game (or any other software for that matter), it is very important to deliver it while it still makes sense market-wise. If you take too long to develop, the whole subject can disappear or at lease become much less popular, or your graphics can become outdated.[4] For example, if you started developing a game about dinosauri during dinosaur craze of 1990's but finished it only by 2015, chances are that

your target audience has shrank significantly (not to mention that they've changed a lot).

"We will inevitably be pushed to deliver our game ASAP (with a common target date being 'yesterday'), there is no way around it.

That's why (unfortunately to us developers) we will inevitably be pushed to deliver our game ASAP (with a common target date being "yesterday"), there is no way around it. If leaving this without proper attention, it will inevitably lead to a horrible rush at the end, dropping essential features (while already a lot of time was spent on non-essential ones), skipping most of testing, and to a low-quality game in the end.

Dealing with this time-to-market problem is not easy, but is possible. To avoid the rush in the end, there are two things which need to be done.

The first such thing is defining a so-called Minimum Viable Product (a.k.a. MVP). You need to define what exactly you need to be in your first release. The common way to do it is to do about the same thing which you're doing when packing for a camping trip. Start with things-which-you-may-want-to-have, which will make your first list. Then, go through it and throw away everything except things which are absolutely necessary. Note that you may face resistance from stakeholders in this regard; in this case be firm: setting priorities is vital for the health of the project.

The second endeavour you need to undertake to avoid that rush-which-destroys-everything, is as much obvious as it is universally hated by developers. It is planning. You do need to have schedule (with an appropriate time reserves), and milestones, and more or less keep to the schedule. A bit more on planning will be discussed in Chapter [[TODO]].

---

[4] this is not to mention that you can simply run out of money for the project

Limited-Life-Span vs Undefined-Life-Span Games

One of the requirements for your upcoming game is extremely important, but is not too-well known, so I'll try to explain it a bit. It is all about projected lifespan of your game. As we will see further down the road, game lifespan has profound implications on the game architecture and design.

Starting from the times of the Ancient Gamers (circa 1980), most games out there were intended to be sold. It has naturally limited their life time (for one simple reason: to get more money, the producer needed to release another game and charge for it). This is a classical (not to say it is necessarily outdated) game business model, and massively multiplayer games which are intended to have a limited life span, share

quite a bit with traditional game development. In particular, limited-life-span games are normally built around one graphics engine. Moreover, very often such an engine is very tightly coupled with the rest of the game.

As games were developing from Ancient Gamer Times towards more modern business approaches, game producers have come up with a brilliant idea of writing a game once and exploiting it pretty much forever. Therefore, these days quite a few multi-player games are intended to have a potentially-unlimited life-span. The idea behind is along the following lines:

"Let's try to make a game and get as much as we can out of it, keeping it while it is profitable and developing it along the road"

Indeed, games such as stock markets, poker sites, or MMORPGs such as World of Warcraft are not designed to disappear after a predefined time frame. Most of them are intended to exist for a while (providing jobs to developers and generating profits to owners), and this fact makes a very significant difference for some of the architectural choices.

"Indeed, games such as stock markets, poker sites, or MMORPGs such as World of Warcraft are not designed to disappear after a predefined time frame.

Most importantly, for unlimited-life-span games, there is a risk with relying on one single graphics engine. If your engine is not 100% your own, a question arises: "Are you 100% sure that the engine will be around and satisfy your crowd in 5-10 years time?" This, in turn, has several extremely important implications, shifting the balance towards DYI (see Chapter [[TODO]]) and/or going for ability to switch the engines, severely reducing coupling with the engine (this will be discussed in Chapter [[TODO]]).

Your Requirements List

So, after reading[5] all the stuff above about "how to write your business requirements", you've finished your business-requirements session, and got your list of business requirements for your game. While your list is unique for your game, there are some things that need to be present there for sure:

- A very detailed description of the user experience (including game logic, UI, graphics, sounds, etc.). This is what is often referred to as a "Game Design" document; it is going to take most of your Business Requirements document, but it is game-specific so we cannot really elaborate on it here. However, there are lots of much-less-obvious (and MMO-specific) things which need to be written down, see below.
- Game projected life span (is it "Release, then 3 DLCs over 2 years, and that's it",

or “running forever and ever – until the death will us part”?). For further discussion, see [[TODO]] section above

- Do you need some kind of “invite your Facebook friend” feature (or anything similar)?
- Is your game supposed to be 3D or 2D? Note that at least in theory, dual 2D/3D interface can be implemented, especially for those games with “forever” lifespan
- List of platforms you would like to support for the client-side app
  - List of platforms you want to support in the very first release
  - Note that the list of platforms for the server-side is normally an implementation detail, and as such doesn’t belong to the business requirements. Neither do programming languages, frameworks etc.
- Timing requirements (how long it should take for the player to see what is going on). These are very important for our network programming, so you need to insist on specifying them. “As fast as possible” is not really useful, but “at least as fast as such and such game” is much better (if you can get “at most X milliseconds delay between one user presses a button and another one sees the result”, it’s even better, but don’t count on getting it).
  - closely related to timing requirements is a question about your game being “synchronous” or “asynchronous”. In other words, do you players need to be simultaneously online when they’re playing?[6] At least most of the time, fast-paced games will be “synchronous” (it doesn’t make much sense to play MMOFPS via e-mailing “I’m shooting at you, what will be your response?”), while really slow-paced ones (think chess by snail mail) will be “asynchronous”.
- What types of connection do you need to support? Do you need to support dial-up (hopefully not)? But what about connection-over-3G? What about working over GPRS?
- What is your target geographical area? While “worldwide” always sounds as a good idea, for some very-fast-paced games it might be not an option (this will be discussed in Chapter [[TODO]]). Also considerations “when most of the players are available” can affect some types of gameplay too (for example, if in your game one player can challenge another one, with a loser losing by default, you most likely will need to have “time windows” where such challenges are allowed, with timing of these “time windows” tied to real-world clock in the relevant time zone).
- Are you planning to have Big Finals shown in real-time to thousands and hundreds-of-thousands of observers? *NB: we’ll see why it is important techically, in Chapter [[TODO]]*
- Do you need to implement i18n in the very first release or it can be postponed?
- Client update requirements. There is a part of it that is

i18n

Internationalizati (frequently abbreviated as i18n) is the process of designing a software application so that it can potentially be adapted to various languages and regions without engineering changes.

— Wikipedia —

(almost) universal for all the games: "we do need a way to update the client automatically, simply when the player starts the app" – still, make sure to write it down. However, there are two more subtle questions:

- is it acceptable to stop the game world at all while the clients are being updated? How long this stop-the-world is allowed to take?
- Is it acceptable to force-update client apps (or at least not to allow playing with an out-of-date client)?
  - If not – for how long (in terms of "months back" or "versions back") do you need to support backward compatibility?

- Server update requirements. Most of the server-side stuff qualifies as "implementation details"; however, whenever the server is stopped, it's certainly visible to the players, so "how often we need to stop the server for software upgrades" is a perfectly valid business-level question. Is it acceptable to stop the game while server being updated? How often are server updates planned? With the game being multi-player, stopping and then resuming the game world may become Quite a Pain in the Neck for players. However, allowing for server updates without stopping the game world can easily become a Much Bigger Pain when developing your system (see [[TODO]] chapter for some hints in this direction), so you need to think in advance whether the effort is worth the trouble. Unless a non-stopping server requirement is Really Significant business-wise – you may want to try dropping it from the requirements list, and explicitly say that you can stop the server once-per-N-weeks (and also whenever an emergency server update is required) to update server-side software (where N depends on the specifics of your game).
- In-game payment systems which *may* need to be supported in the long run (these have implications on security, not to mention that you need to have a place for them within your architectured system). Even if it is "the game will be free forever and ever", or "all the payments will be done via Apple AppStore" – it needs to be written down. Oh, and if it is "all the payments will be done via Apple AppStore" *and* there is a "Windows" in the list of the platforms to be supported – there is a likely inconsistency in your requirements, so either drop "Windows", or think about specific AppStores for the Windows platform, or be ready to support payments yourself (which is doable, but is a Really Big Pain in the Neck, so it's better to know about it well in advance).

"Even if it is "the game will be free forever and ever", or "all the payments will be done via Apple AppStore" - it needs to be written down.

As we will see later, we will need all these things to make decisions about network architecture. It means that if your list is missing any of these – you need to go back ~~to the drawing board~~ to the meeting room and get them out of the project stakeholders.

---

[5] And hopefully agreeing with, as blindly following the advice won't get you far enough

[6] I don't want to go into lengthy splitting-hair discussion whether this property should be named "temporal" or "synchronous"; let's simply use the name "synchronous" for the purposes of this book

[[To Be Continued…

This concludes beta Chapter I 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 II, "Games Entities and Interactions"

EDIT: Chapter II, "Game Entities and Interactions", has been published]]

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Contents of "Development and Deployment of Massively Multip..*

Chapter II: "Game Entities and Interactions" from upcoming boo.. »

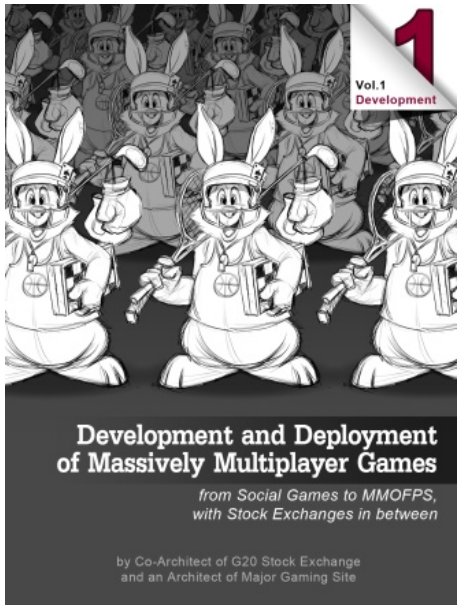
Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: business requirements, game, multi-player

Copyright © 2014-2015 ITHare.com

Chapter II: "Game Entities and Interactions" from upcoming book "Development and Deployment of MMOG"

posted November 2, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

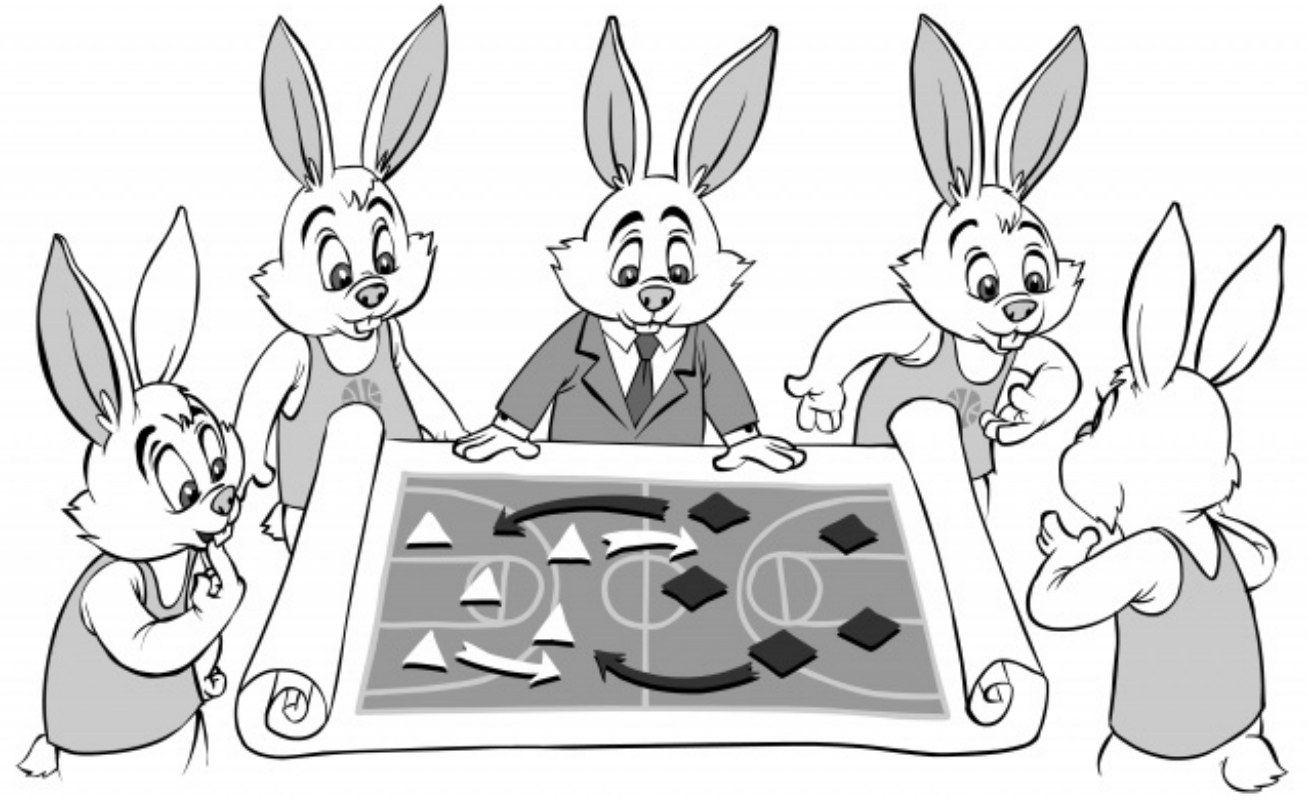
[[*This is Chapter II from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further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



Please note that this Chapter II may look boring for some of the developers; don't worry, there will be a lot of code-related stuff starting from Part B, but at the moment we need to describe what we're dealing with.

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

So, after universally-hated unacceptably-long 2-hour meeting (the one where you discussed your business requirements) you've got your very first idea about what you're going to do. With luck (or if you've read previous Chapter on what you'll need from the network perspective) it contains answers to all the questions we need. Of course, these answers are subject to change, but at least you know what you will be dealing with at the moment. Now, you may think that you know enough to start architecting the game; however, it is not the case (yet). The next step after specifying business requirements is not about drawing an architecture diagram, or (Stroustrup forbid!) a class diagram. This next step should be a thorough understanding of game entities and their interactions.



On Importance of Holding Your Horses

After you've got your Business Requirements, it is often tempting to say "hey, we will be using such-and-such game engine, so all we need is to implement our game around this engine". Or (especially if you're coming from web development) to say the same thing, but building the game around the database. Or to build your game around some protocol (TCP or UDP). However, at this stage you still don't really have sufficient information to make architectural decisions. All these engines, databases, and protocols are nothing more than implementation details, and we're not at the implementing stage yet.

"While your game is likely to have a 3D engine, and very likely to have some DB to provide

While your game is likely to have a 3D engine, and very likely to have some DB to provide persistence, and will certainly need to run on top of some IP-based protocol, it is too early to make any of them a center of your game universe. In particular, even decision whether your game should be game-engine-centric, or 3D-engine-centric, or DB-centric, or protocol-centric, requires more thorough understanding of the game mechanics, that usually arises right after reading Business Requirements.

Making these decisions (and actually any architectural decisions for that matter) before you have Entities&Relations diagram described in this Chapter, can severely restrict your choices, and if you have made a mistake in making such a

persistence, and will certainly need to run on top of some IP-based protocol, it is too early to make any of them a center of your game universe.

decision (and when you're deciding without having sufficient information, mistakes are more than likely), it may easily lead to grossly inefficient and even completely unworkable implementations.

For example, if you decide that "our system should be DB-centric, with 100% of the state being written to DB at all times", and your system happens to be a blackjack site, your implementation will cause about 10x more DB load than an alternative one, plus you will get a bunch of issues with implementing rollback in case if your site has crashed (which causes many games to be interrupted in the middle, and with a multiplayer site you do need some kind of rollback). Usually, the most optimal implementation for many of casino multi-player games is with state of the *table* being stored in-memory only (and synchronized with DB only when one single game is completed), but this won't become obvious until you draw your Entities&Relations diagram.

In an another example, if you decide that "our system should be game-engine-centric", and your game engine of choice doesn't support a concept of *zones* (and doesn't have another way to calculate set of players-who-need-to-know-what-this-player-does, assuming that everybody-interacts-with-everybody instead), you may end up with a system which works reasonably well for small virtual worlds, but which is completely unscalable to larger ones due to O(N^2) traffic which pretty much inevitably arises from everybody-interacts-with-everybody assumption.

TL;DR:

- **DON'T Start with Architecting Around Game Engine**
- **DON'T Start with Architecting Around DB**
- **DON'T Start with Architecting Around Protocol**
- **Overall, it is still too early to start architecting your game.**

Instead,

What you need before architecturing is a diagram showing all the game entities, and all their interactions

Whether this step needs to involve project stakeholders, depends on the nature of the game and on the level of details in your business requirements. However, in any case it is advisable to have project stakeholders available during this stage, as

questions on interactions such as “is entity A allowed to interact with entity B?” are very likely to arise.

Game Entities: What are You Dealing With?

In each and every game, you have some game entities, which you’ll be dealing with. For example, in a MMORPG you’re likely to have PCs, NPCs, zones, and cells; in a casino game you have lobbies, tables, and players; in a social farming game you have players and player farms. Of course, every game will contain many more entities than I’ve mentioned above, but they depend on specifics of your game, so you’re certainly in much better position than myself to write them down. And if you feel that you’re about to be hit by “not seeing forest for the trees” syndrome, you can always replace your diagram with several ones (organized in a hierarchical manner), so that each one contains only a manageable number of entities.

Interactions Between Game Entities

Those game entities from the previous chapter, usually need to interact with each other. Players reside within cells which in turn reside within zones, PCs interact with NPCs, players sit and play on casino tables, and players interact with other player’s farms. All these interactions are very important for the game architecture, and need to be written down as a part of your Entities diagram. Even more importantly, you need to be reasonably sure that you have listed *all* the interactions which you can think of at the moment.

“Even more importantly, you need to be reasonably sure that you have listed *all* the interactions which you can think of at the moment.

What Should You Get? Entities&Interactions Diagram

As a result of the process of identifying your game entities, you should get a diagram (let’s name it “Entities&Interactions Diagram”) showing all the major game entities and, even more importantly, *all possible interactions* between these entities.

One thing which MUST be included into the Entities&Interactions Diagram (alongside with gameplay-related entities), is entities related to monetization (payments, promotions) *and* entities related to social interactions. In other words, if you’re going to rely on viral marketing via social networks, better know about it in advance; as discussed below, the impact of social interactions on architecture can be much more significant and devastating than simple “we’ll add that Facebook gateway later”.

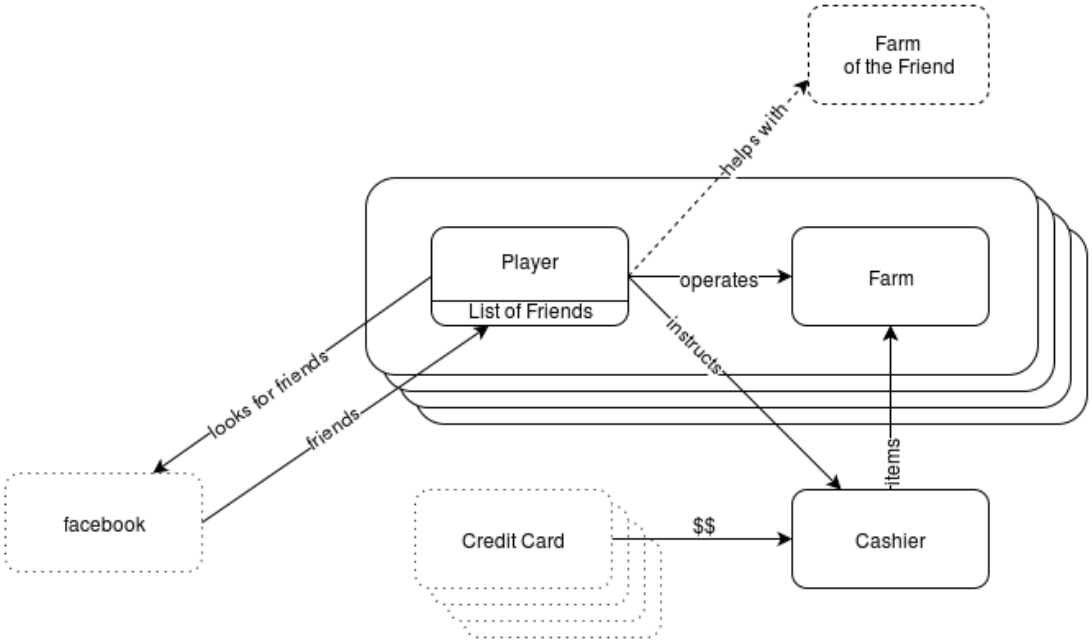
Examples of Entities and Interactions

To give you a bit of idea on entities and interactions, I'll try to describe typical entities for some of popular game genres. Note that (as with any other advice, in this book or elsewhere), *YOUR MILEAGE MAY VARY, and you need to think about specifics of your game rather than blindly copying typical entities mentioned below! Also note that example diagrams provided here illustrate only a few aspects of the game; in practice, your diagrams will usually be much more complicated.*

Social Farming and Farming-Like Games

While social games genre is very wide and is difficult to generalize, one sub-genre, social farming games, is straightforward enough to describe. In farming and farming-like games number of different entities and especially interactions between them are quite limited. Entities are usually limited to *players* and their *farms* (the latter including everything-which-can-be-found-on-the-farm). Interactions (beyond *player* interacting with their own *farm*) are also traditionally very limited (though they are important from the social point of view).

Fig II.1



NB: On all our example Entities&Interactions Diagrams, we will draw external (to our game) entities as dotted;[1] *feel free to use any other convention, the idea here is not to create yet another formal-and-unusable diagram language, but for you to visualize what your game is about.*

You should keep in mind that in most cases there is one significant caveat to remember about: it is a mistake to think that you can arbitrarily separate players onto different servers and allow only interactions within one such server. This "interactions are allowed only within one player server" model would work only until you introduce "Play with your Facebook

friends" feature (which will most likely be a Business Requirement, if not now, then a little bit later, see more on it in "On Arbitrary Player Separation" section below). And introducing "play with real-world friends" concept affects arbitrarily separated player servers dramatically: you no longer know which players want to play together, and inter-server interaction is no longer non-existent (and in fact the inter-server interaction has been observed to cause severe problems to some of the social games).

Overall, you should not rely on arbitrary player separation, even for social games

> "Introducing "play with real-world friends" concept affects arbitrarily separated player servers dramatically: you no longer know which players want to play together, and inter-server interaction is no longer non-existent.

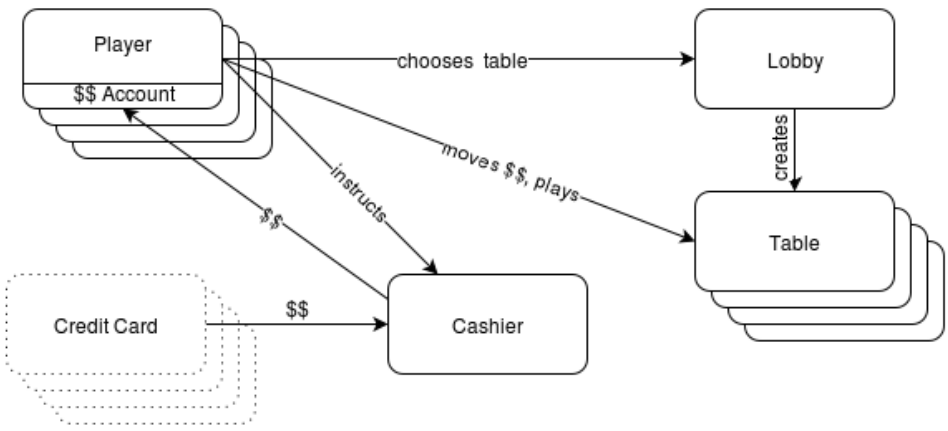
---

[1] at architecture stage, we'll need to insert appropriate gateways to communicate with these external entities, but we're not there yet

Casino Multiplayer Games

With casino multiplayer games, everything looks quite simple: there are *tables* and *players* at these tables. However, in some of the casino games (notably in poker), choosing an opponent is considered a skill, and therefore players should be able to choose who they want to play against. It implies another game entity – lobby, where the opponents can be selected. An example Entities&Interactions Diagram for Multiplayer Blackjack is shown on Fig II.2:

Fig II.2



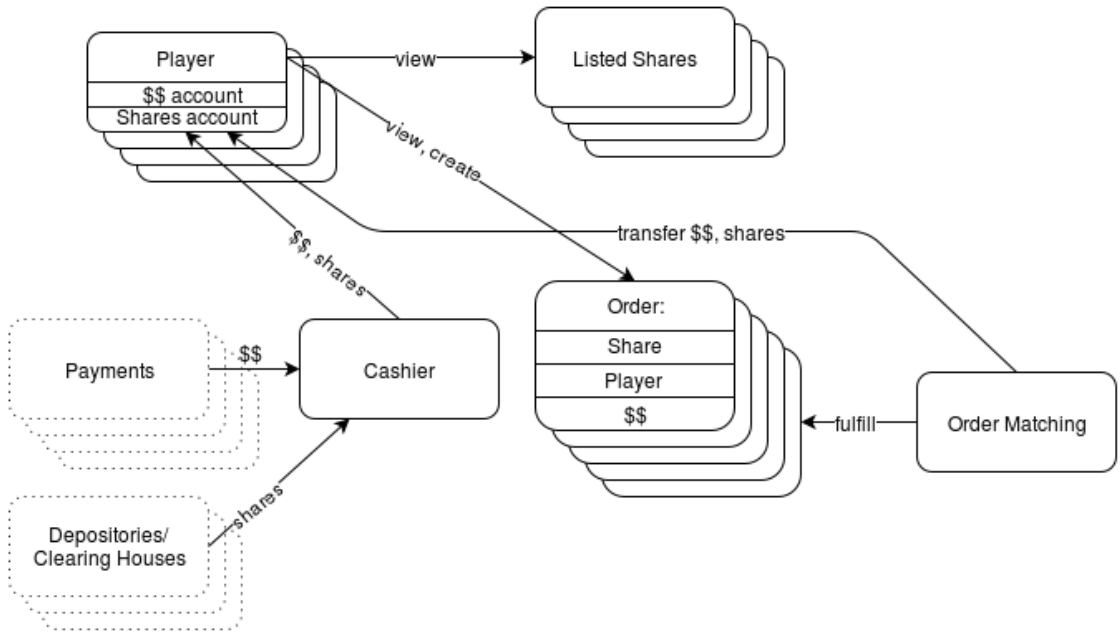
Note that for this example diagram, we've omitted social interaction; you will need to add it yourself, as it is appropriate for your specific game.

Stock Exchanges, Sports Betting and Auction Sites

In fact, stock exchanges and auction sites are so close to betting, that you'll be facing significant difficulties when trying to describe the difference between the three (except, obviously, for social stigma traditionally attached to betting). With stock exchanges, auction sites (think "eBay") and betting sites, entities involved are the same. It is *players* (though, of course, for a stock exchange you need to describe them as "traders" or "dealers"), and *stocks* (or sporting events/products). *Players* don't interact directly; however, indirect interaction does exist via creating some actions ("orders" or "bets") related to stocks or events/products.

Fig II.3 shows an example Entities&Interactions diagram for a stock exchange:

Fig II.3



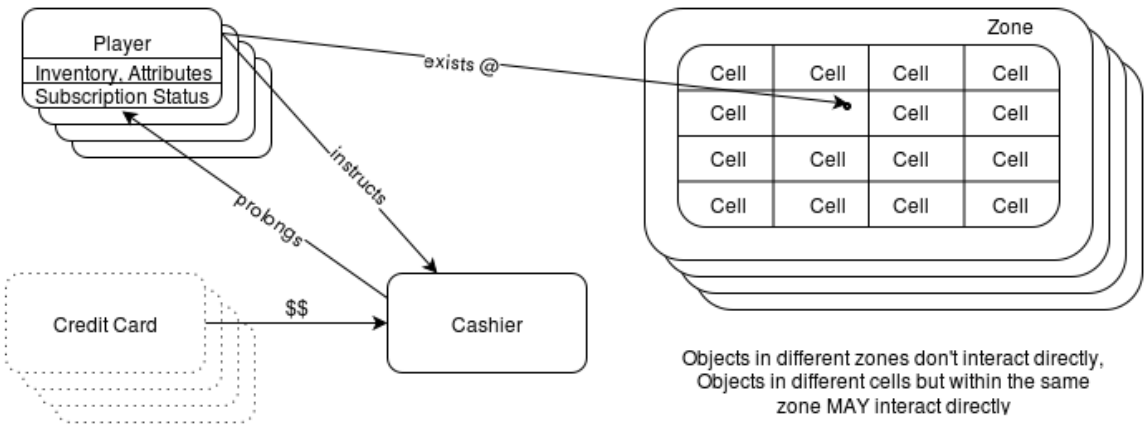
Virtual World Games (MMOTBS/MMORTS/MMORPG/MMOFPS)

Despite all the differences (including those which affect architecture a lot, as it will be described in Chapter [[TODO]]), from the point of view of the entities involved all the virtual world games tend to be more or less similar. In particular, in these games there are players (*PCs*), there are *NPCs*; also there are usually *cells* and *zones* containing those cells, which represent a virtual world (VW) where interactions between PCs and NPCs are occurring. What is important is that players in separate *cells* usually can interact, but players in different *zones* cannot.[2] The player option of choosing who she wants to play with, may or may not be provided; however, even if it is not provided, and you think that you can toss your players around your virtual worlds as you wish, arbitrary player separation (assigning player to servers without any inter-server interaction) becomes infeasible as soon as you introduce a

social feature such as “Recruit a Friend”. More on arbitrary player separation in “On Arbitrary Player Separation” section below.

Fig. II.4 shows an example Entities&Interactions Diagram for an MMORPG:

Fig II.4



[2] terms *cells* and *zones* may vary depending on the engine, but the idea is usually pretty much the same

On Arbitrary Player Separation

As we’ll see in Chapter [[TODO]], implementation-wise it is often very tempting to consider all your players as commodity, and to think that you can arbitrary assign players to servers without any communication between those servers (one of the things I’ve seen, was relying on limited “socializing” within a single server). However, for vast majority of games out there, it is a really bad idea. In short: don’t do it.

IRL
Abbreviation for 'In Real Life.' Often used in internet chat rooms to let people you are talking about something in the real world

The reason behind is the following. Even if your game as such seems to allow such arbitrary player separation (without any interaction between player servers), there are Big Fat Chances that you will need to implement some kind of interaction between players sooner rather than later. Pretty much any kind of IRL integration requires some kind of interaction between players just because they want it (and not because your rule engine decided that these two guys belong to the same server). In other words: if you don’t think that interaction of players just-because-they-want-it is a Business Requirement – think again.

> **and not in the internet world.**
> — *Urban Dictionary*

As just one simple example, even a simple "Play with your Facebook friend" feature requires players to "know" about each other, and to interact with each other. Moreover, you cannot possibly predict in advance which of your players will form a Facebook friendship some months later. In an another simple example, most of payment processing will require you to have some cross-server analysis to prevent fraudsters-who-cheated-one-server to cheat on another one. And so on. And so forth. And we didn't even start to speak about support people, who will need to manage all this mess. Trying to ignore this IRL integration is a pretty much sure way to a post-deployment disaster.

Think of your game as of one single planet with bunches of players living their, not as of cluster of non-interacting asteroids with a few players on each. As a rule of thumb, this stands even for such seemingly unconnected games as farming: as soon as you add some socializing (and it is usually a very important part of business strategy), you do need inter-player interactions, with no ability to control which players are communicating with each other.

Entities&Relations Diagram as a Starting Point to Architect Your Game

This Entities&Interactions Diagram you've just got is one of those things which will affect your architecture greatly. In particular, it is a starting point to realize what kinds of "implementation entities" (such as *servers, OS processes, DB tables, rows, and columns, etc.)* you need to implement your "game entities", and how to map game entities to implementation entities. This "how to map game entities into implementation entities" question will be discussed in Chapter [[TODO]].

[[To Be Continued…

This concludes beta Chapter II 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 III]]

EDIT: beta Chapter III. On Cheating, P2P, and [non-]Authoritative Servers, has been published.

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter I: "Business Requirements" from upcoming book "Dev*.

Chapter III. On Cheating, P2P, and [non-]Authoritative Server.. »

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

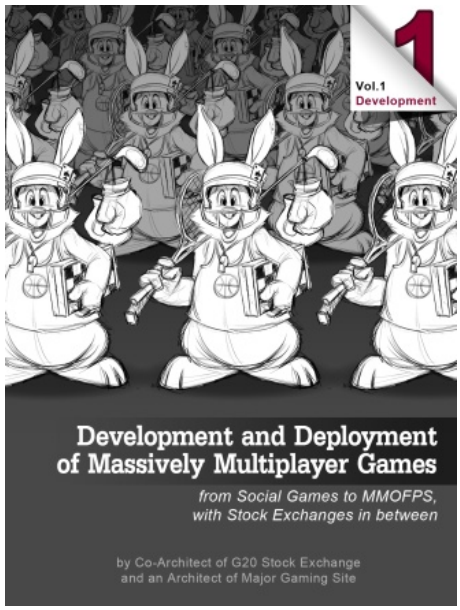
Copyright © 2014-2015 ITHare.com

Chapter III. On Cheating, P2P, and [non-]Authoritative Servers from "D&D of MMOG" book

posted November 9, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter III 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.]]



While developing an MMOG, there is one extremely important thing to remember about. This thing is almost non-existent for non-multiplayer games, and is usually of little importance for LAN-based multiplayer games. I'm speaking about player cheating.

Player cheating is One Big Problem for all successful MMO games. The problem is that ubiquitous for such games, that we can even say that if you don't have players cheating – it is either you're not looking for cheaters thoroughly enough, or you are not successful yet.

If you're popular enough, they Will find Reasons to Cheat

Two things are infinite: the universe and human stupidity; and I'm not sure about the universe.
— Albert Einstein —

You may think that players have no reason to cheat for your specific game. For example, if your game has nothing which can be redeemed for money – you may think that you're safe regardless of your number of players. In practice, it is exactly the other way around: if your game is popular enough, they will find a reason to cheat regardless of direct redemption options.

"The thing was that the players were able to push all their "play chips" on the table; while

Just one real-life example. Once upon a time, there was a poker site out there, where players got "play chips" for free, and were able to play with them. There was nothing which can be done with that "play chips", except for playing (so they cannot be redeemed for anything-which-has-real-value). At that time it seemed to the team that there was no reason to cheat on the site, none whatsoever, right? The real life has proven this assumption badly wrong.

The thing was that the players were able to push all their "play chips" on the table; while doing it has made very little sense, they were using the amount of their chips to imply "how good the player I am". And as soon as they started to brag about the

doing it has made very little sense, they were using the amount of their chips to imply "how good the player I am"

play chips, one guy has thought "hey, I can sell these play chips on eBay, and they will pay!" And as soon as eBay sales went on, the cheating went rampant (with lots of multiple accounts to get those free chips, and with lots of "chip dumping" to pass them along).

While I (and probably you) cannot imagine spending 20 real dollars to get two million of "play chips" with no other value than being able to boast that you're a Really Good Player (while you're not) – we know for sure that there is a certain percentage of people out there who will do it. If you're big enough, such things will happen for sure, the only question is about your site popularity and probabilities.

BTW, exactly the same aspect of human nature is currently successfully being exploited for monetization purposes by numerous modern games (especially social games); however, at this point we're not concerned about the exploiting human vices ourselves (it is a job for monetization guys, and beyond the scope of this book), but about technical aspects of preventing cheating.

The moral of the story:

Even if you think that players have zero reason to cheat
Given your site is popular enough, they will find such a reason

As soon as your game reaches 1'000 simultaneous players, you're likely to have singular cheaters. And when the number goes up to 100'000, you can be 100% sure that cheaters are there (and if you don't see them – it just means that you're not looking for them hard enough). While it depends on the kind of goodies you provide to your players, and numbers above may easily vary by an order of magnitude, I daresay that chances of you having a game with 100'000 simultaneous players and not having any cheaters, are negligible, pretty much regardless of what exactly is the game you're playing.

The Big Fat Hairy Difference from e-commerce

One thing to be kept in mind is that game cheaters are very different from e-commerce fraudsters. With e-commerce, those who're trying to get around the system, are either those trying to angle the promotions, or outright fraudsters.[1] When speaking about games, the reasons behind cheating are much more diverse. For players, in addition to all the reasons to cheat described above, there are many others.

For example, as it has happened with "play chips" (see [[TODO]] section above), people can cheat just to claim that they're better players than they really are. Or they can cheat because they feel that the game rules are unfair. Or they can cheat just because of (wrong) perception that "everybody else does it anyway", so they need to cheat just to level the field. Or they can just try to save some time by using "bots" for "grinding". Possibilities are really endless here.

This means that the line which separates "cheaters" from "honest players" is much more blurred with games, than in e-commerce. Throw in the fact that e-commerce fraud is an outright crime, and, say, using "bots" to avoid "grinding" is punishable at most by the ban on the site (which can be bypassed rather easily, at least unless the name on your credit cards is Rumpelstiltskin), and you will realize that

some of the people who would never ever cheat in e-commerce, will easily cheat in online games

While the number of "honest players" in online games still exceeds the number of "cheaters" by a wide margin, you cannot rely on your e-commerce experience of "Oh, merely 1% of our customers are cheating". Also you need to keep in mind that, due to much more significant interaction between players in games than in e-commerce,

unlike with e-commerce, even a small number of game cheaters can easily ruin the whole game ecosystem

Just as one example: if enough people are using bots to get an unfair advantage with your game (for example, to react to threats more quickly than a human can), your game will start deteriorate to the point of being completely unplayable. In other words: dealing with cheaters is not all about money, it is about preserving the very substance of your game.

"dealing with cheaters is not all about money, it is about preserving the very substance of your game.

[1] There are also people who want to use your site as testing grounds to improve their hacker skills or to brag about them after breaking you, and hacktivists, but fortunately, they're relatively few and far between.

Dealing with Cheaters

So, cheaters are pretty much inevitable. The question is: what can/should we do about it? In general, there are two things which need to be done.

First and foremost, you need to make sure that your architecture at least doesn't help cheaters. If it does – you will be in a Really Big Trouble as soon as your game becomes popular.

The second aspect of dealing with the cheaters is a direct cheater fighting, and it can usually (well, unless you're a stock exchange) be postponed until you deploy your game; then you need to start actively looking for cheaters, and to fix the problems as they arise. Details of the direct fighting with cheaters will be described in Chapters [[TODO]] and [[TODO]]; for now we just need to ensure that our architecture will allow to perform such cheater fighting without rewriting the whole thing.

Attacks: The Really Big Advantage of the Home Turf

When dealing with cheaters (in the realm of classical security they are usually named "attackers"), it is very important to understand the fundamental differences between two classes of the attack scenarios.

In the first class of scenarios, cheater/attacker tries to affect something which is under your direct control. This "something" can be your server, or a communication channel between the client and server. In this case you essentially have an upper hand to start with; while attacks are always a possibility, for this first class of attacks all of them are inevitably related to the bugs in your implementation.

In other words, whenever you have something which is under *your* control, you're generally safe, saving for implementation problems. Of course, there are lots of bugs to be exploited, but you do have a fighting chance, and as soon as a specific bug is fixed, the attacker will need to find another bug, which is not that easy if you've done your job properly.

Security by Obscurity is the use of secrecy of the design or implementation to provide security. A system relying on security through obscurity may

The second class of the attack scenarios is related to those cases when the attacker has your client software (or even hardware device) under his full control, and can do whatever-he-wants with it. In these cases, things are much much worse for you. In fact, whatever you do with your client software, the attackers are able to reverse engineer it and do whatever-they-want with it from that point.

The only protection you have in these attack scenarios, is some kind of obfuscation, but given enough effort (and we're not speaking about "the time comparable with life time of our sun"), any obfuscation can be broken. In terms of classical security, in this second class of attack scenarios, all you have at your disposal, is "Security by Obscurity", which is

> have theoretical or actual security vulnerabilities, but its owners or designers believe that if the flaws are not known, then attackers will be unlikely to find them.
>
> — *Wikipedia* —

traditionally not considered security at all; while we will need to resort to "Security by Obscurity" in some cases[2], we need to realize that

"Security by Obscurity",
while sometimes being the only protection available,
cannot be relied on

To summarize: when speaking about cheaters, an advantage of the "home turf" (having control over software/device) makes a huge difference. In particular, you cannot really protect software which you place into the attacker's hands. The situation in this regard is that bad, that even if you would be able to give each player a device, these devices would also be hacked (to see the spectrum of attack available, see, for example, [Skorobogatov]). In general, whatever-you-give-to-player should be considered hackable; the only thing we can do about it is to increase the cost of hacking, but preventing the hacking completely is out of question.[3]

---

[2] notably for bot fighting and for preventing duplicate accounts, where there are very few other ways of protection, if any

[3] In particular, Skorobogatov (being one the top researchers in the field), says that "given enough time and resources any protection can be broken"

Low-Impact and High-Impact Attacks

As mentioned above, we cannot really prevent 100% of the attacks on our games; some of the attacks (such as bots and duplicate accounts) are protected mostly by Security-by-Obscurity, and protection only by Security-by-Obscurity cannot be considered reliable, so some attackers will be able to slip in, at least for some time. Let's try to see what types of attacks are the most typical in gaming environment, and what is the impact of these attacks if they're successful.

Stealing User DB

One of the worst things which can happen with your game security-wise, is an attack on your user DB (the one which includes all the passwords, e-mails, etc.). It is an extremely juicy target for competitors (to have all the e-mails and to discredit your game at the same time), for disgruntled users [4], and for ordinary cheaters. The impact of such an attack is very high. Fortunately, user DB can be protected beyond "Security by Obscurity". Some details related to protection from stealing

of user DB will be described in Chapter [[TODO]].

---

[4] you can count on having your fair share of disgruntled users as soon as you have millions of players, even if you're 1000% fair and deliver on all your promises

DDoS

DDoS attacks are fairly easily to mount, and the battle really goes both on attacker's "home turf" and on your "home turf" at the same time. Fortunately, DDoS, while painful, usually do not last too long to cause too much trouble.

Affecting Gameplay

If your game is a soccer game, and somebody is able to ensure that they score a goal regardless of actual things happening on the field, you're in trouble. The very same thing applies to any kind of fight (if cheater is able to score a hit when shooting in the opposite direction, the things go pretty bad), and to any other type of competitive game in general. Even not-exactly-competitive games are subject to manipulation in this regard (especially as competitiveness is routinely introduces as different kinds of "leader boards" even to as non-competitive games as social farming).

"If your game is a soccer game, and somebody is able to ensure that they score a goal regardless of actual things happening on the field, you're in trouble.

Cheating-to-affect-gameplay will become known among the players pretty soon, and will break the trust to your game; in the extreme cases your game will become completely unplayable because of number of cheaters being too high. Therefore, the impact of such an attack can be classified as "high" (and becomes "extremely high" if the exploit is published). Whether you can protect from this type of attacks beyond "Security by Obscurity", depends on your architecture. We'll discuss the attacks related to affecting gameplay, in this chapter below.

Duplicate Accounts

Whatever your game is about, there is usually enough motivation for players to have duplicate accounts. As protection from duplicate accounts is mostly based on "Security by Obscurity" (except for paid accounts, where you can use credit card number or equivalent to identify your player), preventing duplicate accounts completely is not realistic, but we can still make it a bit more complicated for the attacker (especially on non-jailbroken phones and consoles). Fortunately, while

duplicate accounts are usually prohibited in T&C, and do affect gameplay in subtle ways, their impact on the game is usually very limited. Some ways of dealing with duplicate accounts will be described in Chapter [[TODO]].

Attacking Another User's Device

One of less common scenarios is placing a keylogger or some other kind of backdoor onto another player's device (PC/phone). Usually the aim for such an attack is to steal the user's password, but things such as "being able to make an action in the name of victim player while he's playing" are not unheard of. While technically this kind of attack is not our problem, from the user's perspective it is ("hey, somebody has logged in as me and lost that Great Artifact I had to somebody else without me even knowing about it!"), so this may need to be addressed if value of the things on player's account is high enough. Fortunately, impact of these attacks on the game ecosystem tends to be low. Dealing with them (if this is deemed necessary) is usually done with so-called two-factor authentication, which will be described in Chapter [[TODO]].

Bots

Bots (automated players) are well-known to be a part of any popular-enough MMOG. As soon as you have "grinding", there is an incentive to bypass the "grinding" and get the end result without spending hours (yes, for a good game, many people find that the "grinding" itself is fun, but this doesn't mean that *all* the players will agree with it). For the other games, reasons behind bots are different, but they do exist pretty much regardless; hey, bots are known to represent a Big Problem even for poker sites!

"Abuse scenarios using bots are endless.

Abuse scenarios using bots are endless. Just as one example, if there are goodies associated with new accounts, bots may automatically register, play enough to get those goodies, and then to pass them along to a consolidation account; then consolidation account can be used, say, to sell the stuff on eBay. Been there, seen that.

Bot fighting itself is a Big Task and if your game is really successful, you're likely to end up with a whole department dealing with bots; this is especially true as bot fighting is inherently in "Security-by-Obscurity" domain, so it is always pretty much going back and forth between you and the bot writers. Impact of bots on the game depends on them being published or not. For the unpublished bots, the impact is usually fairly low; for the published/commercially available bots, the impact is significantly higher. Fortunately, when you're fighting bots which are already published, it is you who has a "home turf" advantage half of the time (as you can get/buy the bot and dissect

it's logic pretty much in the same way as the bot writers have done with your program when they wrote the bot).

Attack Type Summary

Let's summarize the attacks mentioned above in one table:[5]

Attack	Impact	"Home turf" Advantage	Chapter where ProtectionWill Be Discussed
Stealing User DB	Very High	Yours	[[TODO]]
DDoS	Medium	None	[[TODO]]
Affecting Gameplay	High to Extremely High	Depends on Architecture	This one
Duplicate Accounts	Very Low	Cheaters'	[[TODO]]
Another User's Device	Low	None	[[TODO]] (only protecting logins will be discussed)
Non-published Bots	Low	Cheaters'	[[TODO]]
Published Bots	High	Back and Forth	[[TODO]]

As we can see from this table, only one of the attack types heavily depends on the architecture. Moreover, as it has one of the highest possible impacts on the game, we'd better to take a look at it before making any decisions related to the game architecture. Let's take a look at three different approaches to MMOG from this perspective.

[5] As usual, only typical values are provided, and your mileage may vary

Peer-to-Peer: Pretty Much Hopeless

SPOF
A single point of failure (SPOF) is a part of a system that, if it fails, will stop the entire system from working
— Wikipedia —

From time to time, a question arises in various forums: why bothering about servers, when we can have a SPOF-free, perfectly scalable system using P2P (as in "peer-to-peer")? Moreover, there exists an article [Skibinsky] which argues (I presume, with a straight face) that the client-server is not scalable, and that the future lies with MMO games being P2P.

With P2P, each client performs its own calculations, which are then used to determine the state of the game world. The way how the state of the game world is determined based on the results of individual computations, needs to be described in detail when you define your P2P game; ironically, I didn't really see any specific architectures specifying "how it should work", not even in [Skibinsky].

Still, let's take a look at P2P for gaming purposes, first of all from the point of view of "Affecting Gameplay" type of attacks. With P2P, we're essentially operating on "attackers home turf", making us to resort to "Security by Obscurity". [Skibinsky] recognizes it (albeit using different terminology), and proposes essentially three techniques to address this security issue.

"As soon as end-user himself wants to break code signing – it becomes at best "Security by Obscurity"

The first technique proposed in [Skibinsky] is code signing. However, the problem with the code signing of the game (as with any other code signing), is that *as soon as end-user himself wants to break code signing – it becomes at best "Security by Obscurity".* This is a direct result of the fact that when we're operating on attacker's "home turf", then all the signing keys (both private keys and root certificates) are under control of the attacker, making them essentially useless. BTW, [Skibinsky] also recognizes this weakness, stating "That still doesn't provide 100% security".

The second technique proposed in [Skibinsky], is a kind of "web of trust" system, with some of the nodes being trustworthy, and some being untrustworthy. While the idea looks attractive at the very first glance, there are two problems when applying it to the MMOG. Without (a) a reliable way to identify nodes, and with (b) not-really-tied-to-real-world-gaming-logins I expect any kind of "web of trust" thing to fall apart very quickly when facing a determined adversary; this is not to mention that those trusted nodes will quickly become a target for "Another User's Device" attack, which we cannot really do much about beyond protecting user's login.

The third technique to address inherent vulnerability of P2P systems to "Affecting Gameplay" attacks, is about cross-checking of the calculations-made-by-our-potential-cheater by other peers. While the idea sounds nice, on this way there are several Big Problems too.

First of all, the cross-checks are themselves vulnerable to cheating. Note that even Bitcoin system (which solves only a singular problem which is extremely narrow compared to general gaming) has an inherent 50% attack, and with inevitably selective nature of the cross-checks for gaming worlds (we simply cannot perform *all* the calculations on *all* the nodes), the number of nodes necessary to "take over the gaming world" is going to be drastically lower.

Second, all these cross-checks inevitably lead either to additional delays (which is unacceptable for the vast majority of the games), or to cross-checks being performed not in real time, but "a bit later". The latter approach raises another Big Question: "What shall we do with the game world when the cheater is caught?" Sure, we can ban the cheater for life (or more precisely, "until he opens a new e-mail account and registers again"), but what should we do with consequences of his cheating actions? This question, to the best of my knowledge, has no good general answer: leaving cheater deeds within the world is at best unfair to the others (not to mention that a cheater may cheat in the interests of another player), and rolling the whole world back whenever the cheater is found, is usually impractical ☹ (not to mention frustration of all the players not affected by cheating, but needing to lose significant time of their play).

As a result,

as of now, I don't see how peer-to-peer game (which goes beyond closed communities where people can trust each other), can provide reasonable protection from the cheaters

And we didn't even start to mention issues related to P2P complexity, which tends to go far beyond any complications related to client-server systems (especially if we're considering "web of trust" and "cross-checks").

That being said, I don't mean that P2P MMOG is unfeasible in principle; what I mean is that as of now, there is no good way to implement gaming over P2P (and whether such a way will ever be found, is an open question).

Non-Authoritative Client-Server: Simpler but still Hopeless

The whole idea of "non-authoritative server" has arisen when developers tried to convert classical single-player 3D game into an MMO. And classical single-player 3D game is very often pretty much built around 3D engine (which usually also performs some physics-related calculations).

To convert the classical-built-around-3D-engine game to an MMO, the simplest way was to just send the data about player movements to the other players, which then reflect these movements in their 3D engines. To send the data between the players, a "non-authoritative server" was routinely used, which is usually nothing more than taking the data from the clients and forwarding it pretty much "as is" to the other clients interested in what's going on in this part of the world.

"While simple to implement, essentially this "non-authoritative server" model is as much vulnerable as P2P model.

While simple to implement (and also avoiding NAT problems which tend to be quite unpleasant with P2P), essentially this "non-authoritative server" model is as much vulnerable as P2P model. With a non-authoritative server, you need to rely on clients not cheating; for example, if your client controls all the movements of your character, then it can say "hey, my coordinates just changed to the ones being 5 meters back" to avoid being hit, and server won't prevent your client from doing it. And if you try to detect this kind of cheating on the other clients, you will inevitably (pretty much the same way as described in section on P2P) get into need to a kind of "vote" on "who's good and who's bad", with different versions of the worlds appearing on different clients and needing to be consolidated, with a question "what to do if a cheater is detected", and so on.

Therefore, from a cheating prevention standpoint, non-authoritative servers are pretty much the same as P2P systems; the difference is that non-authoritative servers are simpler to implement, but at the cost of paying significantly more for server traffic (and arguably worse scalability).

However, these differences are pretty much irrelevant from preventing-cheating perspective, and

because of the cheating issues, I don't recommend using non-authoritative servers

While in theory, there might be games which can be protected using non-authoritative servers (as in "I don't have a formal proof that such games don't exist"), think more than twice when choosing to use non-authoritative servers. Oh, and make sure to re-read the "If You're Popular Enough, They Will Find Reasons to Cheat" section above.

Authoritative Server: As Safe As They Go

"With authoritative server, it is the server which moves the players (and other stuff) around; it is also the server who makes all the decisions about collisions, hits, etc.

The most popular approach when it comes to MMOG, is a so-called "authoritative server". In the usual approach to authoritative servers for a virtual world game, clients usually have a 3D engine, but this 3D engine is used purely for rendering, and not for decision-making. On the other hand, all the movements (not "coordinates resulting from movements", but more or less "player keypresses and mouseclicks themselves") are sent to the server, and it is the server which moves the players (and other stuff) around; it is also the server who makes all the decisions about collisions, hits, etc. Moreover, with an authoritative server, it is the server who makes all the changes in it's own copy of the game world (and the server's copy is an authoritative copy of the game world, which is then replicated to the clients to be rendered).

It means that for Virtual World games with an authoritative server, it is the server (and not the clients) who needs to implement the physics engine (though 3D rendering engines are still on the clients).

However, for fast-paced games, the delays of going-to-server-and-back-to-client with every keystroke are often not acceptable. In such cases, client often implements some kind of extrapolation (usually referred to as "client-side prediction") based on it's own picture of the game world; in other words, client shows the game world *assuming* that it's own picture of the game world is the same as the server one. For example, it may even show hits based on its own understanding of the game world. On the other hand, the client copy is *not* authoritative, so if the vision of the server and the vision of the client become different, it is server's copy which always "wins" (i.e. in such cases it is perfectly possible to see the hit which should have killed the opponent, only to see that the opponent is well alive). More details on client-side prediction for fast-paced games based on an authoritative servers will be provided in Chapter [[TODO]].

RTT is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement

From the point of view of cheaters trying to affect gameplay, authoritative servers are by far the best thing you can have. If you have enough checks on the server side, you always can enforce game rules with a relative ease. And while when using client-side prediction, temporary disagreements between clients and server are possible, it is always clear how to resolve the conflict; also these disagreements are always of a very-limited time (of the order of magnitude of RTT), which makes them not that noticeable in practice (except for first-

of that signal to be received.
— Wikipedia —

person shooters and fast-paced combat in RPGs).

It is worth noting that merely using authoritative server doesn't necessarily imply security; it merely *provides means* to make your game secure, and you will need to work on actual security later; fortunately, usually most of special work in this regard can be pushed down the road, after your game becomes more-or-less popular.

Authoritative Servers: Scalability is Imperfect, But is Workable

There is only one objection against this theory, and it is that the theory is wrong.
— C.N. Parkinson —

Before committing to go with authoritative servers, let's consider one common argument pushed by proponents of using-P2P-for-gaming; this is that client-server systems are not scalable. In particular, such an argument is presented in [Skibinsky].

The first line of the argument of alleged non-scalability of client-server games, revolves around the "$O(W^2)$ traffic estimate". The idea behind the argument goes as follows: first, let's consider a game world with W players within; now let's consider each player making some kind of change every N seconds, and let's assume that this change needs to be communicated to all the W-1 of the other players. Hence (they argue), for W players in the world, we need to push $O(W^2)$ bytes of traffic per second, making client-server non-scalable.

"In practice this $O(W^2)$ estimate doesn't really stand

If $O(W^2)$ would be indeed the case, then we'd indeed have quite significant scalability problems. However, in practice this $O(W^2)$ estimate doesn't really stand; let's take a closer look at it.

First, let's note that in real-world the number of people we're directly interacting with, has no relation to the number of people in the world. In virtual worlds it is normally *exactly the same thing* – the number of people (or other entities) players are interacting with, is limited not by the world population, but by our immediate vicinity, which in most cases has nothing to do with the world size. This is the point where $T=O(W^2)$ estimate falls apart (assuming reasonable implementation), and is replaced with $T=O(W)*C$ (when $W->infinity$), where C is the constant representing "Immediate Vicinity"[6] From this point, the estimate is no longer $T=O(W^2)$, but just $T=O(W)$ (with mathematicians among us sighing in relief).

Second, if $T=O(W^2)$ would be the case, it would mean that limits on the bandwidth of individual users would be hit pretty soon, so that even if somebody designs a world with everybody-to-everybody direct interaction all the time, it still won't run regardless of architecture (i.e. it won't run in client-server, but it won't run in P2P either).[7]

These observations are also supported by practical experiences; while the dependency of traffic from the world size is usually a bit worse than simple $T=O(W)$, it is never as bad as $T=O(W^2)$. So, we can make an observation that

In a properly implemented Client-Server game, for large enough world population W, traffic T is much closer to O(W) than to $O(W^2)$

This observation has one very important consequence: as soon as T is close to O(W), it means that your traffic is roughly proportional to world population W, which means that your expenses E is also proportional to W. On the other hand, within certain non-so-implausible assumptions, your income I is also more or less proportional to W. If this stands, it means that both your income I and your expenses E grow more or less proportional to W; this in turn means that if you were making money with 10'000 players, you will still make money (and even more of it) with 1'000'000 players.

---

[6] in [Skibinsky] this effect is referred to as *immediate action/reaction manifold,* and it is relied on to ensure P2P scalability, though for some reason it is mentioned only in the P2P context

[7] This effect is mentioned in [Skibinsky] too, though strangely enough, the way to mitigate this problem is once again mentioned only in the P2P context

A Very Example Calculation

To bring all the big-O notation above a bit more down to earth and to demonstrate these effects from a more practical perspective, let's consider the following example.

Let's consider a game where you can interact directly only with at most C=1000 other players, regardless of the world size and regardless of world population W. Of course, architecting and implementing your game to make sure that you don't send your updates to those-players-who-don't-really-need-them will be a challenge, but doing it is perfectly feasible[8]

Let's take the traffic estimate per player-interacting-with-another-player, from

[Skibinsky], i.e. as 50/3 bytes/sec (in practice, your mileage will vary, but if you're doing things right, usually it won't be off by more than an order of magnitude, so we can take it as a rather reasonable estimate). Let's also assume that all your players are paying you $10/month as a subscription fee. And let's further assume that your servers are residing in the datacenter (not in your office, see Chapter [[TODO]] why), and that you're paying $1000 per Gbit/s per month in your datacenter (once again, YMMV, but again, this number is not that far off – that is, if you're paying per GBit/s and have spent your time on finding a reasonably good deal; there is no doubt you can get it for a *lot* more money than that).

"At the same time, with your subscription fees you'll be making around $100'000/month, which means that your traffic costs are negligible.

Therefore, when you have 10'000 simultaneous players, you'll have traffic of *at most* 50/3*10000*1000 bytes/sec ~= 1.6e8 bytes/sec ~= 1.3 GBit/s; this will cost you around $1300/month. At the same time, with your subscription fees you'll be making around $100'000/month, which means that your traffic costs are negligible.[9]

When you grow to 1'000'000 simultaneous players, then your traffic per user will increase. As noted above, T won't grow as $T \sim W^2$, but there will be modest increase in per-user traffic because while each part of traffic T' (with sum of all T's being T) can be in most cases optimized to plain $T' \sim W$, in practice usually you're too lazy (or have too little time) to optimize all of them. For the purposes of our example let's assume that the traffic has grown 5-fold (you should be rather lazy – or busy – to get to 5x per-user traffic increase, but well, it can happen). Then, when you grow to 1'000'000 simultaneous players, your traffic will grow 500-fold, bringing it to 650 GBit/s, costing you $650000/month (in practice, the price will go lower for this kind of traffic, but let's ignore it for the moment). While this may sound as tons of money, you should note that with your $10/month subscription fee and million of users you'll be making $10M/month, which is still much more than enough to cover traffic bills (and note that if it becomes a problem, you still have that about-5x-times overhead, most of which can be recovered given sufficient development time).

---

[8] this has been demonstrated by numerous games-which-are-making-money out there, all of which, to the best of my knowledge, are client-server.

[9] Don't rush to buy that house on Bahamas though – while traffic costs are indeed negligible, other costs, especially advertisement costs to keep new players coming, are not

Summary: Authoritative Server is not ideal, but is The

Only Thing Workable

Let's summarize our findings about three different approaches in the following table:

	Scalability	Resilience to Cheating	Complexity
P2P	Good	Poor	from High (when not dealing with cheating) to Very High (Otherwise)
Non-Authoritative Server	Ok	Poor	from Low (when migrating from classical 3D game *and* not dealing with cheating), to High (if dealing with cheating)
Authoritative Server	Ok	Good	Medium

Actually, this table is not that different from that of in [Skibinsky]; the main difference between this book and [Skibinsky] is about estimating the impact of the differences between the approaches, in the real world. In particular, I'm sure that 'Poor' resilience to cheating is bad enough to rule out the relevant models, especially when there is an "Authoritative Server" model which has 'Ok' scalability (which is explained above) and 'Good' resilience to cheating. This point of view seems to be supported by MMOG developers around the world: as far as I know, as of now there are no real MMOGs which are implemented as P2Ps, there are quite a few of those based on non-authoritative servers, but the players are complaining about it, and there are lots of MMOGs based on authoritative servers.

Bottom Line: Yes, It is Going to Be Client-Server

TL;DR for Chapter III:

- Cheating is One Big Problem for MMOGs
- Players will cheat even if you're sure they have a zero reason to
- Gameplay cheating is one of the Big Potential Problems for your game

- P2P and non-authoritative servers provide very poor protection against Gameplay Cheating
- Despite some claims to the contrary, Client-Server (in particular, authoritative servers) can be made scalable
- Given the balance of pros and cons, Authoritative Servers look as the best option as of now; some (including myself) will even argue that in most cases it is *the only viable option*. While exceptions are theoretically possible, they are quite unlikely.

BTW, when speaking about Client-Server, I'm not ruling out multiple datacenters on the server side (this is referred to as "Grid Computing" in [Skibinsky]); on the other hand, delegating any kind of authority and decision-making to the client looks way too risky for practical MMO.

[[To Be Continued…

This concludes beta Chapter III 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 IV, "DIY vs Re-use"

EDIT: beta Chapter IV. DIY vs Re-Use: In Search of Balance, has been published.

]]

[–] References

[Skibinsky] Max Skibinsky, "The Quest for Holy Scale", in "Massively Multiplayer Game Development 2", pp. 339-373.
[Skorobogatov] Sergey Skorobogatov, "Hardware Security of Semiconductor Chips: Progress and Lessons"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter II: "Game Entities and Interactions" from upcoming b…*

Chapter IV. DIY vs Re-Use: In Search of Balance from upcomi… »

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

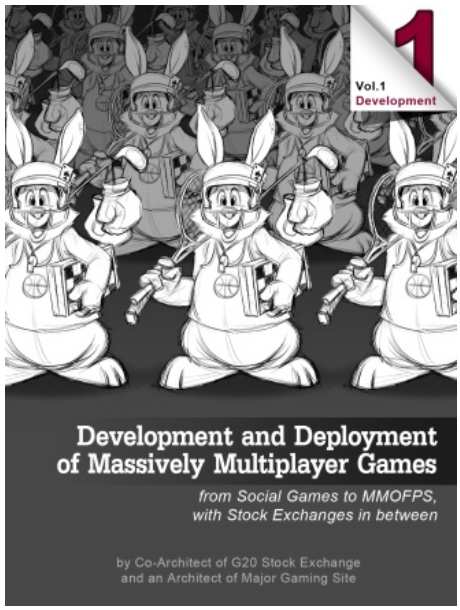
Copyright © 2014-2015 ITHare.com

Chapter IV. DIY vs Re-Use: In Search of Balance from upcoming book "Design&Development of MMOG"

posted November 16, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter IV 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.]]



DIY
Initialism of *do it yourself*
— Wiktionary —

In any sizable development project there is always a question: "What should we do ourselves, and what we should reuse?" Way too often this question is answered as "Let's re-use whatever we can get our hands on", without understanding all the implications of re-use (and especially about the implications of improper re-use, see, for example, [NoBugs2011]). On the other hand, an opposite approach of "DIY Everything" can easily lead to the projects which cannot possibly be completed on one person's life time, which is usually "way too long" for games. In this chapter we will try to discuss this question in detail.

In the game realm the answers to “DIY vs Re-Use” question reside on a pretty wide spectrum, from “DIY pretty much nothing” to “DIY pretty much everything”. On the one end of the spectrum, there are games which are nothing more but “skins” of somebody-else’s game (in such cases, you’re usually able to re-texture and re-brand their game, but without any changes to gameplay; changes to meshes and/or sounds may be allowed or disallowed). In this case, you’re essentially counting on having better marketing than your competition (as everything else is the same for you and your competition). This approach may even bring some money, but if you’re into it, you’re probably reading the wrong book (though if you’re running your own servers, some tricks from Part [[TODO]] might still be useful and may provide some additional competitive advantage, but don’t expect miracles in this regard).

“On the one end of the spectrum, there are games which are nothing more but “skins” of somebody-else’s game. In this case, you’re essentially counting on having better marketing than your competition

On the other end of the spectrum, there are game development teams out there which try to develop pretty much everything, from their own 3D engine, their own TCP replacement, and their own channel security (using algorithms which are “much better” than TLS), to their own graphics and sounds (fortunately, cases when the developers are trying to develop their own console and their own OS are very few and far between). This approach, while may be fun to work on, may

have problems with providing results within reasonable time, so your project may easily run out of money (and as the investors understand it too, running out of money will happen sooner rather than later).

Therefore, it is necessary to find a good balance between the parts which you need to re-use, and the parts you need to implement yourself.

Business Perspective: DIY Your Added Value

First of all, let's take a look at "DIY vs Re-Use" question from the business point of view. While business perspective is not exactly the point of this book, in this case is way too intertwined with the rest of our discussion to set it aside.

From the business point of view, you should always understand what "added value" your project provides for your customers. In other words – what is that thing which differentiates you from your competition? What is the unique expertise you provide to your players?

When speaking about "DIY vs 3rd party reuse" question, it is safe to say that

At least, you should develop your Added Value yourself

The motivation behind the rule above is simple: if you're re-using everything (including gameplay, world map, and meshes), with only cosmetic differences (such as textures) then your game won't be really different from the other games which are doing the same thing. To succeed commercially, you need a distinguishing factor (sometimes 'pure luck' qualifies as such, but luck is not something you can count on).

The rule of Added Value is normally taken care of at a business level. However, even after this rule is taken into consideration, you still need to make "DIY vs reuse" decisions for those things which don't constitute the added-value-for-end-users (or at least are not *perceived* to constitute the added value at the first glance). In this regard, usually it more or less boils down to one of three approaches described below.

Engine-Centric Approach: an Absolute Dependency a.k.a. Vendor Lock-In

Probably the most common approach to game development is to pick a game engine, and to try building your game around that engine. Such game engines usually don't implement all the gameplay (instead, they *provide you with a way* to implement your own gameplay on top of the engine), so you're fine from the Added

Value point of view. For the sake of brevity, let's refer to this "3rd-party engine will do everything for us" approach as a much shorter "Engine-Centric" Approach.

"The biggest problem with building your game around 3rd-party game engine is that in this case, the game engine becomes your Absolute Dependency

The biggest problem with building your game around 3rd-party game engine is that in this case, the game engine becomes your Absolute Dependency; in other words, it means that "if the engine is discontinued, we won't be able to add new features, which will lead us to close sooner rather than later". Another way to see the very same thing, is in terms of Vendor Lock-In: as soon as you have an Absolute Dependency, you're locked in to a specific 3d engine vendor, and vice versa. Therefore, we will use terms Absolute Dependency and Vendor Lock-In interchangeably.

While by itself Absolute Dependency a.k.a. Vendor Lock-In is not a show-stopper for building around 3rd-party game engine (and indeed, there are many cases when you should do just that), you need to understand implications of this Absolute Dependency.

First of all, (as we will discuss in more detail in Chapter [[TODO]]), for "Games with Undefined Life Span" (as defined in Chapter I), the risks of having 3rd-party Absolute Dependency are much higher than for "Games with Limited Life Span". Having your game engine as a Vendor Lock-In for a limited-time project is often fine even if your choice is imperfect; having the very same Absolute Dependency "forever and ever till death do us part" is a much bigger deal, which can easily lead you to a disaster if your choice turns out to be a wrong one.

Moreover, usually, for "Games with Undefined Life Span", you shouldn't count on assumptions such as "Oh, it is a Big Company so they won't go down" (while the company might not go down, they still may drop this engine, or drop support for those-features or those-platforms you cannot survive without). While for a limited time, such risks can be estimated and are therefore manageable (in many cases, we can say with enough confidence "they will support such-and-such feature in 3 years from now"), relying on a 3rd party doing something "forever and ever" is usually too strong of an assumption.

Engine-Centric Approach: Pretty Much Inevitable for MMORPG/MMOFPS

In spite of the risks above, it should be noted that there are several MMO genres where developing a game engine yourself is rarely feasible. In particular, it applies to MMORPGs and MMOFPS. The engines for these games tend to be extremely complicated, and it will normally take much-more-time-than-you-have to develop them. Fortunately, in this field there are quite a few very decent engines with pretty good APIs separating the engine itself and your game logic. In future chapters we will keep in mind three specific game engines, and will discuss their pros and cons with relation to the issues we are raising. These engines are Unity 5, Unreal Engine 4, and CryEngine (previously known as CryEngine 3). Apologies to fans of other game engines, but I simply cannot cover all of the engines in existence; still, principles behind are usually rather similar, so you should be able to make your own judgements based on general principles outlined in this book.

"In future chapters we will keep in mind three specific game engines, and will discuss their pros and cons with relation to the issues we are raising. These engines are Unity 5, Unreal Engine 4, and CryEngine

For MMORTS the situation is much less obvious; depending on specifics of your game, there are much more options. For example, (a) you may want to use 3rd-party 3D engine like one of the above (though this will work only for relatively low number of units, you need to study very carefully engine's capabilities in this regard), (b) you may use 2D graphics (or pre-rendered 3D, see Chapter [[TODO]] for details), with your own engine, (c) you may want to develop your own 3D engine (optimized for large crowds but without features which are not necessary for you), or (d) you may even make a game which runs as 2D on some devices, and as 3D on some other devices (see Chapter [[TODO]] for further discussion of dual 2D/3D interfaces).

For all the other genres, whether to use 3rd-party engine, is a completely open question, and you will need to decide what is better for your game; often, for non-MMORPG/non-MMOFPS games, and if your game is intended to have an Undefined Life Span, it is better to develop game engine yourself than to re-use a 3rd-party game engine (even when you have your own game engine, you may use 3rd-party 3D rendering engine, or even several such 3D engines – see Chapter [[TODO]] for further details).

And if you're going to re-use a 3rd-party engine (for whatever reason), make sure to read and follow "You Still Need to Understand How It Works" section below.

Engine-Centric Approach: You Still Need to Understand How It Works

When introducing 3rd-party game engine as an Absolute Dependency, you still need to understand how the engine works under the hood. Moreover, you need to know a lot about engine-you're-about-to-choose *before* you make a decision to allow the engine Vendor to Lock you In. Otherwise, 6 months down the road you can easily end up in situation "oh, this engine *apparently* cannot implement this feature, and we absolutely need it, so we need to scrap everything and start from scratch using different game engine".

Of course, there will be tons of implementation details which you're not able to know right now. On the other hand, you should at least go through this book and see how what-you-will-need maps into what-your-engine-can-provide, aiming to:

- understand what exactly are the features you need
- make sure that your engine provides those features you need
  - if some of the features you need, are not provided by your game engine (which is almost for sure for an MMOG), at least that you should know that you can implement those "missing" features yourself on top of your game engine

While this may look time consuming, it will certainly save a lot of time down the road. While introducing Absolute Dependency may be a right thing to do for you, this is a Very Big decision, and as such, MUST NOT be taken lightly.

Engine-Centric Approach: on "Temporary" dependencies

Nothing is so permanent as a temporary government program
— Milton Friedman —

"Eliminating dependency on your game engine is pretty much hopeless without rewriting the whole thing.

If you want to use 3rd-party game engine to speed up development, and count on the approach of "we'll use this game engine for now, and when we're big and rich, we will rewrite it ourselves", you need to realize that removing such a big and fat dependency as game engine, is not realistic. Eliminating dependency on 2D engine, sound engine, or any other such engine may be possible (though requires extreme vigilance during development, see "Modular Approach: on "Temporary" dependencies" section below). On the other hand, eliminating dependency on your game engine is pretty much hopeless without rewriting the whole thing.

The latter observation is related to number of "interface points"[1] which arise when you integrate with your game engine; for a typical game engine you have lots and lots of such points. Moreover, these interface points tend to be of very different

nature (ranging from mesh file formats to API callbacks with pretty much everything else you can think of, in between). To make things worse, the better is the game engine you're using, the more perfectly legitimate uses you have for those interface points, and the more locked-in you become as a result (while having all the good reasons for doing it). Due to these factors, IMNSHO, the task of making your program game-engine-agnostic is orders of magnitude more complicated than making your program cross-platform (which is also quite an effort to start with), so think more than twice before attempting it.

---

[1] let's define an "interface point" as a point, where your program (and more generally, your whole game development process) interacts with the game engine

"Re-Use Everything in Sight" Approach: An Integration Nightmare

If you've decided not to make a 3rd-party engine your Absolute Dependency, then the second approach often comes into play. Roughly it can be described as "we need such-and-such feature, so what is the 3rd-party component/library/... we want to ~~borrow~~ re-use to implement this feature?"

Unfortunately, way too many developers out there think that this is exactly the way it should be done. (mis-)Perception along the lines of "hey, re-use is good, so there can be nothing wrong with re-use" is quite popular with developers; for managers it is "it saves on the development time" pro-reuse argument which usually hits home.

"When your code does *nothing* beyond dealing with peculiarities and outright bugs of 3rd-party libraries, *it cannot possibly*

However, in practice it is not that simple. Such "reuse everything in sight" projects way too often become an integration nightmare. As one of developers of such a project (who was responsible for writing an installer) has put it: "Our product is load of s**t, and my job is to carry it in my hands to the end-user PC, without spilling it around". As you can see, he wasn't too fond of the product (and the product didn't work too reliably either, so the product line was closed within a few years). Even worse, such "reuse everything in sight" projects were observed to become spaghetti code very quickly; moreover, from my experience, when your code does *nothing* beyond dealing with peculiarities and outright bugs of 3rd-party libraries, *it cannot possibly be anything but spaghetti*. Oh, and keep in mind that indiscriminate re-use has been observed as a source of some of the worst software bugs in the development history [NoBugs2011].

be anything but spaghetti

The problem with reusing everything you can get your hands on, can be explained as follows. With such an indiscriminate re-use, some of modules/components you are using, will be inevitably using less-than-ideal for the job; moreover, even if the component is good enough now, it may become much-less-than-ideal when[2] – the Business Requirements change. And then, given that the number of your not-so-ideal components is large enough, you find yourself in an endless loop of "hey, trying to do this with Component A has broken something-else with Component B, and fixing it in Component B has had such-and-such undesired implication in Component C, and so on…".

To make sure that managers (who're usually very fond of re-use, because of that "it saves the development time" argument), also understand the perils of indiscriminate re-use: you (as a manager) need to keep in mind that indiscriminate re-use very frequently leads to "oh, we cannot implement this incoming Business Requirement because our 3rd-party component doesn't support such-and-such feature" (which, if happens more than a few times over the life span of the project, tends to have rather bad impact on the bottom line of the company). Or describing it from a different perspective: if your developers are doing their own component, it is them who're responsible that this "we cannot implement Business Requirement" thing never happens; at the moment when you force (or allow) them to "use such and such library", you give them this excuse on a plate 😉 .

BTW, to make it perfectly clear: I'm *not* arguing that *any* re-use is evil; it is only *indiscriminate* re-use which should be avoided. What I am arguing for, is "Responsible Re-use" (a.k.a. "Modular") approach described a little bit below.

---

[2] it is indeed 'when', not 'if'! – see Chapter I

"DIY Everything": The Risk of Never-ending Story

Another approach (the one which I myself am admittedly prone to), is to write everything yourself. Ok, very few developers will write OS themselves, but for most of the other things you can usually find somebody who will be arguing that "this is the most important thing in the universe, and you simply MUST do it this way, and there is nothing which does it this way, so we MUST do it ourselves".

There are people out there arguing for rewriting TCP over UDP[3], there are people out there arguing that TLS is not good enough, so you need to use your own security protocol, there are people out there arguing for writing crypto-quality RNG based their own algorithm[4] , there are quite a few people out there writing their own in-memory databases for your game,

and there are even more people out there arguing for writing your own 3D engine.

> "There are people out there arguing for writing crypto-quality RNG using their own algorithm

Moreover, depending on your circumstances, some of these things may even make sense; however, writing *all of these things together* will lead to a product which will never be released, almost inevitably.

As a result, with all my dislike to the 3rd-party dependencies, I shall admit that we do need to re-use *something.* Now the next question is: "What exactly we should re-use, and what should we write ourselves?"

---

[3] I shall admit that I was guilty of such suggestion myself for one of the projects, though it has happened at a later stage of game development, which I'm humbly asking to consider as a mitigating circumstance

[4] once it took me several months to convince external auditor that implementing RNG *his way* is not the only "right" RNG implementation, with the conflict eventually elevated to The Top Authority on Cryptography (specifically, to Bruce Schneier)

"Responsible Re-Use" a.k.a. "Modular" Approach: Looking for Balance

As it was discussed above (I hope that I was convincing enough), there are things which you should re-use, and there are things which you shouldn't. The key, of course, is all about the question "What to Re-use and What to DIY?". While the answer to this question goes into realm of art (or black magic, if you prefer), and largely follows from the experience, there are still a few hints which may help you in making such a decision:

- Most importantly, all decisions about re-use MUST NOT be taken lightly; it means that *no clandestine re-use should be allowed,* and that all re-use decisions MUST be approved by an architect (or by consensus of senior-enough developers). Discussion on "to re-use or not to re-use" MUST include both issues related to licensing, and issues related to reuse-being-a-good-thing-in-the-long-run (you can be sure that arguments about it being a good thing in the short run *are* brought forward).
- To decide whether a specific re-use will be a good-thing-in-the-long-run, the following hints may help:
  - "glue" code is almost universally DIY code; while it is unlikely that you will have any doubts about it, for the sake of completeness I'm still mentioning it here

- if writing your own code will provide some Added Value (which is visible in the player terms), it is a really good candidate for DIY. And even if it doesn't touch gameplay, it can still provide Added Value.
  One example: if your own communication library will provide properties which lead to better user-observable connectivity (than the one currently used by competition), it does provide Added Value (or a competitive advantage, if you prefer), and therefore may easily qualify for DIY (of course, development costs still need to be taken into account, but at least the idea shouldn't be thrown away outright). In another practical example, if you're considering re-using Windows dialogs (or MFC), and your own library provides a way to implement i18n without the need for translators to edit graphics (!) for each-and-every dialog in existence – it normally qualifies as an "Added Value" (at least compared to MFC, let's postpone further discussion about i18n until Chapter [[TODO]]).

"If writing your own code will provide some Added Value (which is visible in the player terms), it is a really good candidate for DIY

- If you're about to re-use something with a very well defined interface (API/messages/etc.), *and* where the interface does what you want *and* is not likely to change significantly in the future – it is a really good candidate for re-use. Examples include TLS, JPEG library, TCP, and so on.
- If you're about to re-use something which has much more non-trivial logic inside than it exposes APIs outside – it *might* be a good candidate for re-use. One such example is 3D engines (unless you're sure you can make them significantly better than the existing ones, see the item on Added Value above). It is usually a good idea, however, to have your own isolation layer around such things, to avoid them becoming an Absolute Dependency. Such an isolation layer should be usually written in a manner described in [[TODO]] section below (as described there, dependencies *are* sneaky, so you need to be vigilant to avoid them).
- If you're about to re-use something for the client side (or for non-controlled environment in general), *and* it uses a DLL-residing-in-system-folder (i.e. even if it is a part of your installer, it is installed in a place, which is well-known and can be overwritten by some other installer) – double-check that you cannot make this DLL/component private[5], otherwise seriously consider DIY. This also applies to re-use of components, including Windows-provided components.
  The reason for this rather unusual (but still *very* important in practice) recommendation is the following. It has been observed for real-world apps with install base in millions, that reliance on something-which-you-

don't-really-control introduces a pretty nasty dependency, with such dependencies failing for some (though usually small) percentage of your players. If you have 10 such dependencies each of which fails for mere 1% of your users – you're losing about 1-(0.99[10])~=9% of your player base (plus also people will complain about your game not working, increasing your actual losses many-fold).
Real-world horror stories in this regard include such things as:

"Don't think that such failures "are not your problem" - from the end-user perspective, it is *your* program which crashes, so it is *you* who they will blame for the crash

- program which used IE to render not really necessary animation, failing with one specific version of IE on player's computer
- some Win32 function (the one which isn't really necessary and is therefore rarely used) was used just to avoid parsing .BMP file, only to be found failing on a certain brand of laptops due to faulty video drivers[6]
- some [censored] developer of a 4th party app replaced stock mfc42.dll with their own "improved" version causing quite a few applications to fail (ok, this one has became more difficult starting from Vista or so, but it is still possible if they're persistent enough).

And don't think that such failures "are not your problem" – from end-user perspective, it is *your* program which crashes, so it is *you* who they will blame for the crash. In general, the less dependencies-on-specific-PC-configuration your client has – the better experience you will be able to provide to your players, and all the theoretical considerations of "oh, having a separate DLL of 1M in size will eat as much as 1M on HDD and about the same size of RAM *while our app is running*" are really insignificant compared to your players having better experience, especially for modern PCs with ~1T of HDD and 1G+ of RAM.

- Keep in mind that "reuse via DLLs" on the client side introduces well-defined points which are widely (ab)used by cheaters (such as bot writers); this is one more reason to avoid re-using DLLs and COM components (even if they're private). This also applies to using standard Windows controls (which are very easy to extract information from); see Chapter [[TODO]] for further discussion of these issues. Re-use via statically linked libraries is usually not affected by this problem.[7]

- If nothing of the above applies, and you're about to write yourself something which is central/critical to your game – it may be a good candidate for DIY. The more critical/central the part of your code is – the more likely related changes will be required, leading to more and more integration work, which can easily lead to the cost of integration exceeding the value provided by the borrowed code. About the same thing from a different angle: for the central/critical code you generally want to have as much control as you possibly can.
- If nothing of the above applies, and you're about to re-use something which is of limited value (or is barely connected) to your game – it may be a good candidate for re-use. The more peripheral the part of the code is – the less likely related changes will have a drastic effect on the rest of your code, so costs of the re-integration with the rest of your code in the case of changes will hopefully be relatively small.
- Personally, if in doubt, I usually prefer to DIY, and it works pretty well with the developers I usually have on my team. However, I realize that I usually work with the developers who qualify as "really really good ones" (I'm sure that most of them are within top-1%), so once again, your mileage may vary. On the other hand, if for some functionality all the considerations above are already taken into account and you're still in doubt (while being able to keep a straight face) on "DIY vs re-use" question, probably this specific decision on this specific functionality doesn't really matter too much.

"The more critical/central the part of your code is – the more likely related changes will be required, leading to more and more integration work, which can easily lead to the cost of integration exceeding the value provided by the borrowed code.

Note that as with most of the other things in real life, all the advice above should be taken with a good pinch of salt. Your specific case and argumentation may be very different; what is most important is to *avoid making decisions without thinking,* and to *take at least considerations listed above into account.*

The approach presented above, can be seen as a "Responsible Re-Use"; on the other hand, we'll refer to it quite a lot in the subsequent chapters, so for the sake of brevity, we'll usually name it as "Modular Approach" (or "Modular Architecture").

---

[5] roughly equivalent to "moving it to your own folder"

[6] why such a function has had anything to do with drivers – is anybody's guess

[7] Strictly speaking, statically linked well-known libraries can also make life of cheater a bit easier, but this effect is usually negligible compared to the Big Hole you're punching in your own code when using DLLs

Modular Approach: Examples

Here are some examples of what-to-reuse and what-not-to-reuse (YMMV really significantly) under the "Responsible Re-Use" (a.k.a. "Modular") guidelines:

- OS/Console: usually don't really have choice about it. Re-use.
- Game Engine: depends on genre, but for MMORPG/MMOFPS is pretty much inevitable (see "Engine-Centric Approach: Pretty Much Inevitable for MMORPG/MMOFPS" section above)
- TCP/TLS/JPEG/PNG/etc.: usually a really good idea to re-use. One potential (though *quite rare*!) exception is TCP, but see detailed discussion on it in Chapter [[TODO]] first. On client-side it is much better to re-use them (and pretty much everything else) as static libraries rather than as DLLs, due to the reasons outlined above
- 3D Engine: an open question; see further discussion on it in Chapter [[TODO]].
- Ever-changing shared controls such as IE HTML Control: many of them are still error-prone, buggy, and are changed a lot depending on version of IE installed on client PC. Hence, it is better to avoid re-using them if you can (replacing them with much simpler 3rd-party libraries, which usually aren't that function-rich, but are much more predictable).
- On the other hand, much simpler basic controls such as text, don't have the problem of being changed too often; still, you need to consider effects related to bot fighting as mentioned above and described in Chapter [[TODO]], so using these for critical information might be not a good idea; on the third hand 🙂 , usually you will be able to replace them later without too much hassle, so it might be ok to use them to speed things up (aiming to replace them later, when bots become a problem)
- Core logic of your game. This is where your added value is. DIY
- Something which is very peripheral to your game. This is what is not likely to cause too much havoc to replace. Re-use (as long as you can be sure what exactly you're re-using on the client side, see above about DLLs etc.)

"Still, you need to consider effects related to bot fighting, so using these for critical information might be not a good idea

Modular Approach: on "Temporary"

dependencies

If you're planning to use some module/library only temporary (to speed up first release), and re-write it later, "when we're big and rich", it might work, but you need to be aware of several major caveats on the way. First of all, you need to realize that this won't work for replacing the whole game engine (see "Engine-Centric Approach: on "Temporary" dependencies" section above).

Second, you need to be extremely vigilant when writing your code. Otherwise, when the "we're big and rich" part comes, the 3rd-party module will become so much intertwined with the rest of your code, that separating it will amount to rewriting everything from scratch (which is rarely an option for an up-and-running MMOG).

So, if you're going to pursue this approach, you should at least:

- write in Big Bold Letters in your design documents, that your dependency on Module X is only temporary, and that you plan to get rid of it later
- make your own Module MyX with it's own API. The closer your own APIs to the needs of your game – the better; dumb wrappers around the 3rd-party modules should be avoided. Your Module MyX should do what-your-specific-game-needs-to-do (and not what-3rd-party-module-is-able-to-provide). The mapping between the two API sets ("your own" one and "3rd-party" one) is what your own module should do, however trivial it may seem at first (don't worry, if your APIs are centered around your game, and not around the 3rd-party comphonent, the "meat" of your own module will grow as you develop). As Peter Wolf has aptly put it: "wrap and wrap some more".
- Use ONLY your-own-API for the rest of the code (i.e. in the code beyond your Module MyX)
- make sure that everybody on the team knows that you're NOT using API of the 3rd-party module directly
- try to prohibit APIs of the 3rd-party module in your build system
    - In C++ this can be achieved, for example, using pimpl idiom for your own module and prohibiting direct inclusion of 3rd-party header files by anybody-except-for-your-own-engine
- unless you have managed to prohibit 3rd-party APIs in your build system (see above), you should have special *periodic* reviews to ensure that nobody uses these prohibited APIs. It is much much simpler to avoid these APIs at early stages, than trying to remove them

pimpl idiom
also known as an opaque pointer, Bridge pattern, handle classes, Compiler firewall idiom, d-pointer, or Cheshire Cat, is a special case of an opaque data type, a datatype declared to be a pointer to a record or data structure of some unspecified type.
— Wikipedia —

later (which can amount to rewriting really big chunks of your code)

"Dependencies are sneaky, and it takes extreme vigilance to avoid them.

While these rules may look overly harsh and too time-consuming, practice shows that without following them you get over-95%-chance that you won't be able to replace the 3rd-party module when you need it. Dependencies are sneaky, and it takes extreme vigilance to avoid them. On the other hand, if you don't want to do these things – feel free to ignore them, just be honest to yourself and realize that Module X is one of your Absolute Dependencies forever with all the resulting implications.

Summary

TL;DR of Chapter IV:

- DON'T take "re-use vs DIY" question lightly; if you make Really Bad decisions in this regard, it can easily kill your game down the road
- Consider using Engine-Centric approach, but keep in mind that Absolute Dependency (a.k.a. Vendor Lock-In) that you're introducing. Be especially cautious when using this way for Games with Undefined Life Span (as defined in Chapter I). On the other hand, this approach is pretty much inevitable for MMOFPS/MMORPG games. If going Engine-Centric way, make sure that you understand how the engine of your choosing implements those things you need.
- If Engine-Centric doesn't work for you (for example, because there is no engine available which allows to satisfy *all* your Business Requirements), you generally should use "Responsible Re-use" a.k.a. "Modular" approach as described above. If going this way, make sure to read the list of hints listed in ""Responsible Re-use" a.k.a. "Modular" Approach: Looking for Balance" section above.

[[To Be Continued…

This concludes beta Chapter IV 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, "Modular Architecture: Client-Side"]]

EDIT: Chapter V(a). Modular Architecture. Client-Side. Graphics has been published.

[–] References

[NoBugs2011] 'No Bugs' Hare, "Overused Code Reuse"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter III. On Cheating, P2P, and [non-]Authoritative Server*..

Due to Popular Demand: PDFs of Beta Chapters from "Develop.. »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: Code Reuse, game, multi-player

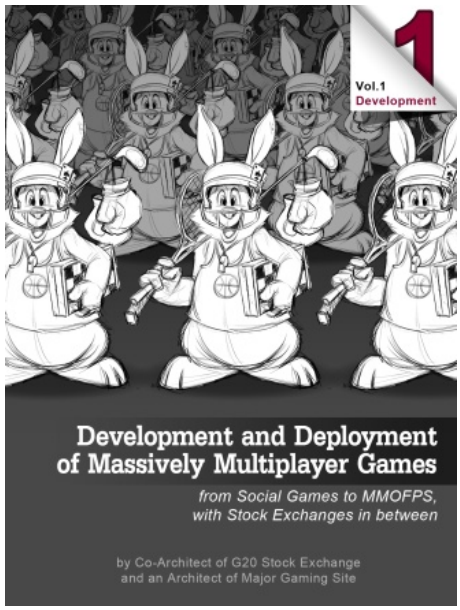
Copyright © 2014-2015 ITHare.com

Chapter V(a). Modular Architecture: Client-Side. Graphics from "D&D of MMOG" upcoming book

posted November 23, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

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



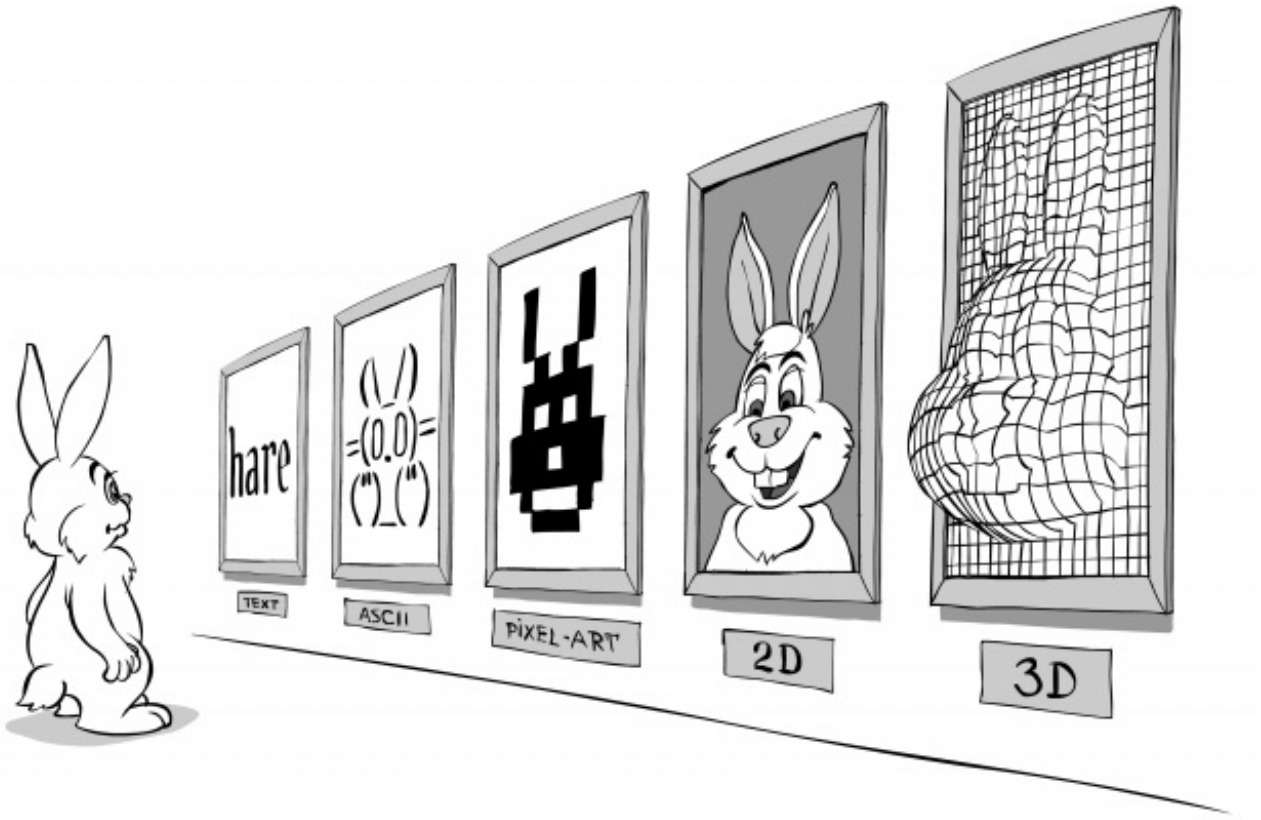
– How do you program an elephant? – One byte at a time!
— (almost) proverb —

As we've discussed in Chapter IV, there are basically only two viable approaches for building your game: we named one of them an "Engine-Centric Architecture", and another a "Modular Architecture". Which of these approaches is right for your game, depends a lot on the genre and other Business Requirements; the choice between the two was more or less explained in Chapter IV.

In this chapter, we'll discuss "Modular Architecture" in more detail. If you're going to implement your game as an "Engine-Centric" one, you still need to read this chapter; while most of these decisions we're about to discuss, are already made for you by your game engine, you still need to know what these decisions are (and whether you like what specific engine has chosen for you); and whatever-your-engine didn't decide for you, you need to make the right decisions yourself. Applicability of the findings from this Chapter to "Engine-Centric Architecture" and to specific popular game engines, will be discussed in the next Chapter ([[TODO]]).

Graphics

One of the first things you need when dealing with client-side, is graphics engine. Here, depending on specifics of your game, there are significant differences, but there are still a few things which are (almost) universal. **Note that at this point we're not about to describe subtle implementation details of graphics engines (these will be discussed in Chapter [[TODO]]). For the time being, we only need to figure out a few very high-level things, which allow us to describe the engine(s) we need in very general terms (to filter out those which are obviously not a good fit for your game) and to know enough to be able to draw an overall client-side architecture.**



On Developers, Game Designers, and Artists

For most of the games out there, there is a pretty obvious separation between developers and artists. There is usually a kind of mutual understanding, that developers do not interfere in drawing pictures (making 3D models, etc. etc.), and artists are not teaching developers how to program. This, however, raises a Big Fat Question about a toolchain which artists can use to do their job. These toolchains are heavily dependent on the graphics type, on the game you're using, etc. etc. When making decisions about your graphics, you absolutely need to realize which tools your artists will use (and which file formats they will produce so that you can use these formats within your game).

For some genres (notably FPS and RPG), there are usually also game designers. These folks are sitting in between developers and artists, and are responsible for creating levels, writing quests, etc. etc. And guess what – they need their own tools too ☹ .

Actually, these toolchains are so important, that I would say that at least half of the value that game engine provides to your project, comes from these toolchains. If you're going to write your own engine – you need to think about these toolchains, as they can easily make-or-break your game (and if you're using 3rd-party game engine – make sure that the toolchain they're providing, is understandable for both your artists and your developers – and for game designers too, if applicable).

"Actually, these toolchains

On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In

are so important, that I would say that at least half of the value that game engine provides to your project, comes from these toolchains.

These days, if you want to use a 3rd-party graphics engine, most of the time you won't find "graphics engine", but will need to choose between "game engines". And "game engines" tend to provide much more functionality than "graphics engines", which has many positives, but there is also one negative too. Additional features provided by "game engines" in addition to pure graphic rendering capabilities, may include such things as processing user input, support for humanoid-like creatures (which may include, for example, inverse kinematics), asset management, scripting, network support, toolchains, etc. etc. etc. And most of these features even work.

However, there is a dark spot in this overall bright picture. Exactly the same thing which tends to help a lot, backfires. The thing is that the more features the engine has – the more you will want to use ("hey, we can have this nice feature for free!"). And the more features you use – the more you're tied to a specific 3rd-party game engine, and this process will very soon make it your Absolute Dependency (as defined in Chapter IV), also known as a Vendor Lock-In.

It is not that Absolute Dependencies are bad per se (and, as mentioned in Chapter IV, for quite a few games the advantages of having it outweigh the negatives), but if you have an Absolute Dependency – it is Really Important to realize that you *are* Locked In, and that you SHOULD NOT rely on throwing it away in the future.

Just one example where this can be important. Let's consider you writing a game with an Undefined Life Span (i.e. you're planning your game to run for a really long while, see Chapter I for further details); then you've decided (to speed things up) to make a first release of your game using a 3rd-party game engine. Your game engine of choice is very good, but has one drawback – it doesn't support one of the platforms which you do want to support (for example, it doesn't support mobile, which you want to have ASAP after the very first release). So you're thinking that "hey, we'll release our game using this engine, and then we'll migrate our game from it (or will support another graphics engine for those platforms where it doesn't run, etc.)".

In theory, it all sounds very good. In practice, however, unless you're extremely vigilant (see on it a bit below), and not taking special measures to deal with dependencies, you'll find yourself in a hot water. By the time when you want to migrate away, your code and game in general will be that much intertwined and interlocked with the game engine, that separating them will amount to a full rewrite (which is rarely

possible within the same game without affecting too many subtle gameplay-affecting issues which make or break your game). It means that in our hypothetical example, you won't be able to support mobile devices, *ever* (well, unless you scrap the whole thing and rewrite it from scratch, which will almost inevitably require a re-release at least on a different set of servers, if not under a different title). This situation tends to be even worse for 3D game engines (to the point that I'm not sure that it is possible at all to avoid your 3D game engine Locking you In).

> "Unless you're extremely vigilant, and not taking special measures to deal with dependencies, you'll find yourself in a hot water.

The only way to avoid this kind of (very unpleasant) scenarios, is to be extremely vigilant and prohibit the use of all the game features, unless their use is explicitly allowed (and before allowing the use of a certain feature, you need to understand – and document! – how you're going to implement this feature when you are migrating away from the engine). For further details on the measures which you need to take to ensure that your component (such as graphics engine) doesn't become your Absolute Dependency – see Chapter IV.

Once again – having an Absolute Dependency is not necessarily evil, but if you have one – you'd better realize that you're pretty much at the mercy of the engine developer (the one who has successfully locked you in).

Games with Rudimentary Graphics

Now, let's start considering different types of graphics which you may need for your game. First of all, let's see what happens if your game requires only a minimal graphics (or none at all).

Contrary to the popular belief, you *can* build a game without any graphics at all, or with a very rudimentary one. When speaking about rudimentary graphics, I mean static graphics, without animation, just pictures with defined areas to click. Such games-with-rudimentrary-graphics are not limited to obvious examples such as stock exchanges, but also include some of social games which are doing it with a great success (with one such example being quite popular Lords&Knights).

> "Contrary to the popular belief, you *can* build a game without any graphics at all, or with a very rudimentary

If your graphics is non-existent or rudimentary, you can (and probably should) write your graphics engine all by yourself. It won't take long, and having a dependency on a 3rd-party engine merely to render static images is usually not worth the trouble.

Artist's toolchain is almost non-existent too; all artists need to work with rudimentary graphics, is their favourite graphics editor to provide you with bitmaps of sizes-which-you-need.

one.

Games with 2D Graphics

The next step on the ladder from non-existent graphics to the holy grail of realistic ray-traced 3D[1] is 2D graphics. 2D graphics is still very popular, especially for games oriented towards mobile phones, and for social games (which tend to have mobile phone version, so there is a strong correlation between the two). This section also covers 2D engines used by games with pre-rendered 3D graphics.

In general, if you're making a 2D game, your development, while more complicated than for games with rudimentary graphics, will be still much much simpler than that of 3D game [2]. First of all, 2D graphics (unlike 3D graphics) is rather simple, and you can easily write a simple 2D engine yourself (I've seen a 2D engine with double-buffering and virtually zero flickering written from scratch within about 8-10 man-weeks for a single target platform; not too much if you ask me).

Alternatively, you can use one of the many available "2D game engines"; however, you need to keep in mind the risk of becoming Locked-In (see section "On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In" above). In particular, if you're planning to replace your 2D game engine in the future, you should stay away from using such things as "2D Physics" features provided by your game engine, and limit it to rendering only. In practice, it *is* possible to avoid Vendor Lock-In (and keep your options to migrate from this 2D engine, or to add another 2D or even 3D one alongside it, etc.); however, it still requires you to be extremely vigilant (see section "On Using Game Engines as Pure Graphics Engines, and Vendor Lock-In" above), but at least it has been done and is usually doable.

MVC

Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three

One good example of 2D game engine (which is mostly a 2D graphics engine), is [Cocos2D-X]. It is a popular enough cross-platform (including iOS, Android, and WinPhone, and going mobile is One Really Popular Reason for creating a 2D game these days), and has API which is good enough for practical use. If you're developing *only* for iOS, [SpriteKit] is a good choice too. BTW, if you're vigilant enough in avoiding dependencies, you can try making your game with Cocos2D-X, and then to support SpriteKit for iOS only (doing it the other way around is also possible, but usually more risky unless you're absolutely sure that most of your users are coming from iOS). NB: if you're serious about such development, make sure to make Logic-to-Graphics layer as described in "Logic-to-Graphics Layer" section below.

> **interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user**
> — *Wikipedia* —

About using 2D functionality of the primarily 3D engines such as Unity or Unreal Engine: personally, I would stay away from them when it comes to 2D development (for my taste, they are way too locking-in for such a relatively simple task as 2D). Such engines would have a Big Advantage for quite a few genres if they could support both 2D and 3D graphics for the same world (kind of MVC for games, also similar to Logic-to-Graphics layer as described below), but to the best of my knowledge, none of the major game engines provides such support. [[NOTE to BETA TESTERS: If you know about such capabilities in these or other engines, please let me know]].

About toolchains. For 2D, artist's toolchains are usually fairly simple, with artists using their favourite animation editor (for example, "Adobe After Effects", but there are other options out there; "Adobe Flash" has also been reported to support "sprite sheets" starting from CS6 version). As a result of their work, they will provide you with sprites (for example, in a form of series of .pngs-with-transparency, or "sprite sheets").

---

[1] Yes, I do know that nobody does raytracing for games (yet), but who said that we cannot daydream a bit?
[2] Hey, isn't it a good reason to scrap all 3D completely in the name of time to market? Well, probably not

On pre-rendered 3D

Now, let's discuss see what happens if your game is supposed to be a 3D game. In this case, first of all, you need to think whether you really need 3D, or you will be fine with so-called pre-rendered 3D.

When speaking about pre-rendered 3D, the idea is to create your 3D models, and 3D animations, but then, instead of rendering them (such as "render using OpenGL") in real-time, to pre-render these 3D models and animations into 2D "sprites", to ship these 2D sprites with your game instead of shipping full 3D models, and then to render them as 2D sprites in your 2D graphics engine.

Fully 3D pre-rendered games[3] allow you to avoid having 3D engine on clients, replacing it with much simpler (and much more portable) 2D engine.

Usually, full 3D pre-rendering won't work good for first-person games (such as MMORPG/MMOFPS) but it may work

pretty good even for (some kinds of) MMORTS, and for many other kinds of popular MMO genres too. Full 3D pre-rendering is quite popular for platforms with limited resources, such as in-browser games, or games oriented towards mobile phones.

> "Fully 3D pre-rendered games allow you to avoid having 3D engine on clients, replacing it with much simpler (and much more portable) 2D engine.

Technically, fully pre-rendered 3D development flow consists of:

- 3D design, usually made using readily available 3rd-party 3D toolchain. For this purpose, you can use such tools as Maya, 3D Max, Poser, or – for really adventurous ones – Blender. 3D design is not normally done by developers, but by designers. It includes both models (including textures etc.) and animations.
- pre-rendering of 3D design into 2D sprites. Usually implemented as a bunch of scripts which "compile" your 3D models and animations into 2D sprites, including animated sprite sequences; the same 3D tools are usually used for this 3D-to-2D rendering
- rendering of 2D sprites on the client, using 2D engine(s)

As an additional bonus, with 3D pre-rendering you don't need to bother with getting low-poly 3D models for your 3D toolchain, and can keep your 3D models in as high number of polygons as you wish. Granted, mostly these high-poly models won't usually make any visual difference (as each of 2D sprites is commonly too small to notice the difference, though YMMV), but at least you won't need to bother with polygon number reduction (and you can be sure that your artists will appreciate it, as low-poly-but-still-decent-looking 3D models are known to be a Big Headache).

3D pre-rendering is certainly not without disadvantages. Two biggest problems of 3D pre-rendering which come to mind, are the following. First of all, you can pre-render your models only at specific angles; it means that if you're showing a battlefield in isometric projection, pre-rendering can be fine, but doing it for a MMOFPS (or any other game with a first-person view) is usually not feasible. Second, if you're not careful enough, the size of your 2D sprites can easily become huge. There are other less prominent issues related to 3D pre-rendering, which we'll discuss in Chapter [[TODO]], but for our purposes now these two things should be enough (i.e. if you're fine with them – you can keep considering 3D pre-rendering).

"If you can survive 3D pre-rendering without making your game unviewable (and without making it too huge in size), you can make your game run on the platforms which have no 3D at all (or their 3D is hopelessly slow)

On the positive side, if you can survive 3D pre-rendering without making your game unviewable (and without making it too huge in size), you can make your game run on the platforms which have no 3D at all (or their 3D is hopelessly slow); I'm mostly speaking about smartphones here (while smartphones have made huge improvements in 3D performance, they are still light years away from PCs – and it will probably stay this way for a long while).

Artist's toolchains are usually not a problem for pre-rendered 3D. In this case, artists are pretty much free what to use (though it is still advisable to use one tool across the whole project) ; it can be anything ranging from Maya to Poser, with 3D Max in between. They can keep all their work within this tool, and to provide you with ways to produce 2D sprites. In most cases, your job in this regard is about making artists backup their work on regular basis, and about writing the scripts for automated "build" of their source files (those in 3D Max or whatever-else-they're-using) into 2D.

Bottom Line. Whether you want/can switch your game to 3D pre-rendering – depends, but at least you should consider this option (that is, unless your game is an MMOFPS/MMORPG). While this technique is often frowned upon (usually, using non-arguments such as "it is not cool"), it might (or might not) work for you.

Just imagine – no need to make those low-poly models, no need to worry that your models become too "fat" for one of your resource-stricken target platforms as soon as you throw in 100 characters within one single area, no need to bother with texture sizes, and so on. It does sound "too good to be true" (and in most cases it will be), but if you're lucky enough to be able to exploit pre-rendering – you shouldn't miss the opportunity.

If you manage to get away with pre-rendered 3D, make sure to read section on 2D graphics above.

---

[3] in fact, partial 3D pre-rendering is also perfectly viable, and is used a lot in 3D games which do have 3D engine on the client-side, but this is beyond the scope of our discussion until Chapter [[TODO]]

Games with 3D Graphics

If you have found that your 3D game is not a good match for pre-rendered 3D, you do need to have 3D engine on the client-side. This tends to unleash a whole lot of problems, from weird exchange formats between toolchain and your engine, to the inverse kinematics (if applicable); we'll discuss some of these problems in Chapter [[TODO]], for now let's just write down that non-pre-rendered 3D is a Big Pain in the Neck (compared to the other types of graphics). If you do need a 3D engine on client side, you basically have two distinct options.

Option 1 is along "DIY" lines, with you writing your own rendering engine over either OpenGL, or over DirectX. In this case, be prepared to spend a lot of time on making your game look anywhere reasonable. Making 3D work is not easy to start with, but making it look good is a major challenge. In addition, you will need to remember about the artist's toolchain; at the very least you'll need to provide a way to import and use files generated by popular 3D design programs (hint: supporting wavefront .obj is not enough, you'll generally need to dig much deeper into specifics of 3D-program-you're-supporting and its formats).

"Making 3D work is not easy to start with, but making it look good is a major challenge.

On the plus side, if you manage to survive this ordeal and get a reasonably looking graphics with your own 3D engine, you'll get a solid baseline which will give you a lot of flexibility (and you may need this flexibility, especially if we're speaking about the games with Undefined Life Span).

Option 2 is to try using some "3D game engine" as your "3D engine". This way, unless you already decided that your game engine is your Absolute Dependency, is a risky one. 3D game engines tend to be so complicated, and have so many points of interaction with the game, that chances are that even if you're Extremely Vigilant when it comes to dependencies, you won't be able to replace the engine later. Once again – I am not saying that Vendor Lock-In is necessarily a bad thing, but you do need to realize that you're Locked In.

Logic-to-Graphics Layer

Unless you've already decided that you want to be 100% Locked In, it is usually a good idea to have a separation layer between your logic and your graphics engine (whether it is 2D engine or 3D engine). Let's name this separation layer a Logic-to-Graphics Layer; this layer resides completely on the client side, and doesn't really affect your communication protocols or the server side. In a sense, it can be seen as a subset of a Model-View-Controller pattern (with game logic representing Model, and graphics engine representing View).

Let me explain the idea on one simple example. If your game is a blackjack, client-

side game logic needs to produce rendering instructions to your graphics engine. Usually, naive implementations will just have client-side game logic to issue instructions such as “draw such-and-such bitmap at such-and-such coordinates”. This approach works well, until you need to port your client to another device (in the extreme case – from PC to phone, with the latter having much less screen real estate).

With Logic-to-Graphics layer, your client-side blackjack game logic issues instructions in terms of “place 9S in front of player #3 at the table” (and *not* in terms of “draw 9S at the (234,567) point on screen”). Then, it becomes a job of Logic-to-Graphics Layer to translate this instruction into screen coordinates. And if your game is a strategy, client game logic should issue instructions in terms of “move unit A to position (X,Y)” (with the coordinates expressed in terms of simulated-world coordinates, not in terms of on-screen coordinates(!)), and again the translation between the two should be performed by our Logic-to-Graphics layer.

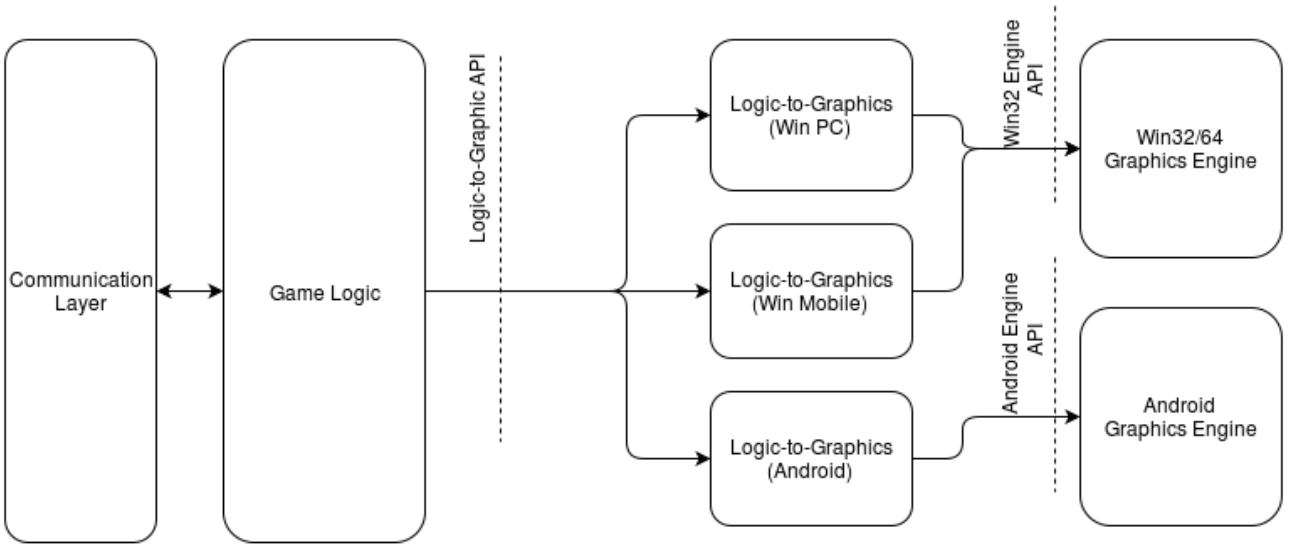


Fig V.1

One example incarnation of a system built using Logic-to-Graphics approach, is shown on Fig 1. Here, “Game Logic” doesn’t depend on a graphical engine (or a platform) and can be developed separately (which is very important because it will change very frequently). In contrast, two “Graphical Engines” are specific to the respective platforms, but they don’t know/depend on game logic at all, and are very-rarely changed. The “Logic-to-Graphics” layer is a “glue” layer which belongs in between “game logic” and “graphical engine”; by design, it depends both on game logic and graphical engine (ouch); however (provided that there is a reasonably clean separation achieved, see examples above) it doesn’t change nearly as often as “game logic” itself, so the whole thing becomes manageable. On Fig. 1, there are three implementations of “Logic-to-Graphics” layer: one is for Android and two for Windows; the reason for having two different implementations of Logic-to-Graphics layer for the same Win32 graphics engine, is that PC and mobile versions are usually quite different in terms of layout, and therefore it may be

simpler just to have two different implementations of Logic-to-Graphics layer (which is responsible, among other things, for translation of coordinates into screen coordinates).

If doing it this way, you'll get quite a few benefits.

"First of all, you will have a very clear separation between the different layers of the program, which tends to help a whole lot in the long run.

- First of all, you will have a very clear separation between the different layers of the program, which tends to help a whole lot in the long run.
- Second, even if you're supporting only one platform now, you're leaving the door open to adding support for all the platforms you might want, in the future. This includes such things as adding an option to have a 3D version to your currently-2D-only game.
- Third, you don't have a strong dependency on any graphical engine, so if in 5 years from now a new, much-better engine will arise, you'll be able to migrate there without rewriting the whole thing.
- Fourth, such a clean separation facilitates using authoritative servers (which we'll discuss in Chapter [[TODO]], and which are extremely important for the reasons described there).
- Fifth, with Logic-to-Graphics layer, for quite a few genres you'll be able to produce a command-line client, which comes handy for testing (including automated testing, and testing of game logic without being affected by graphics), and also for development-of-the-new-features while the graphics is not ready yet.

We've discussed the benefits of this Logic-to-Graphics layer, but what about the costs? Is it all 100% positive, or there are some drawbacks? In fact, I can only think of two realistic negatives for having it:

- There is a certain development overhead which is necessary to achieve this clean separation. I'm not talking about performance overhead, but about development overhead. If the game logic developer needs to get something from the graphics engine, he cannot just go ahead and call the graphics-engine-function-which-he-wants. Instead, an interface to get whatever-he-needs should be created, has to be supported by all the engines, etc. etc. It's all easily doable, but it introduces quite a bit of mundane work. On the other hand, I contend that in the long run, such clean interfaces provide much more value than this development overhead takes away; in particular, clean interfaces have been observed as a strong obstacle to the code becoming "spaghetti code", which is already more-than-enough enough to justify them.

- A learning curve for those game developers coming from traditional limited-life-span (and/or not-massively-multiplayer) 3D games. In these classical games (I intentionally don't want to use the term "old-fashioned" to avoid being too blunt about it 😉) everything revolves around the 3D engine, so for these developers moving towards the model with clean separation between graphics and logic will be rather painful. However, unless you decided your game to be Engine-Centric, you need to move away from this approach anyway, and even for those guys-coming-from-classical-3D-games this clean separation model will be quite beneficial in the long run, so I wouldn't say that this drawback is that important.

Personally, for games with a potentially unlimited life span (and not having 3rd-party game engine as an Absolute Dependency a.k.a. Vendor Lock-In), I almost universally recommend to implement this Logic-to-Graphics Layer.

Dual Graphics, including 2D+3D Graphics

In quite a few cases, you may need to support two substantially different types of graphics. One such example is when you need to support your game both for PC and phone; quite often the difference between available screen real estate is too large to keep your layout the same, so you usually need to redesign not only the graphics as such, but also redesign layout.

> "In such cases of dual graphics, it is paramount to have Logic-to-Graphics layer as described above.

In such cases of dual graphics, it is paramount to have Logic-to-Graphics layer as described above. As soon as you have Logic-to-Graphics layer, adding new type of graphics is a breeze. You just need to add another implementation of Logic-to-Graphics layer (using either the same graphical engine, or different one, depending on your needs), and there is no need to change game logic (!). These two different implementations of Logic-to-Graphics layer may have different APIs on the boundary with graphics engines, but they always have the same API on the boundary with Game Logic. The latter fact will allow you to keep developing your game logic without caring about the specific engines you're using.

The reason why it is so important to have Logic-to-Graphics Layer is simple – for such a frequently changed piece of code as a client-side game logic, maintaining two separate code bases is usually not realistic. Pretty much any feature you're adding, will require some changes in game logic on the client side (hey, at least you need to receive that new server message you've just introduced and parse it!), and having two code bases for game logic will mean that you need to duplicate all such changes all the time. I've observed much more than one competitor going the route of multiple code bases, only to see that one of these code bases starts to lag behind the other,

and scrapping it 6 months down the road. It just illustrates the main point: you do need to keep your frequently-changed portions of the code as a single code base. And Logic-to-Graphics Layer allows to achieve it.

Of course, if you need to add a new instruction which comes from game logic to Logic-to-Graphics Layer (for example, if you're adding a new graphical primitive), you will still need to modify both your implementations of the Logic-to-Graphics Layer. However, if your separation API is clean enough, you will find that such changes, while still happening and causing their fair share of trouble, are the orders of magnitude more rare than the changes to game logic; this difference in change frequencies is the difference between workable and unworkable one.

An extreme case of dual graphics is dual 2D+3D graphics. Not all the game genres allow it (for example, first-person shooters usually won't work too good in 2D), but if your game genre is ok with it, *and* you have Logic-to-Graphics separation layer, this becomes perfectly feasible. You just need to have 2 different engines, a 3D one and a 2D one (they can be in separate clients, or even switchable in run-time), and an implementation of Logic-to-Graphisc layer for each of them. As soon as you have this, Bingo! – you've provided your players with a choice between 2D and 3D graphics (depending on their preference, or platform, or whatever else). Even better, when using a Logic-to-Graphics layer, you can start with the graphics which is simpler/more important/whatever, and to add another graphics (or even multiple ones) later.

[[To Be Continued…

This concludes beta Chapter V(a) 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(b), "Modular Architecture: Client-Side. Programming Languages"

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

]]

[–] References

[Cocos2D-X] http://www.cocos2d-x.org/
[SpriteKit] https://developer.apple.com/spritekit/

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« Due to Popular Demand: PDFs of Beta Chapters from "Develop..

Chapter V(b). Modular Architecture: Client-Side. Programmin.. »

Filed Under: Distributed Systems, Programming, System Architecture
Tagged With: 2D, 3D, client, game, graphics, multi-player

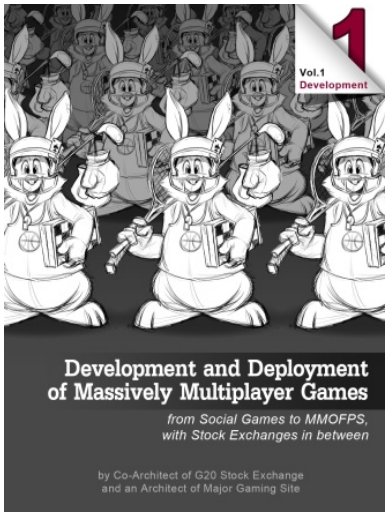
Copyright © 2014-2015 ITHare.com

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.]]

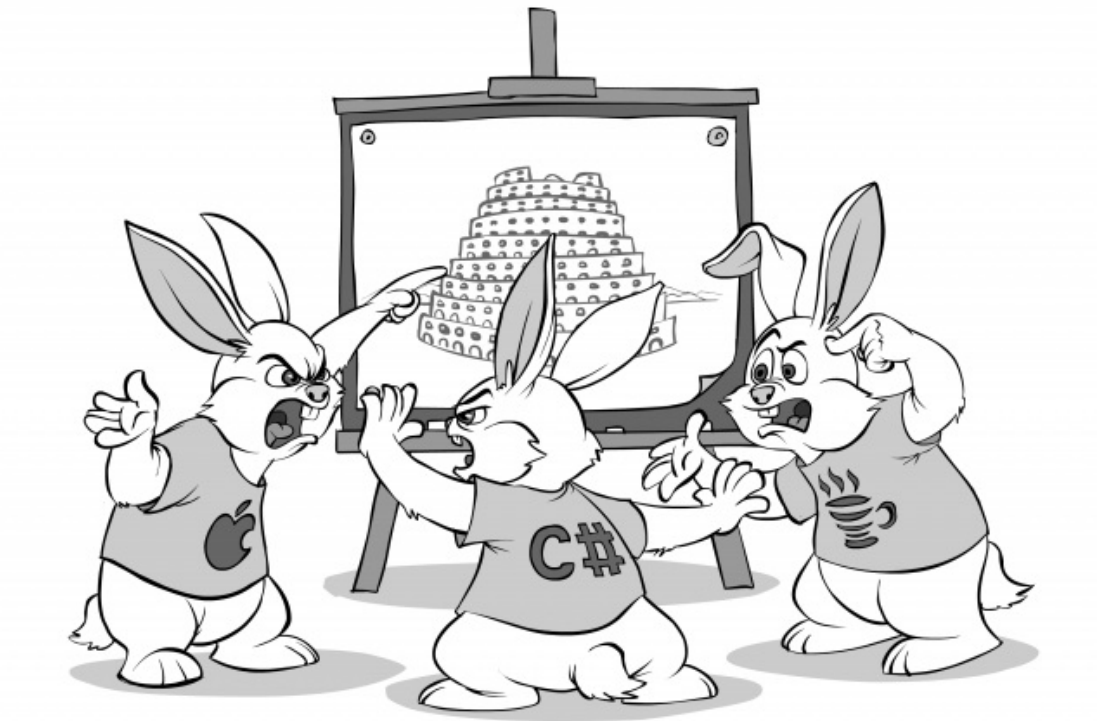


Programming Language for Game Client

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

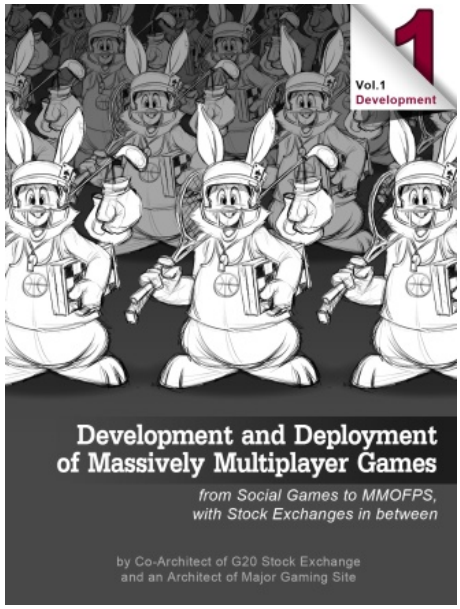
Copyright © 2014-2015 ITHare.com

Chapter V(c). Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines

posted December 7, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

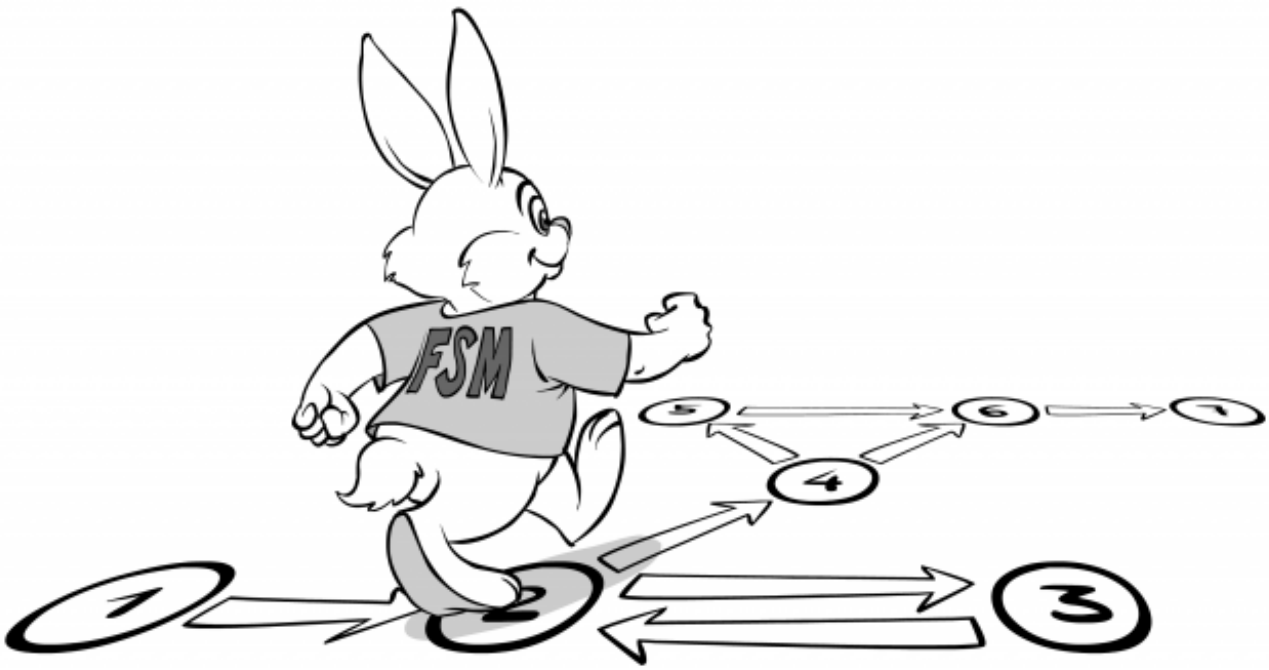
[[*This is Chapter V(c) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further 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.]]



Distributed Systems: Debugging Nightmare

Any MMOG is a distributed system by design (hey, we do need to have a server and at least a few thousands of clients). While distributed systems tend to differ from non-distributed ones in quite a few ways, one aspect of distributed systems is especially annoying. It is related to debugging.



The problem with debugging of distributed systems is that it is usually impossible to predict all the scenarios which can happen in real world. In short, we're speaking about race conditions. While it is usually possible to answer "What will happen if such a packet arrives at exactly such and such moment", making an *exhaustive* list of such questions is unfeasible for any distributed system which is more complicated than a stateless HTTP request-response "Hello, Network". If you didn't try creating such an *exhaustive* list yourself for a non-trivial system – you may want to try doing it, but it will be much cheaper to believe my experience in this field – for any non-trivial stateful system it won't work, period.

This automatically means that even the best possible unit testing (while still being useful) inevitably fails to provide any guarantees for a distributed system. Which in turn means that in many cases you won't be able to see the problem until it happens in simulation testing, or even in real world. To make things even worse, in simulation testing it will happen every time at a different point. And when it happens in real world, you usually won't be able to reproduce it in-house. Sounds grim, right? It certainly does, and for a reason too.

Race condition

A race condition or race hazard is the behavior of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events.

— Wikipedia —

As a result, I am going to make the following quite bold statement:

If you don't design your distributed system for testing and post-mortem analysis,[1] you will find yourself in lots of trouble

Fortunately, it *is* possible to design your system for distributed testing, and it is *not* that difficult, but it requires certain discipline and is better to be done from the very beginning; we'll discuss one of the ways to do it a little bit later.

The Holy Grail of Post Mortem

In practice, whatever amount of testing you do, test cases produced by real life and by your inventive players, will inevitably go far beyond everything you were able to envision in your tests. It means that from time to time your program will fail. While reducing time between failures is very important, another thing is arguably even more important than that: it is the time which takes you to fix the bug after it was reported. And for reducing number of times which the program needs to fail before you can fix it, post-mortem analysis is of paramount importance. The holy grail of post-mortem, of course, is when you can fix any bug using the data from one single crash, so it doesn't affect anybody anymore. This holy grail (as well as any other holy grail) is not achievable in practice. However,

I've seen systems which, using techniques similar to those described in this book, were able to fix around 75% of all the bugs after a single post-mortem

---

[1] here we're speaking about post-mortem analysis after program failure, core dump or otherwise, and *not* about "project post-mortem"

Portability: Platform-Independent Logic as "Nothing But Moving Bits Around"

Now let's set aside all the debugging for a moment, and speak a little bit about platform independent stuff. Yes, I know I am jumping to quite a different subject, but we do need it, you will see how portability is related to debugging, just half a page later.

In most cases graphics, input, and network APIs on different platforms will be different. Even if all your current platforms happen to have the same API for one of the purposes, chances are that your next platform will be different in this regard.

"It is necessary to separate your code into two very-well-defined parts: platform-dependent one and platform-independent one.

As a result, it is necessary to separate your code into two very-well-defined parts: platform-dependent one and platform-independent one. Moreover, similar to what we've discussed with regards to Logic-to-Graphics Layer (see "Logic-to-Graphics Layer" section above), your platform-dependent code needs to be very-rarely-changing, and your frequently-changing game logic needs to be platform-independent.

When speaking about platform-independent logic, a friend and colleague of mine, Dmytro Ivanchykhin, likes to describe it as "nothing more then moving bits around". Actually, this is quite an accurate description. If you can isolate a portion of your program in such a way that it can be described as mere "taking some bunches of bits, processing them, and giving some other bunches of bits back", all of this while making only those external calls which are 100% cross-platform (ok, "100% cross-platform for all the platforms you need"), you've got your logic platform-independent.

Having your game logic on the client side as a platform-independent logic, is absolutely necessary for any kind of cross-platform development. There is no way around it, period; any attempts to have your game logic interspersed with the platform-dependent calls will doom your code sooner rather than later. This is just a common wisdom of cross-platform development, and not really specific to games or distributed systems.

Stronger than Platform-Independent: Strictly-Deterministic

The approach described above, is very well-known and is widely accepted as The Right Way to achieve platform-independence. However, having spent quite a bit of time with debugging of distributed systems, I've became a strong advocate of making your logic part not only platform-independent, but also *strictly-deterministic*. While strictly speaking, one is not a superset of the other one, in practice these two concepts are very closely interrelated.

"The idea here

The idea here is to have your game logic consisting of the functions, which are 100% defined by their-input-data plus by internal-game-logic-state; as we'll see below, strict determinism can be achieved when you call system-dependent functions inside, though it comes with some caveats and should be usually avoided. For most of the well-written code out there, a large part your game logic is already written more

or less around these lines, and there are only a few relatively minor modifications to be made. In fact, modifications can be that minor, that if your code is reasonably well-written and platform-independent, you may even be able to introduce strict determinism as an afterthought. I've done such things before, and it is not that much of a rocket science, but honestly, it is still much better to go for strict determinism from the very beginning, especially as the cost is very limited.

is to have your game logic consisting of the functions, which are 100% defined by their-input-data plus by internal-game-logic-state

Strictly-Deterministic Logic: Benefits

At this point, you should have two very reasonable questions. The first one is "What's in this strict determinism for me?", and the second one is "How to implement it?"

To answer the first question and to explain *why* you should undertake this effort, all the benefits of implementing your logic this way actually result from one single word: determinism. When all the outputs of your logic are completely defined by its internal state plus its inputs, your program module (class, etc.) becomes perfectly deterministic. And while you may think that this is a purely theoretical advantage, determinism provides several very practical benefits. Most of these benefits result from one all-important property of a strictly deterministic system:

if you record all the inputs of a strictly deterministic system, and re-apply these inputs to another instance of the same strictly deterministic system in the same initial state, you will obtain exactly the same results

For practical purposes, let's assume that we have mechanics to write an *inputs-log*, recording all the inputs to our strictly-deteministic logic (see "Implementing Strictly-Deterministic Logic: Definitions" section below for implementation details):

- Your testing becomes deterministic, reproducible, and reversible
  - it means that as soon as you've got a failure, you can repeat the whole thing, and get the failure at exactly the same place in code. Such 100% reproducibility, in particular, allows things such as "let's stop our execution at 5 iterations *before* the failure."[2] If you have ever debugged a distributed program with a difficult-to-reproduce bug, you will understand that one this single item is worth all kinds of trouble.
  - in addition, your testing becomes more meaningful; without 100% determinism, any testing has a chance to fail depending on certain

conditions, and having your tests failing randomly from time to time is the best way I know to start ignoring such sporadic failures (which often indicate race-related and next-to-impossible-to-figure-out bugs). On the other hand, with 100% determinism, each and every test failure means that there is a bug in your code, that cannot be ignored and needs to be fixed (and can be fixed too, improving quality of your production code significantly)

- 100% reproducible bugs during post-mortem, both client-side and server-side
  - if you can log all the inputs to your logic in production (and quite often you can, at least on circular basis, see "EventProcessor Variations: Circular Buffers" section below for details), then after your logic has failed in production, you can "replay" this *inputs-log* on your functionally identical in-house system, and the bug will be reproduced at the very same point where it has originally happened. Even better, your in-house system needs be only functionally identical to production one (i.e. performance is a non-issue, and any PC will do); also you are not required to replay the whole system, you can replay only suspicious module instead. Moreover, during such *inputs-log* replay it is not necessary to run it on the same time scale as it was run in production; it can be run either faster (for example, if there were many delays, and delays can be ignored during replay), or slower (if your test rig is slower than the production one).

"After your logic has failed in production, you can "replay" this *inputs-log* on your functionally identical in-house system, and the bug will be reproduced at the very same point where it has originally happened.

- Regression testing using production data
  - if you've got your *inputs-log* just once, you can "replay" it to make sure that your code changes are still working. In practice, it comes handy in at least two all-important cases:
    - when your new code just adds new functionality, and unless this new functionality is activated, the system should behave exactly as before
    - when your new code is a pure optimization of previous one; when dealing with many thousands of simultaneous users, such optimizations can be Really Complicated (including major rewrites of certain pieces), and having an ability to make sure that new code works *exactly* as the old one (just faster), is extremely important.
- Keeping code bases in sync
  - If you're unlucky enough to have 2 code bases (or even "1.5 code bases", see Chapter [[TODO]] for details), then running the same *inputs-log* over

the two code bases provides an easy way to test whether the code bases are equivalent. Keep in mind that it requires cross-platform determinism, which has some additional issues, discussed in "Cross-Platform Issues" section below.

- User Replay, see discussion in "On User Replay" subsection below.
- Last but not least – determinism may allow you to run exactly the same logic (or even physics) both on client and server, feeding them with the same data and obtaining the same results. This will allow to save A LOT on network traffic (we'll discuss it in more detail in Chapter [[TODO]]); on the other hand, it requires cross-platform determinism across *all* your platforms, which is much more difficult to achieve than a single-platform one (and is more difficult to achieve than cross-platform determinism for two selected platforms).

Keeping in mind that:

- if you have a good development team, any reproducible bug is a dead bug
- the most elusive and by far time-consuming bugs in distributed systems tend to be race-related
- the race-related bugs are very difficult to reproduce, we can easily conclude that

having deterministic testing makes a Really Big Difference when it comes to distributed systems.

With strictly deterministic systems (and appropriate testing framework), all those elusive and next-to-impossible-to-locate race-related bugs are brought to you on a plate.[3]

There are also additional benefits of being deterministic[4], but these are beyond the scope of this book.

---

[2] to be fair, similar things in non-production environments are reportedly possible with GDB reverse debugging; however, it is platform-dependent, and is out of question for production, as running production code in reverse-enabled debug mode is tantamount to a suicide for performance reasons

[3] I don't want to say that you're like Pumba or Timon from Lion King series

[4] examples include, for example, an ability to perform incremental backup just by recording all the inputs (will work if you're careful enough), and an additional ability to apply an existing *inputs-log* to a recently fixed code base; the latter, while being quite esoteric, may even save your bacon in some cases, though admittedly rather exotic ones

Strictly-Deterministic Logic: On User Replay

With traffic being cheap and YouTube videos ubiquitous, User Replay is not that important these days, but still deserves being mentioned. When your game logic is fully deterministic, it will be possible for the player to record the game as it was played, get a very small (!) file with the record, and then share this file with the other players. Which in turn may help to build your community, etc. etc. As mentioned above, it is not *that* attractive these days, but you might still want to think about User Replay (which is coming to you more or less "for free", as you need determinism for other reasons too). If you add some interactive features during replay (such as changing viewing angle and commenting features such as labels attached to some important units, etc.), it might (or might not) have business sense.

A few points about implementing User Replay via deterministic replay:

- Usually, you need to record (and replay) only one entity – the one which is most close to the graphics. In other words, in terms of Generic QnFSM Architecture described below, you need to record/replay only "Animation&Rendering" FSM (and "Game Logic FSM" doesn't need to be recorded to enable User Replay)
- Keep in mind that User Replay will normally require you to adhere to the most stringent version of determinism, including cross-platform issues (see "Cross-Platform Issues" section below)
    - As a bit of relief, you MIGHT (or might not) be able to get away with not-that-strict determinism when it comes to floating-point issues (see "Cross-Platform Issues" section below); however, there is a big open question whether these really subtle differences will accumulate into some kind of macroscopic effects. [5]
- When implementing User Replay as deterministic replay, you'll need to deal with the "version curse". The problem here is that strictly speaking, replay won't run correctly on a different version of FSM 🙁 . So you will need to add FSM version number to all of these files, and then:
    - either to anayse track which version of replay file will run on which of the FSMs. I don't think is realistic (as analysis is too complicated)
    - or to keep *all* the different publicly-released versions of the FSM in the client, so *all of them* are available for replay. This one might fly, because FSM code size is usually fairly small (at most of the order of hundreds of kilobytes), and updates to post-Logic-to-Graphics Layer are relatively rare.
        - even in this case, your Animation&Rendering FSM will have external dependencies (such as DirectX/OpenGL), which can be updated

and cause problems. However, as long as external dependencies are 100% backward-compatible – you should be fine (at least in theory)

    - while adding meshes/textures isn't a problem, replacing them is. For most of the purely cosmetic texture updates, you may be fine with using newer versions of textures on older replays, but for meshes/animations – probably not, so you may need to make them versioned too (ouch!)

- Ultimately, this is still a business decision, so even if you like the idea of User Replay a lot, but business guys say that they don't need this kind of stuff – don't bother with implementing it; while seemingly trivial, it will require quite a bit of time to implement in a way which makes it both replayable and interesting for your players.
    - On the other hand, decision whether you want to have determinism for testing/post-mortem purposes – is *not* a business decision, and you may (and IMNSHO should) do it pretty much regardless of Business Requirements (that is, unless Business Requirements state "crash for each user at least twice a day"); at the very least you should go for determinism for most of the games with Undefined Life Span.

---

[5] personally, I've never faced these things, so I cannot provide any real-world comments

Implementing Strictly-Deterministic Logic: Definitions

I hope I've managed to convince you that strictly-deterministic systems are a Good Thing™, and that now we can proceed to the second question: how to implement these strictly-deterministic systems?

"I hope I've managed to convince you that strictly-deterministic systems are a Good Thing™

First, let's define what we want from our strictly-deterministic system. Practically (and to get all those benefits above) we want to be able to run our code in one of three modes:

- **Normal Mode.** The system is just running, not actively using any of its strictly-deterministic properties
- **Recording Mode.** The system is running exactly as in Normal Mode, but is additionally producing *inputs-log*
- **Replay Mode.** The system is running using only information from *inputs-log* (and no other information), reproducing exact states and processing sequences which have occurred during Recording Mode

Note that Replay Mode doesn't require us to replay the whole system; in fact, we can replay only one part of the whole thing, the one which we suspect to be guilty. If after analysis we find that it was behaving correctly and that we have another suspect – we can replay that suspect from its own *inputs-log* (which hopefully has been written too during the same session which has caused failure).

Implementing Inputs-Log

Implementation-wise, *inputs-log* is usually organized as a sequence of "frames", with each "frame" depending on the type of data being written. Each of the "frames" usually consists of a type, and serialized data depending on type.

Let's discuss how it can/should be done in C++ (other languages are usually simpler, or MUCH simpler), and which caveats need to be avoided. Below are a few hints in this regard:

- don't serialize your data as plain C structures; use serialization library instead.
  - it is often a good idea to use your marshalling library (see Chapter [[TODO]]) for serialization purposes too
- it doesn't really make much difference which serialization format (binary or text-based) you use; it is much more important to encapsulate your serialization format from your state machine logic (i.e. to have serialization library which takes care of all the formatting stuff)
  - as a part of this encapsulation, I strongly suggest to define your own (and opaque!) stream class , using this your-own-and-opaque-class as a parameter to your serialization functions. The only thing which you're allowed to do with an object of this class, is to pass it to a function from your serialization library. This approach will save you quite a lot of trouble down the road.
  - despite exact format being more or less irrelevant, make sure that your serialization format is portable between your plaftorms

"it is much more important to encapsulate your serialization format from your state machine logic

- while we're speaking about *inputs-log,* most of the time your data will be plain and without any pointers; however, in some cases (and when state serialization becomes necessary, see, for example, "EventProcessor Variations: Circular Buffers" below), the need to serialize more complicated data structures may arise.
  - In such cases, usually, you fill find that your data (whether for *inputs-log* frames or for states) is still simple enough to be described by levels 1 to 3 on the sophistication scale as defined by [ISOCPP]. You may want to use

suggestions described there.

Implementing Strictly-Deterministic Logic: Original Non-Strictly-Deteministic Code

Now as we have our *inputs-log,* let's see how we can implement our logic which will record/replay it. Let's start with a simple example: a class, which implements a "double-hit" logic. The idea is that if the same NPC gets hit twice within certain pre-defined time, something nasty happens to him.[6] Usually, such a class would be implemented along the following lines:

```
class DoubleHit {
  private:
  const int THRESHOLD = 5;//in MyTimestamp units
  MyTimestamp last_hit;
    //actual type of MyTimestamp may vary
    // from time_t to microseconds, and is not important for our purposes

  public:
  DoubleHit() {
    last_hit = MYTIMESTAMP_MINUS_INFINITY;
  }

  void hit() {
    MyTimestamp now = my_get_current_time();
    if(now – last_hit < THRESHOLD)
      on_double_hit();

    last_hit = now;
  }

  void on_double_hit() {
    //do something nasty to the NPC
  }
};
```

While this example is intentionally trivial, it does illustrate the key point. Namely, while being trivial, function DoubleHit::hit() it is NOT strictly-deterministic. When we're calling hit(), the result depends not only on input parameters of hit() and on members of DoubleHit class, but also on the *time* when it was called.

---

[6] while such logic normally belongs to server-side in MMOs, it may need to be duplicated on the client-side (for example, due to the client-side prediction, see Chapter [[TODO]] for details), so it is relevant for client-side too

Implementing Strictly-Deterministic Logic: Strictly-Deteministic Code via Intercepting Calls

Now let's see what we can possibly do to make our DoubleHit::hit() deterministic. In general, I know two and a half approaches to achieve it.

"The first approach is to "intercept" all the calls to the function my_get_current_time().

The first approach is to "intercept" all the calls to the function my_get_current_time(). "Intercepting calls" here is meant as changing behaviour of the function depending on the mode in which the code is running, adding/changing some functionality in "Recording" or "Replay" modes. "Intercepting" my_get_current_time() can be done, for example, as follows: whenever the system is running in "recording" mode, my_get_current_time() would run as normal, but would additionally store each returned value to the *inputs-log*. And whenever the system is running in "replay" mode, my_get_current_time() would read the next value from the *inputs-log*, and would return that value regardless of actual time (and without making any system calls). This is possible exactly because of 100% determinism: as all sequences of calls during Replay are exactly the same as they were during Recording, it means that whenever we're calling my_get_current_time(), at the "current position" within our *inputs-log* we will always have exactly a record which was made by my_get_current_time() during Recording.

Therefore, "interception" of the function my_get_current_time() may be implemented, for example, as follows:

```
1  MyTimestamp my_get_current_time() {
2    if(Mode == REPLAY_MODE) {
3      MyTimestamp ret =
4        //read next frame from global inputs-log into 'ret'
5        // this frame MUST be a my_get_current_time() frame
6      ;
7      return ret;
8    }
9
10   MyTimestamp ret =
11     // code for my_get_current_time() before "call interception"
12     ;
13
14   if(Mode == RECORDING_MODE) {
15     //write my_get_current_time() as a 'frame'
16     //  to a global inputs-log
17   }
18
19   return ret;
20 }
```

Bingo! This approach would make our implementation strictly-deterministic, and without any code changes too! Actually, this is pretty much what [liblog] replay tool[7]

did.

However, there is a significant caveat with this way of making your logic strictly deterministic. If we add (or remove) any calls to my_get_current_time() (or more generally, to any of the functions-which-record-to-inputs-log), the replay will fall apart. While replay will still work for exactly the same code base, things such as replay-based regression testing will become pretty much unusable in practice, and existing real-world *inputs-logs* (which are an important asset, helping to test things) will be invalidated too frequently.

---

[7] not to be confused with other tools with the same name; as of now, I wasn't able to find an available implementation of liblog 🙁

Implementing Strictly-Deterministic Logic: "Pure Logic"

An alternative way (the one which I usually prefer) of making the class strictly-deterministic, is to change the class DoubleHit itself so that it becomes strictly deterministic without any interception trickery. For example, we could change our DoubleHit::hit() function to the following:

```
1  void hit(MyTimestamp now) {
2    if(now – last_hit < THRESHOLD)
3      on_double_hit();
4    last_hit = now;
5  }
```

If we change our class DoubleHit in this manner, it becomes strictly deterministic without any need to "intercept" any calls; let's name such classes "pure logic" classes.

In general, whenever there is a choice, I usually prefer this "Pure Logic" approach; it is more explicit than intercepting calls, more easily readable, and has better resilience to modifications. However, it has some implications to keep in mind:

- with "pure logic", it becomes a responsibility of the caller to provide stuff such as timestamps
  - this passing parameters may (and usually will) go through multiple levels of calls
  - at some level, however, some caller-of-caller-... needs to call my_get_current_time() and pass obtained value as parameter
- it is a responsibility of whoever-calls-my_get_current_time(), to record data to *inputs-log* (and to handle replay too). See class EventProcessor below for an example.

- the whole chunk of processing (while caller-which-has-called-my_get_current_time() passes parameter around) is deemed to happen at the same point in time. While this is exactly what is desired for 99.9% of game logic, you need to be careful not to miss remaining 0.1%.

Implementing Strictly-Deterministic Logic: TLS-based Compromise

As an alternative to passing parameters around, you might opt to pass parameters via TLS instead of stack.The idea is to store MyTimestamp (alongside with any other parameters of similar nature) to the TLS, and then whenever my_current_get_time() is called, merely read the value from TLS.

> **TLS**
> **Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.**
> — *Wikipedia* —

In practice, it means doing the following:

- keep your original logic code intact, with my_get_current_time() calls within
- rename my_get_current_time() to my_real_get_current_time()
- at those points where you'd call my_get_current_time() (for passing result as a parameter) in "pure logic" model, call my_real_get_current_time() and write the result to TLS[8]
- implement my_get_current_time() as simply reading of the value from TLS

This model is a kind of compromise between the two approaches above; it is less verbose (and less explicit) then "pure logic", but it is functionally equivalent to "pure logic", and therefore it doesn't suffer when somebody inserts yet another my_get_current_time(). If you prefer this model to "Pure Logic" – it is fine, but you'll need to figure out fine details of TLS yourself, as I will describe things mostly in terms of "Pure Logic" (though it can be converted to TLS-based Compromise Model in a very straightforward manner).

In TLS-based Compromise Model, handling of the recording/replay is exactly the same as in "pure logic" model (see also class EventProcessor below); the only thing which TLS-based Compromise changes compared to the "pure logic", is how the data is passed from caller to callee; everything else (including the data written to the *inputs-log*) is exactly the same.

---

[8] for C++, see C++11's *thread_local* storage duration specifier, but there are usually other platform-dependent alternatives

Implementing Strictly-Deterministic Logic: Passing Input Parameters as Data Members

Yet another way to handle it is to put all the input parameters as data members of your class DoubleHit:

```
1  class DoubleHit {
2   private:
3   const int THRESHOLD = 5;
4   MyTimestamp last_hit;
5   MyTimestamp now;//NOT recommended because of confusion!
6
7   public:
8   //...
9   void hit() {
10   if(now – last_hit < THRESHOLD)
11     on_double_hit();
12
13   last_hit = now;
14  }
15  //...
```

While it again is functionally equivalent to “Pure Logic”, and will work, I shall say that I don’t like it on readability grounds. Perceptionally, members of class DoubleHit clearly represent “current state” of class DoubleHit, and putting *now* (which is an “input parameter”, and is semantically very different from “current state”) there will be too confusing as soon as you give the code to somebody not familiar with your conventions.

Implementing Strictly-Deterministic Logic: Which Model to Choose?

Personally, I usually prefer “Pure Logic” approach described above. However, I admit that “TLS-based Compromise” is functionally equivalent to “Pure Logic” one, and that a discussion about being explicit vs being brief in this case is pretty much about personal preferences.

Therefore, I think that any of these two models is fine, just stay away from “intercepting calls” and “passing parameters as data members”; actually, even “intercepting calls” and “passing parameters as data members” are light years ahead of having no strict determinism at all, but why settling for something worse when you can have something better at the same price?

Implementing Strictly-Deterministic Logic: Which system functions we’re speaking about?

In general,

each and every of system calls (including system calls made indirectly via wrappers), creates a danger of deviating your class from being strictly-deterministic.

As a result, some of the readers will say: "hey, this way we will end up with millions of parameters (or function calls we need to intercept)!". Fortunately, it is not that bad.[9] Let's take a closer look at the question "what exactly do we need to intercept/provide?"

"Let's take a closer look at the question "what exactly do we need to intercept / provide?"

As we've already discussed in "Logic-to-Graphics Layer" section above, all the code which works directly with graphics, should be separated from game logic by Logic-to-Graphics Layer; moreover, the interface which we've defined for this Layer, was essentially one-way communication (with game logic sending instructions to draw something, to the Layer), and one-way communication which goes in "from logic" direction, doesn't affect determinism. In a similar manner, all the code which directly calls network sockets or input, should be moved to the platform-dependent part; moreover, these parts will usually reside in a different thread, or at least "higher" than our game logic (so that will act as callers with regards to game logic), see section [[TODO]] below; such usage won't affect determinism either.

This leaves us with two major items which are closely related to the logic, and are not deterministic per se; these are time and client-side configuration.[10] Let's take a look at each of these two items:

- Time. Time as such is not deterministic by design, but obtaining time is generally cheap, so usually there is no problem for the caller to pass time to the callee, even if callee won't use it. Therefore, for time we can use "pure logic" approach above, to make the things more explicit (and to avoid problems when existing input-logs get incorrect when somebody inserts another my_get_current_time() into processing logic); functionally equivalent "TLS Compromise" will be less explicit and less verbose too.
- Client-side Configuration. Client-side configuration is pretty much the only case when you may need an access to the client-side file system (leaving aside caching, see on it below). With regards to the client-side configuration, you usually can set it to one fixed value for the whole session, and to put this one fixed value into the very beginning of your input-logs. If you want to test *manipulating* client-side configuration (which I never needed and never heard of somebody who needed it, but in the game world anything can happen) – you may choose either to intercept calls, or to use a kind of exception-based

trickery which will be described in Chapter [[TODO]] with regards to conditional handling of real (hardware-based) randomicity.

- Other stuff. There is a chance that you will genuinely need to use a non-deterministic call which is not covered in this Chapter, and to do it from within your game logic. One such example includes re-formatting of the server time into local time (which is better to be avoided anyway to avoid confusion, replacing it with system-independent "it happens in 37 minutes from now" kind of stuff, but sometimes you just don't have a choice). In such cases, you have the same two choices as right above – either to intercept relevant calls[11], or to use exception-based approach described in Chapter [[TODO]]. As long as such calls are rare, both these approaches will work reasonably well in practice.

---

[9] it took me quite a few years to realize, how actually *good* it is
[10] for server-side, there is also real (hardware-based) randomicity and databases, but we'll set this discussion aside until Chapter [[TODO]]
[11] it is MUCH better to do this interception after conversion to system-independent data formats, so it is system-independent data which gets into *inputs-log*

Strictly-Deterministic Logic: Non-Issues

In addition to the non-deterministic issues described above, there are also three non-issues; these are pseudo-random numbers, logging and caching.

Pseudo-random numbers as such are perfectly deterministic; that is, as long as you're storing PRNG state as a part of your logical state (if you're using FiniteStateMachine as described below – as a member of specific class MyFiniteStateMachine). Instead of using non-deterministic rand() (which implicitly uses a global, see also below), you can either implement your own linear congruential PRNG (which is literally a one-liner, but is not really good when it comes to randomicity, see also Chapter [[TODO]]), or use one of those Mersenne Twister classes which are dime a dozen these days (just make sure that those PRNG classes have PRNG state as a class member, not as a global); for C++ you can use something like boost::mt19937. Note that to get your PRNG seeded, you still need to provide some seed which is external to your deterministic logic, but this is rarely a problem.

Logging/tracing (as in "log something to a text/binary log file"), while it does interact with an outside world, is usually strictly deterministic per se. Moreover, even if your logging procedure prints current time itself (and to do it, calls my_get_current_time() or something else of the sort), and technically becomes non-deterministic from the "all the world outside of our logic" point of view (this happens because it's end-result depends on the current time), it still stays strictly

deterministic from the point of view of the logic itself (as long as the logic cannot read the log).[12] Practical consequence: even in "Pure Logic" model, there is no need to pass 'now' parameter to those framework-level functions which implement logging (even if they call my_get_current_time() inside, but only as long as the result of this my_get_current_time() is not used other than to write data to the log).

The second deterministic non-issue is related to caching. Caching (whether file-based or memory-based), when it comes to the determinism, is an interesting beast. As long as all your caching does, is strictly caching of the data and nothing else, it is deterministic, regardless of all the reads and writes (provided that original state of the cache is stored, if applicable, in *inputs-logs*). While relying on caching being implemented as a correct cache won't allow you to "replay-test" caching itself, as long as you're sure that your caching is working – you can rely on it's determinism.[13]

"The second deterministic non-issue is related to caching.

More generally, such things as logging and caching (if they are strictly deterministic themselves), can be considered "outside" of our logic (more strictly – outside of "isolation perimeter" as defined in "Event-Driven Programming and Finite State Machines" section below); this approach greatly reduces amount of logging which is required to guarantee correct recording/replay, at the cost of the recording/replay being unable to aid with testing of your logging/caching routines. In practice, as logging/caching are relatively simple and are rarely changed, the latter restriction doesn't cause too much trouble.

---

[12] I know that this explanation reads quite ugly, but I cannot find better wording now; regardless of the wording, the statements in this paragraph stand

[13] strict proof is beyond the scope of this book

Strictly-Deterministic Logic: No Access to Globals

This might go without saying, but let's make it explicit:

for your logic to be strictly-deterministic, you MUST NOT use any global variables. Yes, it means "No Singletons" too.

Actually, it is not just a requirement to be strictly-deterministic, but is a well-known "best practice" for your code to be reasonably reliable and readable, so please don't take it as an additional burden which you're doing just to become

strictly-deterministic; following this practice will make your code better in the medium- and long-run even if you're not using any of the benefits provided by strict determinism.

The only exception to this rule is that accessing constants is allowed without restrictions (well, as long they you don't modify them 😉).

As an consequence,

you SHOULD NOT use any function which implicitly uses globals

Identifying such functions can be not too trivial, but if you need to stay strictly deterministic, there it is a requirement to avoid them. Alternatively, you may decide to "intercept" these calls (and write whatever-they-return into *inputs-log*) to keep your logic strictly deterministic, but as noted above, "intercepting calls" is better to be avoided when feasible.

C standard library is particularly guilty of providing functions which implicitly access globals (this includes *rand()*). Most of these functions (such as strtok()) should be avoided anyway due to the logic being non-obvious and being potentially thread-unsafe on some of the platforms. One list of such functions can be found in [ARM]; note that the problem here is not about thread-safety, so rand() and strtok() are still non-deterministic even on those platforms (notably Windows) which make them thread-safe by replacing globals with TLS-based stuff.

Strictly-Deterministic Logic: Pointers

C/C++ pointers are quite a nasty beast in general, and can cause quite a few problems when it comes to determinism too 🙁 . The problem with pointers from determinism point of view is that in general, you cannot guarantee that allocated pointers are the same on different runs of the program. [14] As a result, below is a list of things which should be avoided when writing for determinism:

- using convoluted pointer arithmetic (and "convoluted" here means "anything beyond simple array indexing). Seriously, if you're relying on this kind of stuff, you'd better write for Obfuscated C contest and stay away from any serious development.
- sending pointers over the network (and writing them to *inputs-log*), regardless of marshalling used. Actually, this one should be avoided regardless of determinism.
- using pointers as identifiers
- using pointers for ordering purposes; even using pointers to get "just some

kind of temporary ordering" is not good for determinism, sorry about that

While this looks as quite a few items to remember about, it is not too bad in practice.

Strictly-Deterministic Logic: Cross-Platform Issues

Achieving strict determinism on one single platform is significantly easier than across different platforms. For many practical purposes (such as post-mortem and debugging), it is sufficient to have strict determinism only within one single platform. However, to obtain some other properties (for example, cross-platform-equivalence testing, User Replay, and identically running physics engines) you may need to have cross-platform determinism. In such a case, additional considerations apply:

- non-ordered and partially-ordered collections may produce different results on different platforms while staying compliant. For C++, examples include hash-table-based unordered_map<>/unordered_set<> containers, and tree-based partially ordered multiset<>/multimap<> containers.
    - a funny thing about them is that they ARE indeed nothing more than "moving bits around", it is just that bits are moved in a bit different (but compliant) manner for different implementations
    - it means that one way to deal with them, is to write your own version (or just to compile *any* of existing ones to *all* the platforms); as long as the code for all the platforms is (substantially) the same, it will compile into the code which behaves exactly the same
    - for tree-based partially ordered sets/maps, you often can make them fully-ordered by adding an artificial ID (for example, incremented for each insert to the container) and using it as a tie-breaker if original comparison returns that objects are equal. It is quite a nasty hack, but if you don't need to care about ID wraparounds (which is almost universally the case if you're using 64-bit IDs), and you don't care about storing an extra ID for each item in collection, it works.
- floating-point arithmetic issues. In short: while floating-point will return *almost* the same results on different platforms, making them *exactly* the same across different hardware/compilers/... is very challenging at the very least. For further details, refer to [RandomASCII2013] and [Fiedler2015]. A few minor but important points *in addition to the discussion in those references*:
    - As floating-point arithmetic is once again all about "moving bits around" (it just takes some bunches of bits and returns some other bunches of bits), it can be made perfectly deterministic. In practice, you can achieve it by using software floating-point library which simulates floating-point via integer arithmetic [[TODO: add ref to Knuth]]
        - Note that such a library (if used consistently for *all* your platforms)

does *not* need to be IEEE compliant; all you need is just to get some reasonable results, and last bit of mantissa really matters in practice (as long as it is the same for all the platforms)

- such libraries are slooooow; for a reasonably good floating-point emulation library (such as [SoftFloat]) you can expect slowdown of the order of 20-50x compared to hardware floating point ☹ .
  - however, certain speed up can be expected if the library is rewritten to avoid packing/unpacking floats (i.e that class MyFloat is actually a two-field struct), and replacing IEEE-compliant rounding with some-reasonable-and-convenient-to-implement rounding; very wild guesstimate for such an improvement is of the order of 2x [JohnHauser], which is not bad, but will still leave us at least with 10x slow-down compared to hardware floating point ☹ .
- however, if you're fine with this 20-50x-slower floating-point arithmetic (for example, because your logic performs relatively few floating-point operations) – such libraries will provide you with perfect determinism.

Strictly-Deterministic Logic: Implementation summary

Given analysis above, we've found that while there are tons of places where your logic can potentially call external functions (and get something from them, making the logic potentially non-deterministic), in practice all of them can be dealt with easily, and in most cases the only thing you'll need to pass around, will be "current timestamp".

All the other system function calls will fall under one of the following:

- function calls which are output-only (drawing, logging, generating output events)
- function calls which shouldn't be called from within the logic (communications, user input)
- function calls which are used to implement caching; as long as caching is working correctly, they can be ignored for the purposes of determinism (see explanation above)

Even if you need more than "current timestamp", nothing prevents you from making a struct, consisting of all-of-your-pre-calculated-input-parameters, and passing around one single parameter (*FSMInputData* input_data* or something).

From my experience, this single extra parameter is not a large price for all the benefits you will get from making your logic strictly deterministic (and if you have strong feelings about this extra parameter, you can avoid it by using TLS Compromise).

"This single extra parameter is not a large price for all the benefits you will get from making your logic strictly deterministic (and if you have strong feelings about this extra parameter, you can avoid it by using TLS Compromise)

As for other issues (those not related to external function calls), they are also of only very limited nature until you're going for a full-scale cross-platform determinism; neither avoiding globals (which is a good practice anyway), nor avoiding pointer-related trickery tends to cause much practical problems.

However, if you're going into realm of cross-platform determinism, things may get quite a bit nastier (and will cause more trouble); while collection differences can be handled if you're careful enough, achieving fully cross-platform floating point calculations can be trouble across different CPUs.

Strictly-Deterministic Logic: Overall summary

TL;DR of the "Stronger than Platform-Independent: Strictly-Deterministic" section:

- Strictly-deterministic logic is a Good Thing™, providing game-changing benefits for debugging distributed systems, including production post-mortem
- When implementing strictly-deterministic logic, either "Pure Logic" approach, or "TLS-based Compromise" is generally preferred
- Implementing strictly-deterministic logic requires rather few changes in addition to following existing best practices, as long as cross-platform determinism is not required
- Going for a full-scale cross-platform determinism can be tricky, especially because of floating-point issues.

Event-Driven Programming and Finite State Machines

when they don't know what to say
and have completely given up on the play
just like a finger they lift the machine
and the spectators are satisfied
— Antiphanes, IV century B.C. —

So, we've got our one single DoubleHit class as a strictly-deterministic logic. Good for us, but in any realistic system there will be much more classes than this. What should we do about it?

Pieces of strictly-deterministic logic can be combined with each other easily; the only two things to keep in mind are the following:

- don't mix strictly-deterministic code with any non-strictly-deterministic code; such a mix will be non-strictly-deterministic, and you will lose all the benefits arising from being strictly-deterministic
- if you're using "Pure Logic" model to achieve determinism, you're not allowed to call my_get_current_timestamp() within your logic. It implies that whenever you need to pass the MyTimestamp parameter to the callee, you yourself should get it from the caller.

The latter observation leads us to a reasonable question: well, somebody will need to call my_get_current_timestamp(), so where this whole calling tree (the one which passes 'now' around) should end? Let's see how it can (and should) be organized.

"We need to make an "isolation perimeter" where we control and log all the inputs of this piece of code.

First of all, let's note that to take advantage of determinism of a certain piece of code, we need to isolate it and make sure that we can control (and log to inputs-log) *all* the inputs of this piece of code. In other words, we need to make an "isolation perimeter" where we control and log all the inputs of this piece of code. Now let's see how we want to build this "isolation perimeter". Systems such as [liblog] are trying to build this "isolation perimeter" around the whole app; actually, without access to internals of the code it is next to impossible for them to do anything else. On the other hand, we do have access to internals of our own code, and we can build our "isolation perimeter" pretty much anywhere. Let's discuss one approach which has been observed to produce very good results in practice.

Let's say that we have a high-level class EventProcessor, which does nothing but processes incoming events (anything can be an event, from the user input up to passing a certain amount of time, with message-from-server in between). At the point of receiving the event, EventProcessor calls my_get_current_time(), and then calls FiniteStateMachineBase::process_event():[15]

```
class FiniteStateMachineBase {
 public:
 virtual void process_event(MyTimestamp now, MyEvent& ev) = 0;
};

class EventProcessor {
 private:
 FiniteStateMachineBase* fsm;
 int mode;
 InputsLogForWriting& ol;

 public:
 void process_event(MyEvent& ev) {
  MyTimestamp now = my_get_current_timestamp();
  if(mode == RECORDING_MODE) {
   //write both ev and now to ol
  }
  try {
   fsm->process_event(now,ev);
  } catch(exception& x) {
   //some error handling
  }
 }

 void replay_event(InputsLogForReading& il) {
  //parse inputs-log and call fsm->process_event() accordingly
 }
};
```

In "Recording" mode, EventProcessor::process_event() will additionally write both *now* and *ev* to inputs-log. In addition, while strictly not required to ensure determinism, usually at least some of the output-only function calls (such as events-generated-for-other-FSMs, instructions to Logic-to-Graphics Layer, etc.) are also written to the same inputs-log, to simplify automated testing and debugging.

In "Replay" Mode, EventProcessor::process_event() is not called at all; instead, EventProcessor::replay_event() is called, which parses an entry in the inputs-log and calls fsm->process_event() accordingly; in addition, EventProcessor::replay_event() may parse expected outputs from the inputs-log and compare them with the calls which actually happened, raising an exception at the first sign of inconsistency.

Both class FiniteStateMachineBase and class EventProcessor mentioned above are merely providing a framework to implement your own FiniteStateMachine to implement some kind of specific logic:

```
class MyFiniteStateMachine : public class FiniteStateMachineBase {
 private:
 //FSM state goes here
 public:
 virtual void process_event(MyTimestamp now, MyEvent& ev) override {
  //within, the function MAY generate output,
  // including sending events intended for other state machines (!)
  // ...
 }
};
```

It is these classes-derived-from-FiniteStateMachineBase which contains actual FSM state (as data members) and actual FSM logic (as process_event() function).

Relation to Finite Automata as taught in Uni

– Have it compose a poem – a poem about a haircut! But lofty, noble, tragic, timeless, full of love, treachery, retribution, quiet heroism in the face of certain doom! Six lines, cleverly rhymed, and every word beginning with the letter s!! - And why not throw in a full exposition of the general theory of nonlinear automata while you're at it?
— Dialogue between Klapaucius and Trurl from The Cyberiad by Stanislaw Lem —

*NB: if you're not interested in theory, you can safely skip this subsection; for practical purposes it suffices to know that whatever event-driven program you've already written, is in fact a finite automaton, so there is absolutely no need to be scared. On the other hand, if you **are** interested in theory, you'll certainly need much more than this subsection. The idea here is just to provide some kind of "bridge" between your uni courses and practical use of finite automata in programming (which unfortunately differ significantly from quite a few courses out there).*

First of all we need to note that our class FiniteStateMachine, strictly falls under definition of Finite Automaton (or more precisely – Deterministic Finite Automaton) given in Wikipedia (and in quite a few uni courses). Namely, deterministic Finite State Machine (a.k.a. Deterministic Finite Automaton) is usually defined as follows [Wiki.DeterministicFiniteAutomaton]:

- Σ is the input alphabet (a finite, non-empty set of symbols).
  - in our FiniteStateMachine, Σ is a set of values which a pair (now,ev) can take; while this set is exponentially huge, it is still obviously finite
- S is a finite, non-empty set of states.
  - in our case, it is represented by all possible combinations of all the bits forming data members of FiniteStateMachine. Again, it is exponentially huge, but certainly still finite.
- s_0 is an initial state, an element of S.
  - whatever state results from FiniteStateMachine::FiniteStateMachine()

- δ is the state-transition function. δ: S × Σ -> S
  - implemented as FiniteStateMachine::process_event();
- F is the set of final states, a (possibly empty) subset of S.
  - for our FiniteStateMachine, F is always empty.

Therefore, our class FiniteStateMachine complies with this defintion, and *is* a Deterministic Finite Automaton (and most of event-processing systems are Finite Automatons, albeit usually non-deterministic ones).

Quite often,[16] in university courses state-transition function δ is replaced with a "set of transitions". From formal point of view, these two definitions are strictly equivalent, because:

- for any state-transition function δ with a finite number of possible inputs, we can run this function through all the possible inputs, and to obtain the equivalent set of transitions.[17]
- having a set of transitions, we can easily define our state-transition function δ via this set

"The problem which kills this neat idea, is known as "state explosion", and is all about exponential growth of states as you increase complexity of your machine.

On the other hand, if you start to define your state machine via set of transitions *in practice (and not just in theory),* most likely you're starting a journey on a long and painful way on shooting yourself in the foot. When used in practice, this "set of transitions" is usually implemented as some kind of a state transition table (see [Wiki.StateTransitionTable]). It all looks very neat, and fairly obvious. There is only one problem with table-driven finite state machines, and the problem is that they don't work in the real world. The problem which kills this neat idea, is known as "state explosion", and is all about exponential growth of states as you increase complexity of your machine. I won't delve into too much details on the "state explosion", but will note that it quickly becomes really really bad as soon as you're starting to develop something realistic; even having 5 different 1-bit fields within your state lead to a state transition table of size 32, and adding anything else is already difficult; make it 8 1-bit fields and corresponding 256 already existing transitions, and adding any further logic has already became unmanageable.[18] In fact, while I've seen several attempts to define state machines via state transition tables, none of them was able to come even somewhat-close to succeeding.

What is normally used in practice, is an automaton which is defined via state-transition function δ (which function δ is implemented as a usual function in an imperative programming language, see, for example,

FiniteStateMachine::process_event() above). Actually, such automatons are used much more frequently than developers realize that they're writing a finite automaton 😉 . To distinguish these real-world state machines from table-driven (but usually impractical) finite state machines, I like the term "ad-hoc state machines" (to the best of my knowledge, the term was coined in [Calderone2013]).

---

[14] for some of the platforms, *and* when you have your whole program recorded/replayed, you may get such guarantees, but as we're aiming to record/replay on smaller-than-whole-program basis, it won't fly for us
[15] yes, it could have be done with less code, but I certainly prefer to be extremely explicit here
[16] see, for example, [CSC173]
[17] Never mind that such enumeration may easily take much longer than the universe ends from something such as Heat Death or Big Rip – in maths world we don't need to restrict ourselves with such silly notions.
[18] while hierarchical state machines may mitigate this problem a bit, in practice they become too intertwined if you're trying to keep your state machines small enough to be table-driven. In other words: while hierarchical state machines are a good idea in general, even they won't be able to allow you to use table-driven stuff

Implementing Deterministic Finite State Machines

Back to the real world, we need to discuss how to implement deterministic finite state machines. In general, while you're staying within your FiniteStateMachineBase interface and restrictions stated above, exact implementation is up to you, and may easily be different for the different state machines you have. Popular possibilities include:

- any deterministic event-driven program (which is inherently a deterministic ad-hoc state machine); this is probably what you really want to do if it is your first experience with the state machines. While it may (or may not) result in the code which is unwieldy, it is a very familiar pattern, and (if you're making it deterministic as discussed above), you will still benefit from all the goodies mentioned in "Strictly-Deterministic Logic: Benefits" section (such as greatly improved debug and post-mortem).
- in many cases, it is useful to have a separate data member called *state,* which takes one of (mutually exclusive) enumerated values. One good example for *state* variable is your PC *running*, or *walking*, or *jumping*, or *croaching*; another good example is *state* reflecting stage of the specific quest. In any case, you're allowed to have other

"You can implement your Finite State Machine as a deterministic variation of a usual event-driven program

data members in addition to *state* (they represent so-called "extended state" in terms of UML state machines)
    - note that in quite a few cases, having *state* member is considered a requirement to be named a "Finite State Machine"; I hate arguing about which terminology is "right", so I will just note that we're using the definition of "Finite State Machine" taken from Wikipedia (see also above), and according to that definition, *state* member is not strictly required (though often convenient).
    - Finite State Machines with *state* member can be implemented in an ad-hoc manner (basically with a *switch* on your *state* in quite a few places); this is simple and is known to work
    - alternatively, you may want to use State pattern from [GameProgrammingPatterns.StatePattern]; the same book also gives some hints on implementing hierarchical state machines and push-down automata.
    - If you have this *state* member, you may want to document a diagram of your state machine using UML state diagrams [Wiki.UMLStateMachine]; they have some useful concepts too. note that I mean using it only for documentation purposes (and not for code generation), so it doesn't really what kind of software you're using for drawing[19]
- As a Big Fat rule of thumb, you SHOULD NOT try table-driven state machines (those which you might have been taught in uni); see "Relation to Finite Automata as taught in Uni" section above if you need justification.

---

[19] for most of the commercial games, you will have a requirement to keep such things private, so double-check your policy before using something like draw.io; something like Visio will usually be ok

EventProcessor Variations: Circular Buffers

The implementation of EventProcessor described above, is certainly not the only possible one. In fact, the beauty (and practical implications) of the separation between EventProcessor and FiniteStateMachine is that we have a liberty to plug our FiniteStateMachine into pretty much any EventProcessor we want.

One practical case for a different EventProcessor arises when we want to have a "post-mortem log" (sufficient to identify the problem), but we don't want to write all the things "forever and ever", as it might cause performance degradation.

Ok, for such cases we can make a different EventProcessor, let's name it EventProcessorWithCircularBuffer. For this EventProcessorWithCircularBuffer, we can implement *inputs-log* as an in-memory circular buffer, avoiding the need to keep the data forever-and-ever. However, for this to work, it will additionally need to:

"We can implement *inputs-log* as an in-memory circular buffer, avoiding the need to keep the data forever-and-ever.

- make sure that underlying FiniteStateMachine has an additional function such as void serializeStateToLog(InputsLogForWriting& ol), and a counterpart function deserializeStateFromLog(InputsLogForReading& il). State serialization should be implemented in a manner which is consistent with serialization used for *inputs-log* in general; see "Implementing Inputs-Log" section above for further discussion on state serialization.
- call this serializeStateToLog() function so that in-memory circular buffer always has at least one instance of serialized state
- make sure that there is always a way to find serialized state even after a circular buffer wrap-around (this can be done by designing format of your inputs.log carefully)
- on failure, just dump the whole in-memory *inputs.log* to disk
- on start of "Replay", find the serialized state in inputs-log, call deserializeStateFromLog() from that serialized state, and proceed with rollforward as usual.

EventProcessorWithCircularBuffer describes only one of multiple possible implementations of EventProcessor; it has an advantage that all the logging can be kept in-memory and therefore very cheap, but in case of trouble this in-memory log can be dumped, usually providing sufficient information about those all-important "last seconds before the crash". Further implementation details (such as "whether implement buffer as a memory-mapped file" and/or "whether the buffer should be kept in a separate process to make the buffer corruption less likely in case of memory corruption in the process being logged") are entirely up to you 🙂 .

One very important usage of EventProcessorWithCircularBuffer is that in many cases it allows to keep the logging running all the time in production, both on client side and on server side. It means near-perfect post-mortem analysis in case of problems

Let's make some very rough estimates. Typical game client receives around a few hundred bytes per second; let's take it at 200 bytes/sec. User input is very rarely more than that. It means that we're speaking about at most 500-1000 bytes/second. 1MByte RAM buffer is nothing for client-side these days, and at a rate of 1000 bytes/second we'll be able to store about 3 hours of "last breath data" for our "game logic" FSM; these 3 hours of data is usually orders of magnitude more than enough to find a logical bug. For an FSM implementing your animation/rendering engine, calculations will be different, but taking into account that all the game resources are well-known and don't need to be recorded, we again can keep the data recorded to the minimum, again enabling a very good post-mortem.

For the server side, you will need much more memory to run recording, and you might not be able to keep circular buffers running all the time, but at the very least you should be able to run them on selected FSMs (those are currently under suspicion, or those which are not-so-time-critical, or just a random sample).

Deterministic Finite State Machines: Nothing too New But…

While there is nothing really new in event-driven programming (and ad-hoc finite state machines used for this purpose), our finite state machines have one significant advantage compared to those usually used in the industry. Our state machines are strictly-deterministic (at least when it comes to one single platform), that allows for lots of improvements for debugging of distributed systems (mostly due to "replay testing/debugging" and "production post-mortem").

On the other hand, in academy Deterministic Finite Automata are well-known, but I don't know of the descriptions on "how to write them in practical applications".

On the third hand 😉 , determinism for games has been a popular topic for a while (see, for example, [Gamasutra2001]) , and in recent years has got a new life with MMOs and synchronous physics simulation on client and server (see, for example, [Fiedler2015-2])).

On the fourth hand (yes, we're exactly half way on becoming an octopus), I didn't see anybody concentrating on using determinism for the purposes of debug and production post-mortem, and from my experience effect of these items on the quality of your game cannot be underestimated. If you want to have your game crashing 10x less frequently than competition – do yourself and your players a favour, and record production *inputs-logs* for post-mortem purposes, as well as perform replay-based testing. I know I sound like a commercial, but as a gamer myself I do have a very legitimate interest in making games crash much more rarely than they do it now; I also know that for most of good game developers out there, deterministic testing and post-mortem will help in this regard, and will help a lot (in addition to any replay/synchronous-physics goodies if you need them).

Deterministic Finite State Machines: Summary

To summarize all this long talk about determinism and state machines:

- for distributed systems, you DO need to have *at least* single-platform determinism. It will help A LOT with testing, debugging, and production post-mortem.
  - achieving this one is not too difficult, and is usually only a minor annoyance
  - on the other hand, it still provides A LOT of useful stuff, mostly for the purposes of bugfixing (including those elusive production-only bugs)
- for other purposes (cross-platform code equivalence testing, User Replay, and physics equivalence) you MAY need cross-platform determinism
  - achieving this one can be a challenge, especially in the field of floating-point calculations
- nothing prevents you from starting small, with single-platform determinism, and then trying to extend it to cross-platform one. Unless the life of your game depends on a cross-platform determinism, this might be a viable option to pursue.
- Finite State Machines are a nice and convenient way to express deterministic (sub)systems
  - this includes (but is not limited to) ad-hoc Finite State Machines, which are nothing more than very-well known event-based systems

[[To Be Continued…

This concludes beta Chapter V(c) 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(d), "Modular Architecture: Client-Side Overall Architecture.]]

[–] References

[ISOCPP] Standard C++ Foundation, "How do I select the best serialization technique?"
[liblog] Dennis Geels, Gautam Altekar, Scott Shenker, Ion Stoica, "Replay Debugging for Distributed Applications"
[ARM] "C library functions that are not thread-safe", ARM Compiler toolchain ARM C and C++ Libraries and Floating-Point Support Reference
[RandomASCII2014] Bruce Dawson, "Floating-Point Determinism", 2013
[Fiedler2015] Glenn Fiedler, "Floating Point Determinism", Gaffer on Games, 2015
[SoftFloat] "Berkeley SoftFloat"
[JohnHauser] John Hauser, author of Berkeley SoftFloat, "Private communications

with”
[Wiki.DeterministicFiniteAutomaton] Wikipedia, “Deterministic Finite Automaton”
[CSC173] Randal Nelson, “CSC 173: Computation and Formal Systems”, University of Rochester
[Wiki.StateTransitionTable] Wikipedia, “State Transition Table”
[Calderone2013] Jean-Paul Calderone, “What is a State Machine?”
[GameProgrammingPatterns.StatePattern] Robert Nystrom, “Game Programming Patterns”
[Wiki.UMLStateMachine] Wikipedia, “UML State Machine”
[Gamasutra2001] Patrick Dickinson, “Instant Replay: Building a Game Engine with Reproducible Behavior”, Gamasutra, 2001
[Fiedler2015-2] Glenn Fiedler, “Deterministic Lockstep”, Gaffer on Games, 2015

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter V(b). Modular Architecture: Client-Side. Programmin...*

Chapter V(d). Modular Architecture: Client-Side. Client Arch... »

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

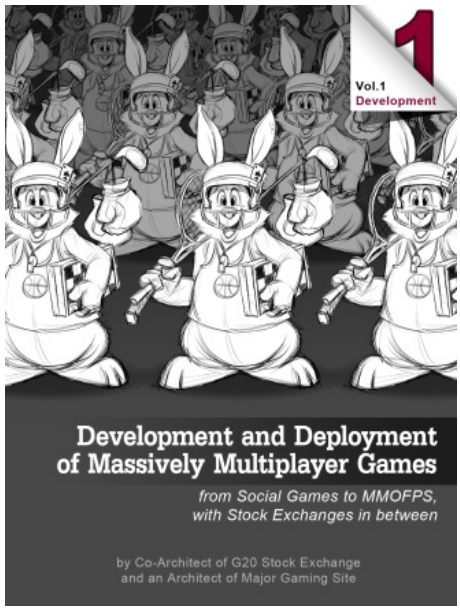
Copyright © 2014-2016 ITHare.com

Chapter V(d). Modular Architecture: Client-Side. Client Architecture Diagram, Threads, and Game Loop

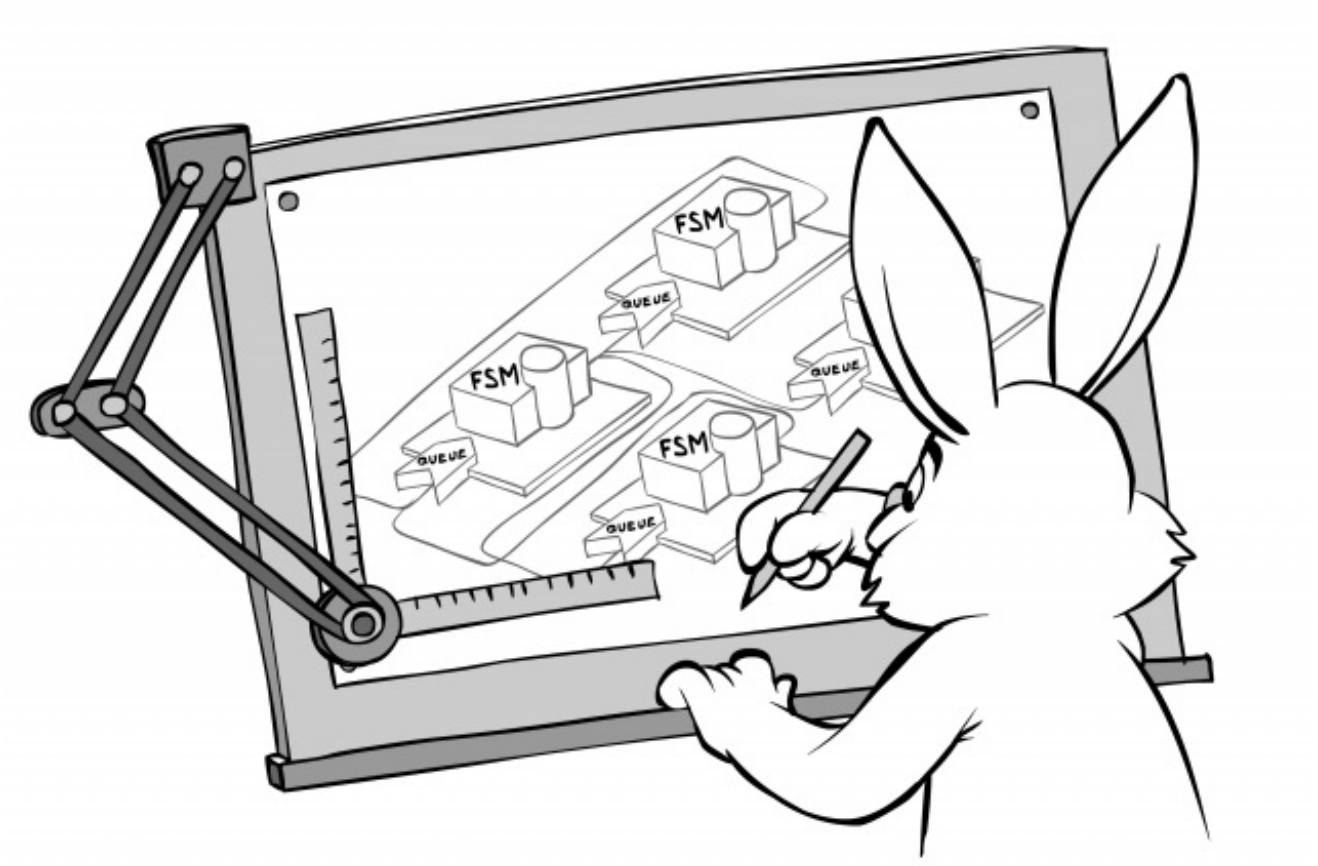
posted December 14, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

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



After we've spent quite a lot of time discussing boring things such as deterministic logic and finite automata, we can go ahead and finally draw the architecture diagram for our MMO game client. Yahoo!



However, as the very last delay before that glorious and long-promised diagram, we need to define one term that we'll use in this section. Let's define "tight loop" as an *"infinite loop which goes over and over without delays, and is regardless of any input"*.[1] In other words, tight loop is bound to eat CPU cycles (and lots of them) regardless of doing any useful work.

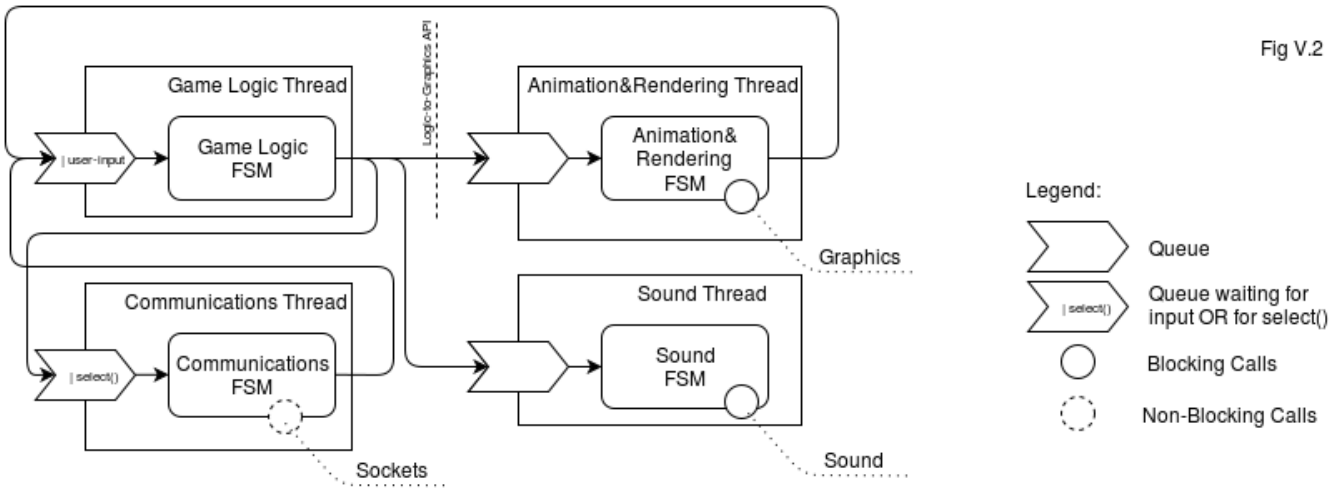
And now we're *really* ready for the diagram 🙂 .

---

[1] while different interpretations of the term "tight loop" exist out there, for our purposes this one will be the most convenient and useful

Queues-and-FSMs (QnFSM) Architecture: Generic Diagram

Fig. V.2 shows a diagram which describes a "generic" client-side architecture. It is admittedly more complicated than many of you will want or need to use; on the other hand, it represents quite a generic case, and many simplifications can be obtained right out of it by simple joining some of its elements.



Let's name this architecture a "Queues-and-FSMs Architecture" for obvious reasons, or "QnFSM" in short. Of course, QnFSM is (by far) not the only possible architecture, and even not the most popular one, but its variations have been seen to produce games with extremely good reliability, extremely good decoupling between parts, and very good maintainability. On the minus side, I can list only a bit of development overhead due to message-based exchange mechanism, but from my experience it is more than covered with better separation between different parts and very-well defined interfaces, which leads to the development speed-ups even in the medium-run (and is even more important in the long-run to avoid spaghetti code). Throw in the ability of "replay debug" and "replay-based post-mortem" in production, and it becomes a solution for lots of real-world pre-deployment and post-deployment problems.

In short – I'm an extremely strong advocate of this architecture (and its variations described below), and don't know of any practical cases when it is not the best thing you can do. While it might look over-engineered at the first glance, it pays off in the medium- and long-run[2]

I hope that the diagram on Fig V.2 should be more or less self-explanatory, but I will elaborate on a few points which might not be too obvious:

- each of FSMs is a strictly-deterministic FSM as described in "Event-Driven Programming and Finite State Machines" section above
  - while being strictly-deterministic is not a strict requirement, implementing your FSMs this way will make your debugging and post-mortem much much much easier.
- all the exchange between different FSMs is message-based. Here "message" is a close cousin of a network packet; in other words – it is just a bunch of bytes formatted according to some convention between sending thread and receiving thread.
  - There can be different ways how to pass these messages around; examples include explicit message posting, or implementing non-blocking RPC calls instead. While the Big Idea behind the QnFSM architecture won't change because of the way how the messages are posted, convenience and development time may change quite significantly. Still, while important, this is only an implementation detail which will be further discussed in Chapter [[TODO]].
  - for the messages between Game Logic Thread and Animation&Rendering Thread, format should be along the lines of "Logic-to-Graphics API", described in "Logic-to-Graphics Layer" section above. In short: it should be all about logical changes in the game world, along the lines of "NPC ID=ZZZ is currently moving along the path defined by the set of points {(X0,Y0),(X1,Y1),…} with speed V" (with coordinates being game world coordinates, not screen coordinates), or "Player at seat #N is in the process of showing his cards to the opponents".[3]

"There can be different ways how to pass these messages around; examples include explicit message posting, or implementing non-blocking RPC calls instead

- each thread has an associated Queue, which is able to accept messages, and provides a way to wait on it as long as is the Queue is empty

- the architecture is “Share-Nothing”. It means that there is no data shared between threads, and the only way to exchange data between threads, is via Queues and messages-passed-via-the-Queues
  - “share-nothing” means no thread synchronization problems (there is no need for mutexes, critical sections, etc. etc. outside of your queues). This is a Really Good Thing™, as trying to handle thread synchronization with any frequently changeable logic (such as the one within at least some of the FSMs) inevitably leads to lots and lots of problems (see, for example, [NoBugs2015])
  - of course, implementation of the Queues still needs to use inter-thread synchronization, but this is one-time effort and it has been done many times before, so it is not likely to cause too much trouble; see Chapter [[TODO]] for further details on Queues in C++
  - as a nice side effect, it means that whenever you want it, you can deploy your threads into different processes without changing *any* code within your FSMs (merely by switching to an inter-process implementation of the Queue). In particular, it can make answering very annoying questions such as “who’s guilty for the memory corruption” much more easily
- Queues of Game Logic Thread and Communications Thread, are rather unusual. They’re waiting not only for usual inter-thread messages, but also for some other stuff (namely input messages for Game Logic Thread, and network packets for the Communications Thread).
  - In most cases, at least one of these two particular queues will be supported by your platform (see Chapter [[TODO]] for details)
  - For those platforms which don’t support such queues – you can always use your-usual-inter-thread-queue (once again, the specifics will be discussed in Chapter [[TODO]]), and have an additional thread which will get user input data (or call select()), and then feed the data into your-usual-inter-thread-queue as a yet another message. This will create a strict functional equivalent (a.k.a. “compliant implementation”) of two specific Queues mentioned above

“In most cases, at least one of these two particular queues will be supported by your platform

- all the threads on the diagram (with one possible exception being Animation&Rendering Thread, see below) are *not* tight-looped, and unless there is something in their respective Queue – they just wait on the Queue until some kind of message comes in (or select() event happens)
  - while “no-tight-loops” is not a strict requirement for the client-side, wasting CPU cycles in tight loops without Really Good Reason is rarely a good idea, and might hurt quite a few of your players (those with weaker rigs).

- Animation&Rendering Thread is a potentially special case, and MAY use tight loop, see "Game Loop" subsection below for details
- to handle delays in other-than-Animation&Rendering Thread, Queues should allow FSMs to post some kind of "timer message" to the own thread
- even without tight loops it is possible to write your FSM in an "almost-tight-loop" manner that is closely resembling real-world real-time control systems (and classical Game Loop too), but without CPU overhead. See more on it in [[TODO!! – add subsection on it to "FSM" section]] section above.

---

[2] As usual, "I don't know of any cases" doesn't provide guarantees of any kind, and your mileage may vary, but at least before throwing this architecture away and doing something-that-you-like-better, please make sure to read the rest of this Chapter, where quite a few of potential concerns will be addressed
[3] yes, I know I've repeated it for quite a few times already, but it is *that* important, that I prefer to risk being bashed for annoying you rather than being pounded by somebody who didn't notice it and got into trouble

Migration from Classical 3D Single-Player Game

If you're coming from single-player development, you may find this whole diagram confusing; this maybe especially true for inter-relation between Game Logic FSM and Animation&Rendering FSM. The idea here is to have 95% of your existing "3D engine as you know it", with all the 3D stuff, as a part of "Animation&Rendering FSM". You will just need to cut off game decision logic (which will go to the server-side, and maybe partially duplicated to Game Logic FSM too for client-side prediction purposes), and UI logic (which will go into Game Logic FSM). All the mesh-related stuff should stay within Animation&Rendering FSM (even Game Logic FSM should know absolutely nothing about meshes and triangles).

"If your existing 3D engine is too complicated to fit into single-

If your existing 3D engine is too complicated to fit into single-threaded FSM, it is ok to keep it multi-threaded as long as it looks "just as an FSM" from the outside (i.e. all the communications with Animation&Rendering FSM go via messages or non-blocking RPC calls, expressed in terms of Logic-to-Graphics Layer). For details on using FSMs for multi-threaded 3D engines, see "On Additional Threads and Task-Based Multithreading" section below. Note that depending on specifics of your existing 3D rendering engine, you MAY need to resort to Option C; while Option C won't provide you with FSM goodies for your rendering engine (sorry, my supply of magic powder is quite limited), you will still be able to enjoy all

the benefits (such as replay debugging and production post-mortem) for the other parts of your client.

threaded FSM, it is ok to keep it multi-threaded as long as it looks 'just as an FSM' from the outside

It is worth noting that Game Logic FSM, despite its name, can often be more or less rudimentary, and (unless client-side prediction is used) mostly performs two functions: (a) parsing network messages and translating them into the commands of Logic-to-Graphics Layer, (b) UI handing. However, if client-side prediction is used, Game Logic FSM can become much more elaborated.

Interaction Examples in 3D World: Single-Player vs MMO

Let's consider three typical interaction examples after migration from single-player game to an MMO diagram shown above.

MMOFPS interaction example (shooting). Let's consider an MMOFPS example when Player A presses a button to shoot with a laser gun, and game logic needs to perform a raycast to see where it hits and what else happens. In single-player, all this usually happens within a 3D engine. For an MMO, it is more complicated:

- Step 1. button press goes to our authoritative server as a message
- Step 2. authoritative server receives message, performs a raycast, and calculates where the shot hits.
- Step 3. our authoritative server expresses "where it hits" in terms such as "Player B got hit right between his eyes"[4] and sends it as a message to the client (actually, to *all* the clients).
- Step 4. this message is received by Game Logic FSM, and translated into the commands of Logic-to-Graphics Layer (still without meshes and triangles, for example, "show laser ray from my gun to the point right-between-the-eyes-of-Player B", and "show laser hit right between the eyes of Player B"), which commands are sent (as messages) to Animation&Rendering FSM.
- Step 5. Animation&Rendering FSM can finally render the whole thing.[5]

While the process is rather complicated, most of the steps are inherently inevitable for an MMO; the only thing which you could theoretically save compared to the procedure described above, is merging step 4 and step 5 together (by merging Game Logic FSM and Animation&Rendering together), but I advise against it as such merging would introduce too much coupling which will hit you in the long run. Doing such different things as parsing network messages and rendering within one tightly coupled module is rarely a good idea, and it becomes even worse if there is a chance that you will ever want to use some other Animation&Rendering FSM (for example, a newer one, or the one optimized for a different platform).

MMORPG interaction example (ragdoll). In a typical MMORPG example, when an NPC is hit for 93th time and dies as a result, ragdoll physics is activated. In a typical single-player game, once again, the whole thing is usually performed within 3D engine. And once again, for a MMO the whole thing will be more complicated:

Ragdoll physics

In computer physics engines, ragdoll physics is a type of procedural animation that is often used as a replacement for traditional static death animations in video games and animated films.

— Wikipedia —

- Step 1. button press (the one which will cause NPC death) goes to authoritative server
- Step 2. server checks attack radius, calculates chances to hit, finds that the hit is successful, decreases health, and find that NPC is dead
- Step 3. server performs ragdoll simulation in the server-side 3D world. However, it doesn't need to (neither it really can) send it to clients as a complete triangle-based animation. Instead, the server can usually send to the client only a movement of "center of gravity" of NPC in question (calculated as a result of 3D simulation). This movement of "center of gravity" is sent to the client (either as a single message with the whole animation or as a series of messages with "current position" each)
  - as an interesting side-effect: as the whole thing is quite simple, there may be no real need to calculate the whole limb movement, and it may suffice to calculate just a simple parabolic movement of the "center of gravity", which MAY save you quite a bit of resources (both CPU and memory-wise) on the server side (!)
- Step 4. Game Logic FSM receives the message with "center of gravity" movement and translates it into Logic-to-Graphics commands. This doesn't necessarily need to be trivial; in particular, it may happen that Game Logic stores larger part of the game world than Animation&Rendering FSM. In this latter case, Game Logic FSM may want to check if this specific ragdoll animation is within the scope of the current 3D world of Animation&Rendering FSM.
- Step 5. Animation&Rendering FSM performs some ragdoll simulation (it can be pretty much the same simulation which has already been made on the server side, or something completely different). If ragdoll simulation is the same, then the process of ragdoll simulation on the client-side will be quite close to the one on the server-side; however, if there are any discrepancies due to not-so-perfect determinism – client-side simulation will correct coordinates so that "center of gravity" is adjusted to the position sent by server. In case of non-deterministic behaviour between client and server, the movement of the limbs on the client and the server may be different, but for a typical RPG it doesn't matter (what is really important is where the NPC eventually lands – here or over the edge of the cliff, but this is guaranteed to

be the same for all the clients as “center of gravity” comes from the server side).

UI interaction example. In a typical MMORPG game, a very common task is to show object properties when the object is currently under cursor. For the diagram above, it should be performed as follows:

- Step 1. Game Logic FSM sends a request to the Animation&Rendering FSM: “what is the object ID at screen coordinates (X,Y)?” (where (X,Y) are cursor coordinates)
- Step 2. Animation&Rendering FSM processes this (trivial) request and returns object ID back
- Step 3. Game Logic FSM finds object properties by ID, translates them into text, and instructs Animation&Rendering FSM to display object properties in HUD

While this may seem as an overkill, the overhead (both in terms of developer’s time and run time) is negligible, and good old rule of “the more cleanly separated parts you have – the easy is further development is” will more than compensate for the complexities of such separation.

---

[4] this is generally preferable to player-unrelated “laser hit at (X,Y,Z)” in case of client-side prediction; of course, in practice you’ll use some coordinates, but the point is that it is usually better to use player-related coordinates rather than absolute game world coordinates

[5] I won’t try to teach you how to render things; if you’re from 3D development side, you know much more about it than myself

FSMs and their respective States

The diagram on Fig. V.2 shows four different FSMs; while they all derive from our FiniteStateMachineBase described above, each of them is different, has a different function, and stores a different state. Let’s take a closer look at each of them.

Game Logic FSM

Game Logic FSM is the one which makes most of decisions about your game world. More strictly, these are not exactly decisions about the game world in general (this one is maintained by our authoritative server), but about *client-side copy* of the game world. In some cases it can be almost-trivial, in some cases (especially when client-side prediction is involved) it can be very elaborated.

In any case, Game Logic FSM is likely to keep a copy of the game world (or of relevant portion of the game world) from the game server, as a part of it's state. This copy has normally nothing to do with meshes, and describes things in terms such as "there is a PC standing at position (X,Y) in the game world coordinates, facing NNW", or "There are cards AS and JH on the table".

"Game Logic FSM is likely to keep a copy of the game world from the game server, as a part of it's state.

Game Logic FSM & Graphics

Probably the most closely related to Game Logic FSM is Animation&Rendering one. Most of the interaction between the two goes in the direction from Game Logic to Animation&Rendering, using Logic-to-Graphics Layer commands as messages. Game Logic FSM should instruct Animation&Rendering FSM to construct a portion of its own game copy as a 3D scene, and to update it as its own copy of the game world changes.

In addition, Game Logic FSM is going to handle (but not render) UI, such as HUDs, and various UI dialogs (including the dialogs leading to purchases, social stuff, etc.); this UI handling should be implemented in a very cross-platform manner, via sending messages to Animation&Rendering Engine. These messages, as usual, should be expressed in very graphics-agnostic terms, such as "show health at 87%", or "show the dialog described by such-and-such resource".

To handle UI, Game Logic FSM MAY send a message to Animation&Rendering FSM, requesting information such as "what object (or dialog element) is at such-and-such screen position" (once again, the whole translation between screen coordinates into world objects is made on the Animation&Rendering side, keeping Game Logic FSM free of such information); on receiving reply, Game Logic FSM may decide to update HUD, or to do whatever-else-is-necessary.

Other messages coming from Animation&Rendering FSM to Game Logic FSM, such as "notify me when the bullet hits the NPC", MAY be necessary for the client-side prediction purposes (see Chapter [[TODO]] for further discussion). On the other hand, it is very important to understand that these messages are non-authoritative by design, and that their results can be easily overridden by the server.

As you can see, there can be quite a few interactions between Game Logic FSM and Animation&Rendering FSM. Still, while it may be tempting to combine Game Logic FSM with Animation&Rendering FSM, I would advise against it at least for the games with many platforms to be supported, and for the games with Undefined Life Span; having these two FSMs separate (as shown on Fig V.2) will ensure much cleaner

separation, facilitating much-better-structured code in the medium- to long-run. On the other hand, having these two FSM running within the same thread is a very different story, is generally ok and can be done even on a per-platform basis; see "Variations" section below.

> "Having these two FSMs separate will ensure much cleaner separation, facilitating much-better-structured code in the medium- to long-run.

Game Logic FSM: Miscellaneous

There are two other things which need to be mentioned with regards to Game Logic FSM:

- **You MUST keep your Game Logic FSM truly platform-independent.** While all the other FSMs MAY be platform-specific (and separation between FSMs along the lines described above, facilitates platform-specific development when/if it becomes necessary), you should make all the possible effort to keep your Game Logic the same across all your platforms. The reason for it has already been mentioned before, and it is all about Game Logic being the most volatile of all your client-side code; it changes so often that you won't be able to keep several code bases reasonably in sync.
- If by any chance your Game Logic is that CPU-consuming that one single core won't cope with it – in most cases it can be addressed without giving up the goodies of FSM-based system, see "Additional Threads and Task-Based Multi-Threading" section below.

Animation&Rendering FSM

Animation&Rendering FSM is more or less similar to the rendering part of your usual single-player game engine. If your game is a 3D one, then in the diagram above,

> **it is Animations&Rendering FSM which keeps and cares about all the meshes, textures, and animations; as a Big Fat Rule of Thumb, nobody else in the system (including Game Logic FSM) should know about them.**

At the heart of the Animation&Rendering FSM there is a more or less traditional Game Loop.

Game Loop

Most of single-player games are based on a so-called Game Loop. Classical game loop looks more or less as follows (see, for example,

[GameProgrammingPatterns.GameLoop]):

```
while(true) {
  process_input();
  update();
  render();
}
```

Usually, Game Loop doesn't wait for input, but rather polls input and goes ahead regardless of the input being present. This is pretty close to what is often done in real-time control systems.

For our diagram on Fig V.2 above, within our Animation&Rendering Thread we can easily have something very similar to a traditional Game Loop (with a substantial part of it going within our Animation&Rendering FSM). Our Animation&Rendering Thread can be built as follows:

- Animation&Rendering Thread (outside of Animation&Rendering FSM) checks if there is anything in its Queue; unlike other Threads, it MAY proceed even if there is nothing in the Queue
- it passes whatever-it-received-from-the-Queue (or some kind of NULL if there was nothing) to Animation&Rendering FSM, alongside with any time-related information
- within the Animation&Rendering FSM's process_event(), we can still have process_input(), update() and render(), however:
  - there is no loop within Animation&Rendering FSM; instead, as discussed above, the Game Loop is a part of larger Animation&Rendering Thread
  - process_input(), instead of processing user input, processes instructions coming from Game Logic FSM
  - update() updates only 3D scene to be rendered, and not the game logic's representation of the game world; all the decision-making is moved at least to the Game Logic FSM, with most of the decisions actually being made by our authoritative server
  - render() works exactly as it worked for a single-player game
- after Animation&Rendering FSM processes input (or lack

"all the decision-making is moved at least to the Game Logic FSM, with most of the

thereof) and returns, Animation&Rendering Thread may conclude Game Loop as it sees fit (in particular, it can be done in any classical Game Loop manner mentioned below)

- then, Animation&Rendering Thread goes back to the very beginning (back to checking if there is anything in its Queue), which completes the infinite Game Loop.

decisions actually being made by our authoritative server

All the usual variations of Game Loop can be used within the Animation&Rendering Thread – including such things as fixed-time-step with delay at the end if there is time left until the next frame, variable-time-step tight loop (in this case a parameter such as elapsed_time needs to be fed to the Animation&Rendering FSM to keep it deterministic), and fixed-update-time-step-but-variable-render-time-step tight loop. Any further improvements (such as using VSYNC) can be added on top. I don't want to elaborate further here, and refer for further discussion of game loops and time steps to two excellent sources: [GafferOnGames.FixYourTimestep] and [GameProgrammingPatterns.GameLoop].

One variation of the Game Loop that is not discussed there, is a simple event-driven thing which you would use for your usual Windows programming (and without any tight loops); in this case animation under Windows can be done via WM_TIMER,[6] and 2D drawing – via something like BitBlt(). While usually woefully inadequate for any serious frames-per-second-oriented games, it has been seen to work very well for social- and casino-like ones.

However, the best thing about our architecture is that the architecture as such doesn't really depend on time step choices; you can even make different time step choices for different platforms and still keep the rest of your code (beyond Animation&Rendering Thread) intact, though Animation&Rendering FSM may need to be somewhat different depending on the fixed-step vs variable-step choice.[7]

"The best thing about our architecture is that the architecture as such doesn't really depend on time step choices; you can even make different time

Animation&Rendering FSM: Running from Game Logic Thread

For some games and/or platforms it might be beneficial to run Animation&Rendering FSM within the same thread as Game Logic FSM. In particular, if your game is a social game running on Windows, there may be no real need to use two separate CPU cores for Game Logic and Animation&Rendering, and the whole thing will be quite ok running within one single thread. In this case, you'll have one thread, one Queue, but two FSMs, with thread code outside of the FSMs deciding which of the

FSMs incoming message belongs to.

However, even in this case I still urge you to keep it as two separate FSMs with a very clean message-based interface between them. First, nobody knows which platform you will need to port your game next year, and second, clean well-separated interfaces at the right places tend to save lots of trouble in the long run.

step choices for different platforms and still keep the rest of your code intact

---

[6] yes, this does work, despite being likely to cause ROFLMAO syndrome for any game developer familiar with game engines

[7] of course, technically you may write your Animation&Rendering FSM as a variable-step one and use it for the fixed-step too, but there is a big open question if you really need to go the variable-step, or can live with a much simpler fixed-step forever-and-ever

Communications FSM

Another FSM, which is all-important for your MMOG, is Communications FSM. The idea here is to keep all the communications-related logic in one place. This may include very different things, from plain socket handling to such things as connect/reconnect logic[8], connection quality monitoring, encryption logic if applicable, etc. etc. Also implementations of higher-level concepts such as generic publisher/subscriber, generic state synchronization, messages-which-can-be-overridden etc. (see Chapter [[TODO]] for further details) also belong here.

"For most of (if not 'all') the platforms, the code of Communications FSM can be kept the same

For most of (if not "all") the platforms, the code of Communications FSM can be kept the same, with the only things being called from within the FSM, being your own wrappers around sockets (for C/C++ – Berkeley sockets). Your own wrappers are nice-to-have just in case if some other platform will have some peculiar ideas about sockets, or to make your system use something like OpenSSL in a straightforward manner. They are also necessary to implement "call interception" on your FSM (see "Implementing Strictly-Deterministic Logic: Strictly-Deteministic Code via Intercepting Calls" section above), allowing you to "replay test" and post-mortem of your Communications FSM.

The diagram of Fig. V.2 shows an implementation of the Communications FSM that uses non-blocking socket calls. For client-side it is perfectly feasible to keep the code of Communications FSM exactly the same, but to deploy it in a different manner, simulating non-blocking sockets via two additional

threads (one to handle reading and another to handle writing), with these additional threads communicating with the main Communications Thread via Queues (using Communication Thread's existing Queue, and one new Queue per new thread).[9]

One more thing to keep in mind with regards to blocking/non-blocking Berkeley sockets, is that getaddrinfo() function (as well as older gethostbyname() function) used for DNS resolution, is inherently blocking, with many platforms having no non-blocking counterpart. However, for the client side in most cases it is a non-issue unless you decide to run your Communications FSM within the same thread as your Game Logic FSM. In the latter case, calling a function with a potential to block for minutes, can easily freeze not only your game (which is more or less expected in case of connectivity problems), but also game UI (which is not acceptable regardless of network connectivity). To avoid this effect, you can always introduce yet another thread (with its own Queue) with the only thing for this thread to do, being to call getaddrinfo() when requested, and to send result back as a message, when the call is finished.[10]

Communications FSM: Running from Game Logic Thread

For Communications FSM, running it from Game Logic Thread might be possible. One reason against doing it, would be if your communications are encrypted, *and* your Game Logic is computationally-intensive.

And again, as with Animation&Rendering FSM, even if you run two FSMs from one single thread, it is much better to keep them separate. One additional reason to keep things separate (with this reason being specific to Communications FSM) is that Communications FSM (or at least large parts of it) is likely to be used on the server-side too.

---

[8] BTW, connect/reconnect will be most likely needed even for UDP

[9] for the server-side, however, these extra threads are not advisable due to the performance overhead. See Chapter [[TODO]] for more details

[10] Alternatively, it is also possible to create a new thread for each getaddrinfo() (with such a thread performing getaddrinfo(), reporting result back and terminating). This thread-per-request solution would work, but it would be a departure from QnFSM, and it can lead to creating too many threads in some fringe scenarios, so I usually prefer to keep a specialized thread intended for getaddrinfo() in a pure QnFSM model

Sound FSM

Sound FSM handles, well, sound. In a sense, it is somewhat similar to Animation&Rendering FSM, but for sound. Its interface (and as always with QnFSM,

interfaces are implemented over messages) needs to be implemented as a kind of "Logic-to-Sound Layer". This "Logic-to-Sound Layer" message-based API should be conceptually similar to "Logic-to-Graphics Layer" with commands going from the Game Logic expressed in terms of "play this sound at such-and-such volume coming from such-and-such position within the game world".

Sound FSM: Running from Game Logic Thread

For Sound FSM running it from the same thread as Game Logic FSM makes sense quite often. On the other hand, on some platforms sound APIs (while being non-blocking in a sense that they return before the sound ends) MAY cause substantial delays, effectively blocking while the sound function finds and parses the file header etc.; while this is still obviously shorter than waiting until the sound ends, it might be not short enough depending on your game. Therefore, keeping Sound FSM in a separate thread MAY be useful for fast-paced frame-per-second-oriented games.

And once again – even if you decide to run two FSMs from the same thread – do yourself a favour and keep the FSMs separate; some months down the road you'll be very happy that you kept your interfaces clean and different modules nicely decoupled.[11]

"Once again – even if you decide to run two FSMs from the same thread – do yourself a favour and keep the FSMs separate

[11] Or you'll regret that you didn't do it, which is pretty much the same thing

Other FSMs

While not shown on the diagram on Fig V.2, there can be other FSMs within your client. For example, these FSMs may run in their own threads, but other variations are also possible.

One practical example of such a client-side FSM (which was implemented in practice) was "update FSM" which handled online download of DLC while making sure that the gameplay delays were within acceptable margins (see more on client updates in general and updates-while-playing in Chapter [[TODO]]).

In general, any kind of entity which performs mostly-independent tasks on the client-side, can be implemented as an additional FSM. While I don't know of practical examples of extra client-side FSMs other than "update FSM" described above, it doesn't mean that your specific game won't allow/require any, so keep your eyes open.

On Additional Threads and Task-Based Multithreading

If your game is very CPU-intensive, and either your Game Logic Thread, or Animation&Rendering Thread become overloaded beyond capabilities of one single CPU core, you might need to introduce an additional thread or five into the picture. This is especially likely for Animation&Rendering Thread/FSM if your game uses serious 3D graphics. While complexities threading model of 3D graphics engines are well beyond the scope of this book, I will try to provide a few hints for those who're just starting to venture there.

As usually with multi-threading, if you're not careful, things can easily become ugly, so in this case:

IIRC
abbv for If I Recall Correctly
— Urban Dictionary
—

- first of all, take another look if you have some Gross Inefficiencies in your code; it is usually much better to remove these rather than trying to parallelize. For example, if you'd have calculated Fibonacci numbers recursively, it is much better to switch to non-recursive implementation (which is IIRC has humongous $O(2^N)$ advantage over recursive one[12]) than to try getting more and more cores working on unnecessary stuff.
- From this point on, to the best of my knowledge you have about three-and-a-half options:
  - **Option A.** The first option is to split the whole thing into several FSMs running within several threads, dedicating one thread per one specific task. In 3D rendering world, this is known as "System-on-a-Thread", and was used by Halo engine (in Halo, they copy the whole game state between threads[GDC.Destiny], which is equivalent to having a queue, so this is a very direct analogy of our QnFSM).
  - **Option B.** The second option is to "off-load" some of the processing to a different thread, with this new thread being just as all the other threads on Fig V.2; in other words, it should have an input queue and a FSM within. This is known as "Task-Based Multithreading" [GDC.TaskBasedMT]. In this case, after doing its (very isolated) part of the job a.k.a. "task", the thread may report back to the whichever-thread-has-requested-its-services. This option is really good for several reasons, from keeping all the FSM-based goodies (such as "replay testing" and post-mortem) for all parts of your client, to encouraging multi-threading model with very few context switches (known as "Coarse-grained parallelism"), and context switches are damn expensive on all general-purpose CPUs.[13] The way how "task off-loading" is done, depends on the implementation. In some implementations, we MAY use data-driven pipelines (similar to those described in [GDC.Destiny]) to enable dynamic task balancing, which allows to optimize core utilization

on different platforms. Note that in pure "Option B", we still have shared-nothing model, so each of the FSMs has it's own exclusive state. On the other hand, for serious rendering engines, due to the sheer size of the game state, pure "shared-nothing" approach MIGHT BE not too feasible.

- **Option B1.** That's the point where "task-off-loading-with-an-immutable-shared-state" emerges. It improves[14] over a basic Option B by allowing for a *very-well-controlled* use of a shared state – namely, *sharing is allowed only when the shared state is guaranteed to be immutable*. It means that, in a limited departure from our shared-nothing model, in addition to inter-thread queues in our QnFSM, we MAY have a shared state. However, to avoid those nasty inter-thread problems, we MUST guarantee that while there is more than one thread which can be accessing the shared state, the shared state is constant/immutable (though it may change outside of "shared" windows). At the moment, it is unclear to me whether Destiny engine (as described in [GDC.Destiny]) uses Option B1 (with an immutable game state shared between threads during "visibility" and "extract" phases) – while it *looks* likely, it is not 100% clear. In any case, both Option B and Option B1 can be described more or less in terms of QnFSM (and most importantly – both eliminate all the non-maintainable and inefficient tinkering with mutexes etc. within your logic). From the point of view of determinism, Option B1 is equivalent to Option B, provided that we consider that immutable-shared-state as one of our inputs (as it is immutable, it is indistinguishable from an input, though delivered in a somewhat different way); while such a game sharing would effectively preclude from applying recording/replay in production (as recording the whole game state on each frame would be too expensive), determinism can still be used for regression testing etc.

- **Option C.** To throw away "replay debug" and post-mortem benefits *for this specific FSM,* and to implement it using multi-thread in-whatever-way-you-like (i.e. using traditional inter-thread synchronization stuff such as mutexes, semaphores, or Dijkstra forbid – memory fences etc. etc.).
  - This is a very dangerous option, and it is to be avoided as long as possible. However, there are some cases when clean separation between the main-thread-data and data-necessary-for-the-secondary-thread is not feasible, usually because of the piece of data to be used by both parallel processes, being too large; it is these cases (and to the best of my knowledge, *only* these cases), when you need to choose Option C. And even in these cases, you might be able to

"If you need Option C for your Game Logic – think

stay away from handling fine-grained thread synchronization, see Chapter [[TODO]] for some hints in this direction.

twice, and then twice more.

- Also, if you need Option C for your Game Logic – think twice, and then twice more. As Game Logic is the one which changes *a damn lot,* with Option C this has all the chances of becoming unmanageable (see, for example, [NoBugs2015]). It is *that* bad, that if you run into this situation, I would seriously think whether the Game Logic requirements are feasible to implement (and maintain) at all.
- On the positive side, it should be noted that even in such an unfortunate case you should be losing FSM-related benefits (such as "replay testing" and post-mortem) only for the FSM which you're rewriting into Option C; all the other FSMs will still remain deterministic (and therefore, easily testable).

- In any case, your multi-threaded FSM SHOULD look as a normal FSM from the outside. In other words, multi-threaded implementation SHOULD be just this – implementation detail *of this particular FSM*, and SHOULD NOT affect the rest of your code. This is useful for two reasons. First, it decouples things and creates a clean well-defined interface, and second, it allows you to change implementation (or add another one, for example, for a different platform) without rewriting the whole thing.

---

[12] that is, if you're not programming in Haskell or something similar
[13] GPGPUs is the only place I know where context switches are cheap, but usually we're not speaking about GPGPUs for these threads
[14] or "degrades", depending on the point of view

On Latencies

One question which may arise for queue-based architectures and fast-paced games, is about latencies introduced by those additional queues (we *do* want to show the data to the user as fast as possible). My experience shows that[15] then we're speaking about additional latency[16] of the order of single-digit microseconds. Probably it can be lowered further into sub-microsecond range by using less trivial non-blocking queues, but this I'm not 100% sure of because of relatively expensive allocations usually involved in marshalling/unmarshalling; for further details on implementing high-performance low-latency queues in C++, please refer to Chapter [[TODO]]. As this single-digit-microsecond delay is at least 3 orders of magnitude smaller than inter-frame delay of 1/60 sec or so, I am arguing that nobody will ever notice the difference, even for single-player or LAN-based games; for Internet-based MMOs where the absolutely best we can hope for is 10ms delay,[17] makes it even less relevant.

In short – I don't think this additional single-digit-microsecond delay can possibly have any effect which is visible to end-user.

---

[15] assuming that the thread is not busy doing something else, and that there are available CPU cores
[16] introduced by a reasonably well-designed message marshalling/unmarshalling + reasonably well-designed inter-process single-reader queue
[17] see Chapter [[TODO]] for conditions when such delays are possible before hitting me too hard

Variations

The diagram on Fig V.2 shows each of the FSMs running within it's own thread. On the other hand, as noted above, each of the FSMs can be run in the same thread as Game Logic FSM. In the extreme case it results in the system where all the FSMs are running within single thread with a corresponding diagram shown on Fig V.3:

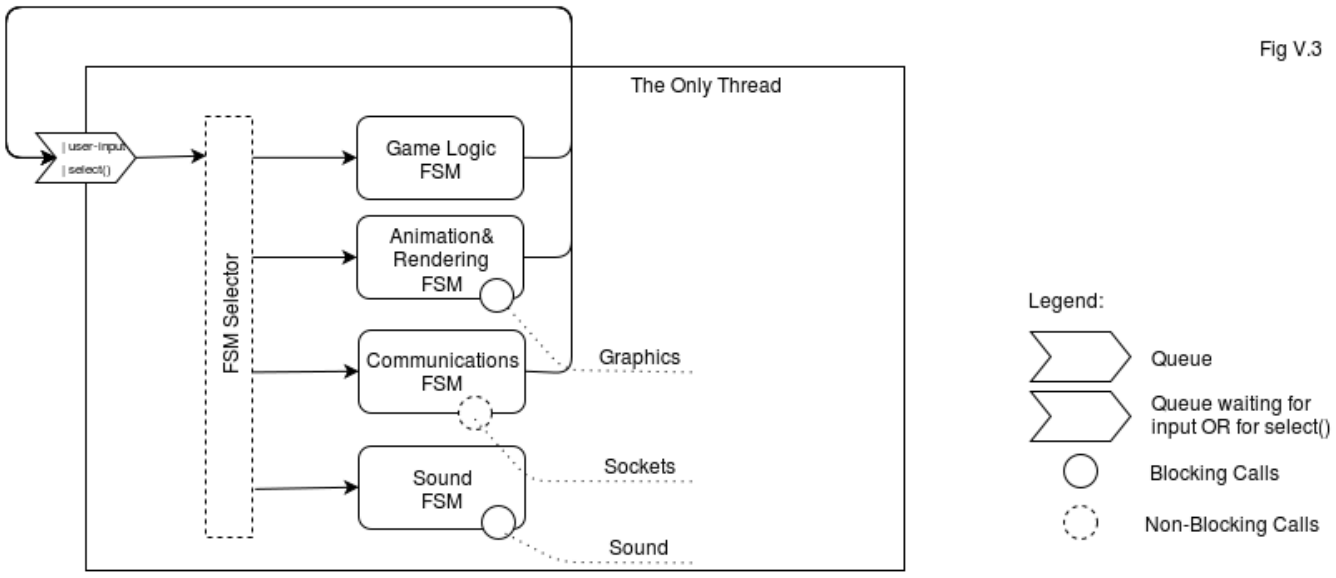
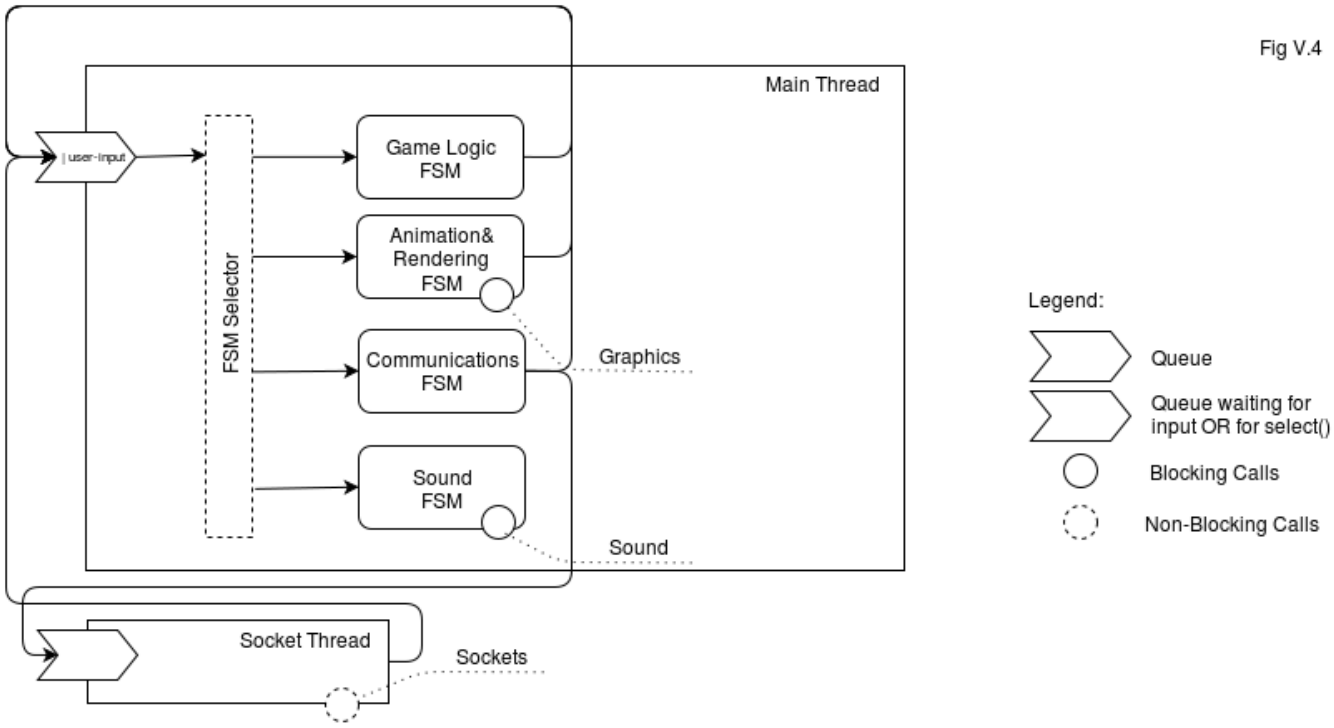


Fig V.3

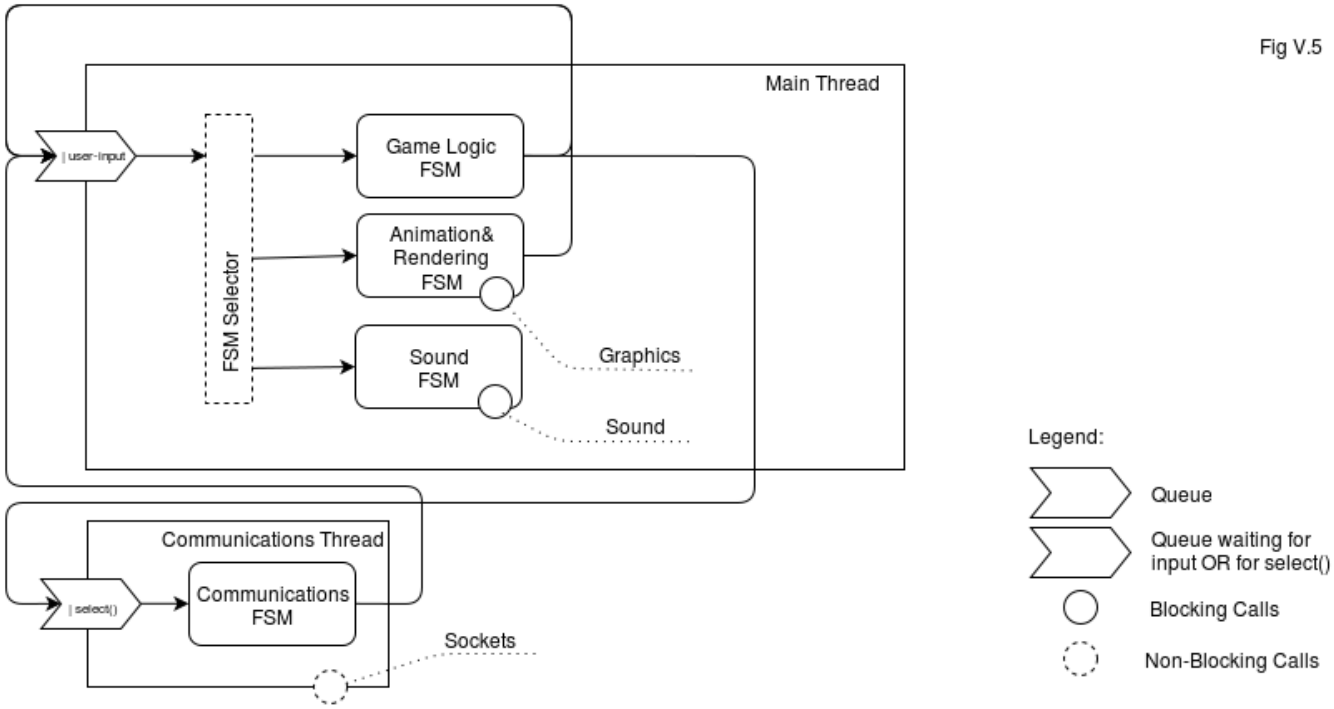
Each and every of FSMs on Fig V.3 is exactly the same as an FSM on Fig V.2; moreover, logically, these two diagrams are exactly equivalent (and "recording" from one can be "replayed" on another one). The only difference on Fig V.3 is that we're using the same thread (and the same Queue) to run all our FSMs. FSM Selector here is just a very dumb selector, which looks at the *destination-FSM* field (set by whoever-sent-the-message) and routes the message accordingly.

This kind of threading could be quite practical, for example, for a casino or a social game. However, not all the platforms allow to wait for the select() in the main graphics loop, so you may need to resort to the one on Fig V.4:



Here *Sockets Thread* is very simple and doesn't contain any substantial logic; all it does is just pushing whatever-it-got-from-Queue to the socket, and pushing whatever-it-got-from-socket – to the Queue of the *Main Thread;* all the actual processing will be performed there, within *Communications FSM.*

Another alternative is shown on Fig V.5:



Both Fig V.4 and Fig V.5 will work for a social or casino-like game on Windows.[18]

On the other end of the spectrum, lie such heavy-weight implementations as the one shown on Fig V.6:

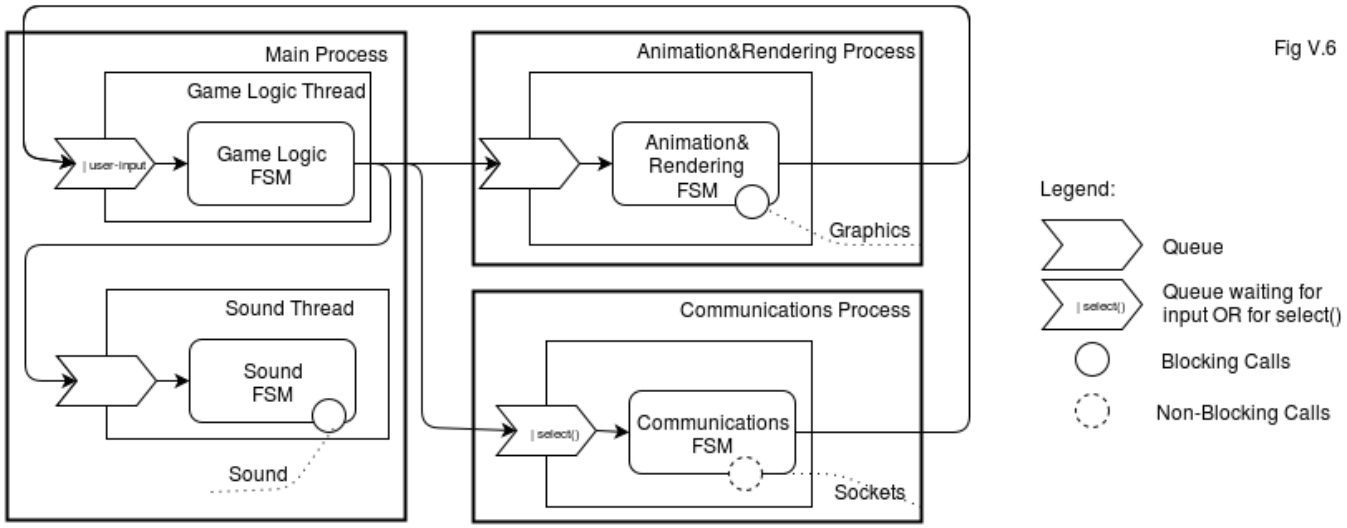


Fig V.6

Here, Animation&Rendering FSM, and Communications FSM run in their own processes. This approach might be useful during testing (in general, you may even run FSMs on different developer's computers if you prefer this kind of interactive debugging). However, for production it is better to avoid such configurations, as inter-process interfaces may help bot writers.

Overall, an exact thread (and even process) configuration you will deploy is not that important and may easily be system-dependent (or even situation-dependent, as in "for the time being, we've decided to separate this FSM to a separate process to debug it on respective developer's machines"). What really matters is that

as long as you're keeping your development model FSM-based, you can deploy it in any way you like without any changes to your FSMs.

In practice, this property has been observed to provide quite a bit of help in the long run. While this effect has significantly more benefits on the server-side (and will be discussed in Chapter [[TODO]]), it has been seen to aid client-side development too; for example, different configurations for different platforms do provide quite a bit of help. In addition, situation-dependent configurations have been observed to help a bit during testing.

---

[18] While on Windows it is possible to create both "|select()" and "|user-input" queues, I don't know how to create one single queue which will be both "|select()" and "|user-input" simultaneously, without resorting to a 'dumb' extra thread; more details on these and other queues will be provided in Chapter [[TODO]]

On Code Bases for Different Platforms

As it was noted above, you MUST keep your Game Logic FSM the same for all the platforms (i.e. as a single code base). Otherwise, given the frequent changes to

Game Logic, all-but-one of your code bases will most likely start to fall behind, to the point of being completely useless.

But what about other FSMs? Do you need to keep them as a single code base? The answer here is quite obvious:

while the architecture shown above allows you to make non-Game-Logic FSMs platform-specific, it makes perfect sense to keep them the same as long as possible

"If your game is graphics-intensive, there can be really good reasons to have your Animation&Rendering FSM different for different platforms

For example, if your game is graphics-intensive, there can be really good reasons to have your Animation&Rendering FSM different for different platforms; for example, you may want to use DirectX on some platforms, and OpenGL on some other platforms (granted, it will be quite a chunk of work to implement both of them, but at least it is possible with the architecture above, and it becomes a potentially viable business choice, especially as OpenGL version and DirectX version can be developed in parallel).

On the other hand, chances that you will need the platform-specific Communications FSM, are much lower.[19] Even if you're writing in C/C++, usable implementations of Berkeley sockets exist on most (if not on *all*) platforms of interest. Moreover, the behavior of sockets on different platforms is quite close from game developer's point of view (at least with regards to those things which we are able to affect).

So, while all such choices are obviously specific to your specific game, statistically you should have much more Animation&Rendering FSMs than Communications FSMs 🙂 .

---

[19] I don't count conditional inclusion of WSAStartup() etc. as being really platform-specific

QnFSM Architecture Summary

Queues-and-FSMs Architecture shown on Fig V.2 (as well as its variations on Fig V.3-Fig V.6) is quite an interesting beast. In particular, while it does ensure a clean separation between parts (FSMs in our case), it tends to go against commonly used patterns of COM-like components or even usual libraries. The key difference here

is that COM-like components are essentially based on blocking RPC, so after you called a COM-like RPC[20], you're blocked until you get a reply. With FSM-based architecture from Fig V.2-V.6, even if you're requesting something from another FSM, you still can (and usually should) process events coming while you're waiting for the reply. See in particular [[TODO!! add subsection on callbacks to FSM]] section above.

From my experience, while developers usually see this kind of FSM-based programming as somewhat more cumbersome than usual procedure-call-based programming, most of them agree that it is beneficial in the medium- to long-run. This is also supported by experiences of people writing in Erlang, which has almost exactly the same approach to concurrency (except for certain QnFSM's goodies, see also "Relation to Erlang" section below). As advantages of QnFSM architecture, we can list the following:

"Most of developers agree that FSM-based programming is beneficial in the medium- to long-run.

- very good separation between different modules (FSMs in our case). FSMs and their message-oriented APIs tend to be isolated very nicely (sometimes even a bit too nicely, but this is just another side of the "somewhat more cumbersome" negative listed above).
- "replay testing" and post-mortem analysis. See "Strictly-Deterministic Logic: Benefits" section above.
- very good performance. While usually it is not *that* important for client-side, it certainly doesn't hurt either. The point here is that with such an architecture, context switches are kept to the absolute minimum, and each thread is working without any pauses (and without any overhead associated with these pauses) as long as it has something to do. On the flip side, it doesn't provide inherent capabilities to scale (so server-side scaling needs to be introduced separately, see Chapter [[TODO]]), but at least it is substantially better than having some state under the mutex, and trying to lock this mutex from different threads to perform something useful.

We will discuss more details on this Queues-and-FSMs architecture as applicable to the server-side, in Chapter [[TODO]], where its performance benefits become significantly more important.

Relation to Actor Concurrency

NB: this subsection is entirely optional, feel free to skip it if theory is of no interest to you

Actor Concurrency Model

The actor model

From theoretical point of view QnFSM architecture can be

seen as a system which is pretty close to so-called "Actor Concurrency Model" (that is, until Option C from "Additional Threads and Task-Based Multithreading" is used), with QnFSM's deterministic FSMs being Actor Concurrency's 'Actors'. However, there is a significant difference between the two, at least perceptionally. Traditionally, Actor concurrency is considered as a way to ensure concurrent calculations; that is, the calculation which is considered is originally a "pure" calculation, with all the parameters known in advance. With games, the situation is very different because we don't know everything in advance (by definition). This has quite a few implications.

Most importantly, system-wide determinism (lack of which is often considered a problem for Actor concurrency when we're speaking about calculations) is not possible for games.[21] In other words, games (more generally, *any distributed interactive system which produces results substantially dependent on timing;* dependency on timing can be either absolute, like "whether the player pressed the button before 12:00", or relative such as "whether player A pressed the button before player B") are inherently non-deterministic when taken as a whole. On the other hand, each of the FSMs/Actors can be made completely deterministic, and this is what I am arguing for in this book.

In other words – while QnFSM is indeed a close cousin of Actor concurrency, quite a few of the analysis made for Actor-concurrency-for-HPC type of tasks, is not exactly applicable to inherently time-dependent systems such as games, so take it with a big pinch of salt.

in computer science is a mathematical model of concurrent computation that treats 'actors' as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received.

— Wikipedia —

---

[20] also DCE RPC, CORBA RPC, and so on; however, game engine RPCs are usually very different, and you're *not* blocked after the call, in exchange for not receiving anything back from the call

[21] the discussion of this phenomenon is out of scope of this book, but it follows from inherently distributed nature of the games, which, combined with Einstein's light cone and inherently non-deterministic quantum effects when we're organizing transmissions from client to server, mean that very-close events happening for different players, may lead to random results when it comes to time of arrival of these events to server. Given current technologies, determinism is not possible as soon as we have more than one independent "clock domain" within our system (non-deterministic behaviour happens at least due to metastability problem on inter-clock-domain data paths), so at the very least any real-world multi-device game cannot be made fully deterministic in any practical sense.

Relation to Erlang Concurrency and Akka Actors

On the other hand, if looking at Erlang concurrency (more specifically, at *!* and *receive* operators), or at Akka's Actors, we will see that QnFSM is pretty much the same thing.[22] There are no shared states, everything goes via message passing, et caetera, et caetera, et caetera.

The only significant difference is that for QnFSM I am arguing for determinism (which is not guaranteed in Erlang/Akka, at least not without "call interception"; on the other hand, you can write deterministic actors in Erlang or Acca the same way as in QnFSM, it is just an additional restriction you need to keep in mind and enforce). Other than that, and some of those practical goodies in QnFSM (such as recording/replay with all the associated benefits), QnFSM is extremely close to Erlang's concurrency (as well as to Akka's Actors which were inspired by Erlang) from developer's point of view.

Which can be roughly translated into the following observation:

to have a good concurrency model, it is not strictly necessary to program in Erlang or to use Akka

[22] While both Erlang and Akka zealots will argue ad infinitum that their favourite technology is much better, from our perspective the differences are negligible

Akka
is... simplifying the construction of concurrent and distributed applications on the JVM. Akka... emphasizes actor-based concurrency, with inspiration drawn from Erlang.
— Wikipedia —

Erlang
Erlang is a general-purpose, concurrent, garbage-collected programming language and runtime system.
— Wikipedia —

Bottom Line for Chapter V

Phew, it was a long chapter. On the other hand, we've managed to provide a 50'000-feet (and 20'000-word) view on my favorite MMOG client-side architecture. To summarize and re-iterate my recommendations in this regard:

- Think about your graphics, in particular whether you want to use pre-rendered 3D or whether you want/need dual graphics (such as 2D+3D); this is one of the most important questions for your game client;[23] moreover, client-side 3D is not always the best choice, and there are quite a few MMO games out there which have rudimentary graphics
    - if your game is an MMOFPS or an MMORPG, most likely you do need

fully-fledged client-side 3D, but even for an MMORTS the answer can be not that obvious

- when choosing your programming language, think twice about resilience to bot writers, and also about those platforms you want to support. While the former is just one of those things to keep in mind, the latter can be a deal-breaker when deciding on your programming language
  - Usually, C++ is quite a good all-around candidate, but you need to have pretty good developers to work with it
- Write your code in a deterministic event-driven manner (as described in "Strictly-Deterministic Logic" and "Event-Driven Programming and Finite State Machines" sections), it helps, and helps a lot
  - This is not the only viable architecture, so you may be able to get away without it, but at the very least you should consider it and understand why you prefer an alternative one
  - The code written this way magically becomes a deterministic FSM, which has lots of useful implications
  - Keep all your FSMs perfectly self-contained, in a "Share-Nothing" model. It will help in quite a few places down the road.
  - Feel free to run multiple FSMs in a single thread if you think that your game and/or current platform is a good fit, but keep those FSMs separate; it can really save your bacon a few months later.
  - Keep one single code base for Game Logic FSM. For other FSMs, you may make different implementations for different platforms, but do it only if it becomes really necessary.

> **"Write your code in a deterministic event-driven manner, it helps, and helps a lot**

---

[23] yes, I know I'm putting on my Captain Obvious' hat once again here

[[To Be Continued…

This concludes beta Chapter V(d) 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 VI, "Modular Architecture: Server-Side. Naive and Classical Deployment Architectures.]]

[–] References

[NoBugs2015] 'No Bugs' Hare, “Multi-threading at Business-logic Level is Considered Harmful”, Overload #128
[GameProgrammingPatterns.GameLoop] Robert Nystrom, “Game Programming Patterns”
[GafferOnGames.FixYourTimestep] Glenn Fiedler, “Fix Your Timestep!”, Gaffer On Games
[GDC.Destiny] Natalya Tatarchuk, “Destiny's Multithreaded Rendering Architecture”, GDC2015
[GDC.TaskBasedMT] Ron Fosner, “Task-based Multithreading - How to Program for 100 cores”, GDC2010

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter V(c). Modular Architecture: Client-Side. On Debuggin..*

Chapter VI(a). Server-Side MMO Architecture. Naïve, Web-Ba... »

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

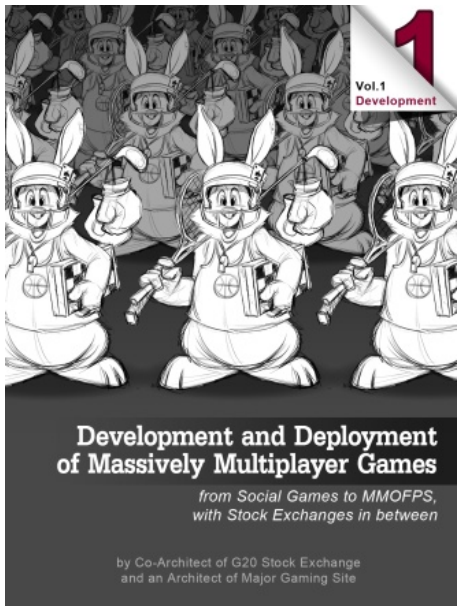
Copyright © 2014-2016 ITHare.com

Chapter VI(a). Server-Side MMO Architecture. Naïve, Web-Based, and Classical Deployment Architectures

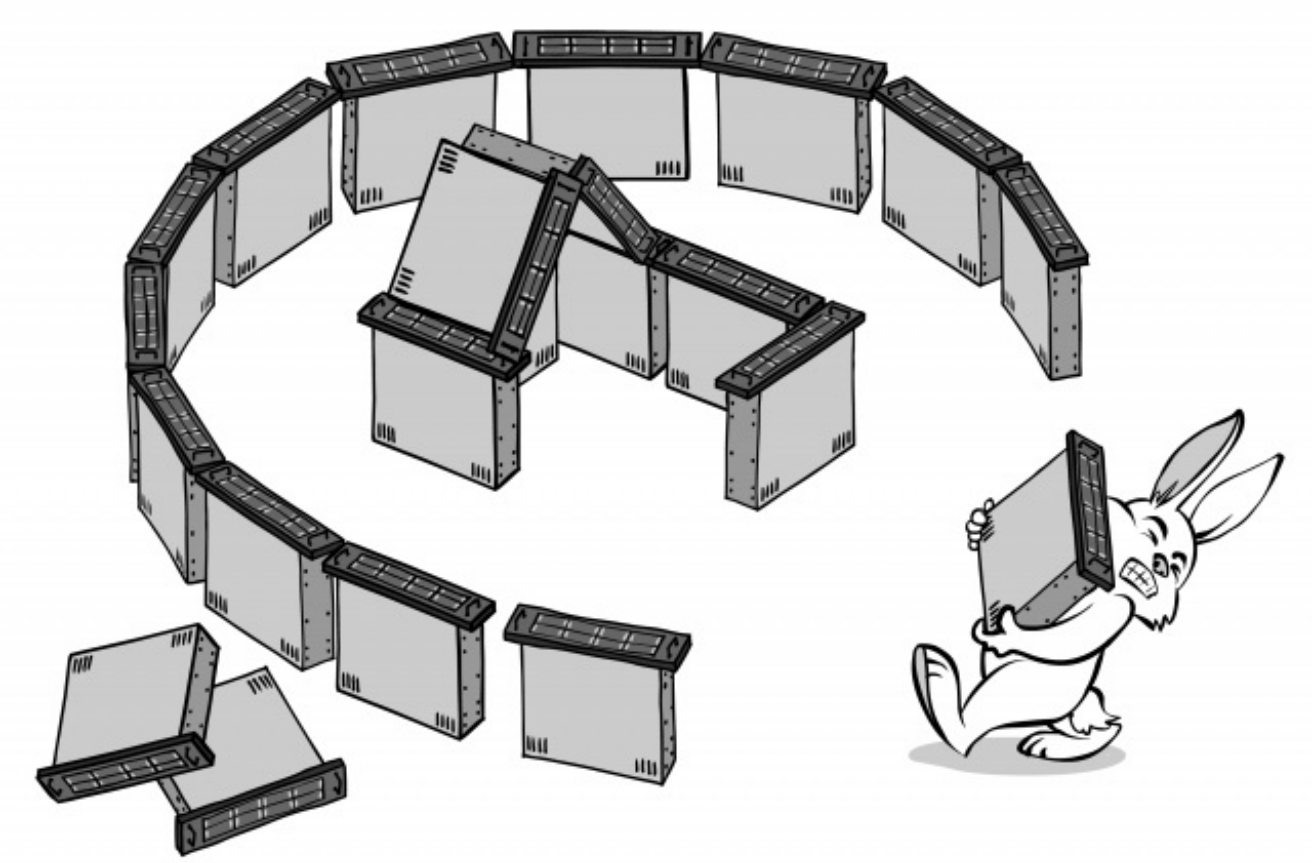
posted December 21, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter VI(a) 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.]]



After drawing all that nice client-side QnFSM-based diagrams, we need to describe our server architecture. The very first thing we need to do is to start thinking in terms of "how we're going to deploy our servers, when our game is ready?" Yes, I really mean it – architecture starts not in terms of classes, and for the server-side – not even in terms of processes or FSMs, it starts with the highest-level meaningful diagram we can draw, and for the server-side this is a deployment diagram with servers being its main building blocks. If deploying to cloud, these may be virtual servers, but a concept of "server" which is a "more or less self-contained box running our server-side software", still remains very central to the server-side software. If not thinking about clear separation between the pieces of your software, you can easily end up with a server-side architecture that looks nicely while you program it, but falls apart on the third day after deployment, exactly when you're starting to think that your game is a big success.



Deployment Architectures, Take 1

In this Chapter we'll discuss only "basic" deployment architectures. These architectures are "basic" in a sense that they're usually sufficient to deploy your game and run it for several months, but as your game grows, further improvements may become necessary. Fortunately, these improvements can be done later, when/if the problems with basic deployment architecture arise; these improvements will be discussed in Chapter [[TODO]].

Also note that for your very first deployment, you may have much less physical/virtual boxes than shown on the diagram, by combining quite a few of them together. On the other hand, you should be able to increase the number of your servers quickly, so you need to have the software able to work in basic deployment architecture from the very beginning. This is important, as demand for increase in number of servers can develop very soon if you're successful. We'll discuss your very first deployment in Chapter [[TODO]].

First, let's start with an architecture you shouldn't do.

Don't Do It: Naïve Game Deployment Architectures

Quite often, when faced with development their very first multi-player game, developers start with something like the following Fig VI.1:

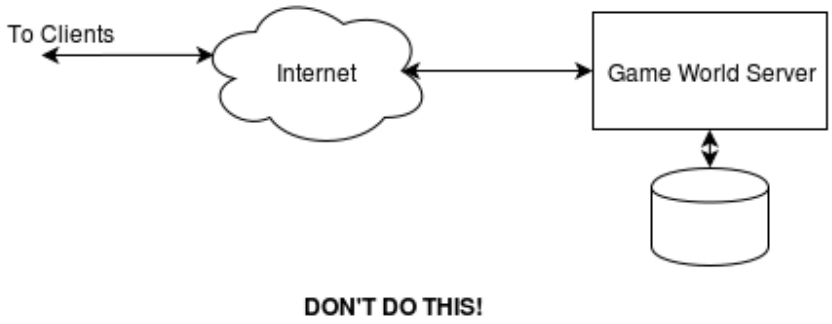


Fig VI.1

It is dead simple: there is a server, and there is a database to store persistent state. And later on, as one single Game World server proves to be insufficient, it naturally evolves into something like the diagram on Fig VI.2:

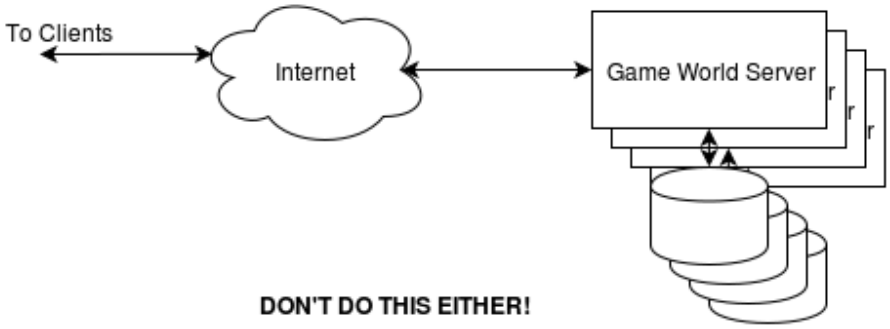


Fig VI.2

with each of Game World servers having its own database.

My word of advice about such naïve deployment architectures:

DON'T DO THIS!

Such a naïve approach won't work well for a vast majority of games. The problem here (usually ranging from near-fatal to absolutely-fatal depending on specifics of your game) is that this architecture doesn't allow for interaction between players coming from different servers. In particular, such an architecture becomes absolutely deadly if your game allows some way for a player to choose who he's playing with (or if you have some kind of merit-based tournament system), in other words – if you're *not* allowed to arbitrary separate your players (and in most cases you will need some kind of interaction at least because of the social network integration, see Chapter II for further discussion in this regard).

CSR
Customer service representatives interact with customers to

For the naïve architecture shown on Fig VI.2, any interaction between separate players coming from separate databases, leads to huge mortgage-crisis-size problems. Inter-DB interaction, while possible (and we'll discuss it in Chapter [[TODO]]) won't work well around these lines and between completely independent databases. You're going to have lots and lots of problems, ranging from delays due to improperly implemented inter-DB transactions (apparently this is not

provide answers to inquiries involving a company's product or services.
— Wikipedia —

that easy), to your CSRs going crazy because of two different users having the same ID in different databases. Moreover, if you start like this, you will even have trouble merging the databases later (the very first problem you will face will be about collisions in user names between different DBs, with much more to follow).

To summarize relevant discussion from Chapter II and from present Chapter:

A. You WILL need inter-player interaction between arbitrary players. If not now, then later. B. Hence, you SHOULD NOT use "naïve" architecture shown above.

Fortunately, there are relatively simple and practical architectures which allow to avoid problems typical for naïve approaches shown above.

Web-Based Game Deployment Architecture

If your game satisfies two conditions:

- first, it is reeeeallyyyy sloooow-paaaaaced (in other words, it is not an MMOFPS and even not a poker game) and/or "asynchronous" (as defined in Chapter I, i.e. it doesn't need players to be present simultaneously),
- *and* second, it has little interaction between players (think farming-like games with only occasional inter-player interaction),

then you might be able to get away with Web-Based server-side architecture, shown on Fig VI.3:

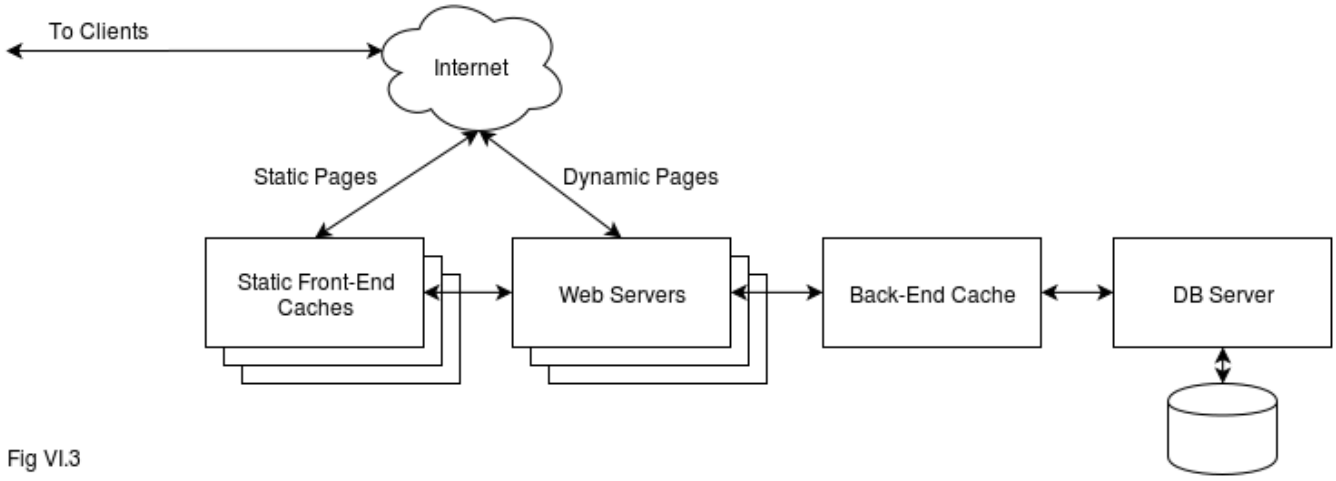


Fig VI.3

Web-Based Deployment Architecture: How It Works

The whole thing looks alongside the lines of a heavily-loaded web app – with lots of caching, both at front-end (to cache pages), and at a back-end. However, there are also significant differences (special thanks to Robert Zubek for sharing his experiences in this regard, [Zubek2016]).

The question "which web server to use" is not *that* important here. On the other hand, there exists an interesting and not-so-well-known web server, which took an extra mile to improve communications in game-like environments. I'm speaking about [Lightstreamer]. I didn't try it myself, so I cannot vouch for it, but what they're doing with regards to improving interactivity over TCP, is really interesting. We'll discuss some of their tricks in Chapter [{TODO]].

Peculiarities in Web-Based Game architectures are mostly about the way caching is built. First, on Fig VI.3 both front-end caching and back-end caching is used. Front-end caching is your usual page caching (like nginx in reverse-proxy mode, or even a CDN), though there is a caveat. As your current-game-data changes very frequently, you normally don't want to cache it, so you need to take an effort and clearly separate your static assets (.SWFs, CSS, JS, etc. etc.) which can (and should) be cached, and dynamic pages (or AJAX) with current game state data which changes too frequently to bother about caching it (and which will likely go directly from your web servers) [Zubek2010].

At the back-end, the situation is significantly more complicated. According to [Zubek2016], for games you will often want not only to use your back-end cache as a cache to reduce number of DB reads, but also will want to make it a write-back cache (!), to reduce the number of DB writes. Such a write-back cache can be implemented either manually over memcached (with web servers writing to memcached only, and a separate daemon writing 'dirty' pages from memcached to DB), or a product such as Redis or Couchbase (formerly Membase) can be used [Zubek2016].

CAS

Compare-And-Swap is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of

Taming DB Load: Write-Back Caches and In-Memory States

"One Big

One Big Advantage of having write-back cache (and of the in-memory state of Classical deployment architecture described below) is related to the huge reduction in number of DB updates. For example, if we'd need to save each and every click on the simulated farm with

Advantage of having write-back cache (and of the in-memory state of Classical deployment architecture described below) is related to the huge reduction in number of DB updates.

> **that memory location to a given new value.**
>
> *— Wikipedia —*

25M daily users (each coming twice a day and doing 50 modifying-farm-state clicks each time in a 5-minute session), we could easily end up with 2.5 billion DB transactions/day (which is infeasible, or at least non-affordable). On the other hand, if we're keeping write-back cache, we can write the cache into DB only once per 10 minutes, we'd reduce the number of DB transactions 50-fold, bringing it to much more manageable 50 million/day.

For faster-paced games (usually implemented as a Classical Architecture described below, but facing the same challenge of DB being overloaded), the problem surfaces even earlier. For example, to write each and every movement of every character in an MMORPG, we'd have a flow of updates of the order of 10 DB-transactions/sec/player (i.e. for 10'000 simultaneous players we'd have 100'000 DB transactions/second, or around 10 billion DB transactions/day, once again making it infeasible, or at the very least non-affordable). On the other hand, with in-memory states stored in-memory-only (and saving to DB only major events such as changing zones, or obtaining level) – we can reduce the number of DB transactions by 3-4 orders of magnitude, bringing it down to much more manageable 1M-10M transactions/day.

As an additional benefit, such write-back caches (as long as you control write times yourself) and in-memory states also tend to play well with handling server failures. In short: for multi-player games, if you disrupt a multi-player "game event" (such as match, hand, or fight) for more than a few seconds, you won't be able to continue it anyway because you won't be able to get all of your players back; therefore, you'll need to roll your "game event" back, and in-memory states provide a very natural way of doing it. See "Failure Modes & Effects" section below for detailed discussion of failure modes under Classical Game Architecture.

A word of caution for stock exchanges. If your game is a stock exchange, you generally do need to save everything in DB (to ensure strict correctness even in case of Game Server loss), so in-memory-only states are not an option, and DB savings do not apply. However, even for stock exchanges at least Classical Game architecture described below has been observed to work very well despite DB transaction numbers being rather large; on the other hand, for stock exchanges transaction numbers are usually not that high as for MMORPG, and price of the hardware is generally less of a problem than for other types of games.

Write-Back Caches: Locking

As always, having a write-back cache has some very serious implications, and will

cause lots of problems whenever two of your players try to interact with the same cached object. To deal with it, there are three main approaches: "optimistic locking", "pessimistic locking", and transactions. Let's consider them one by one.

Optimistic Locking. This one is directly based on memcached's CAS operation.[1] The idea of using CAS for optimistic locking goes along the following lines. To process some incoming request, Web Server does the following:

- reads whole "game world" state as a single blob from memcached, alongside with "cas token". "cas token" is a thing which is actually a "version number" for this object.
- we're optimists! 🙂 so Web Server is processing incoming request ignoring possibility that some other Web Server also got the same "game world" and is working on it
  - Web Server is NOT allowed to send any kind of reply back to user (yet)
- Web Server issues cas operation with both new-value-of-"game-world"-blob, and the same "cas token" which it has received
  - if "cas token" is still valid (i.e. nobody has written to the blob before current Web Server has read it), memcached writes new value, and returns ok.
    - Then our Web Server may send reply back to whoever-requested-it
  - if, however, there was a second Web Server which has managed to write after we've read our blob – memcached will return a special error
    - in this case, our Web Server MUST discard all the prepared replies
    - in addition, it MAY read new value of "game world" state (with new "cas token"), and try to re-apply incoming request to it
      - this is perfectly valid: it is just "as if" incoming request has came a little bit later (which can always happen)

Optimistic locking is simple, is lock-less (which is important, see below why), and has only one significant drawback for our purposes. That is, while it works fine as long as collision probability (i.e. two Web Servers working on the same "game world" at the same time) is low, but as soon as probability grows (beyond, say 10%) – you will start getting a significant performance hit (for processing the same message twice, three times, and so on and so forth). For slow-paced asynchronous games it is very unlikely to become a problem, and therefore by default I'd recommend optimistic locking for web-based games, but you still need to understand limitations of the technology before using it.

---

[1] a supposedly equivalent optimistic locking for Redis is described in [Redis.CAS]

Pessimistic Locking. This is pretty much a classical multi-threaded mutex-based locking, applied to our "how to handle two concurrent actions from two different Web Servers over the same "game world" problem.

In this case, game state (usually stored as a whole in a blob) is protected by a sorta-mutex (so that two web servers cannot access it concurrently). Such a mutex can be implemented, for example, over something like memcached's CAS operation [Zubek2010]. For pessimistic locking, Web Server acts as follows:

- obtains lock on mutex, associated with our "game world" (we're pessimists ☹ , so we need to be 100% sure before processing, that we're not processing in vain).
  - if mutex cannot be obtained – Web Server MAY try again after waiting a bit
- reads "game world" state blob
- processes it
- writes "game world" state blob
- releases lock on mutex

This is a classical mutex-based schema and it is very robust when applied to classical multi-thread synchronization. However, when applying it to web servers and memcached, there is a pretty bad caveat ☹ . The problem here is related to "how to detect hanged/crashed web server – or process – which didn't remove the lock" question, as such a lock will effectively prevent all future legitimate interactions with the locked game world (which reminds me of the nasty problems from the early-90ish pre-SQL FoxPro-like file-lock-based databases).

For practical purposes, such a problem can be resolved via timeouts, effectively breaking the lock on mutex (so that if original mutex owner of the broken mutex comes later, he just gets an error). However, allowing to break mutex locks on timeouts, in turn, has significant further implications, which are not typical for usual mutex-based inter-thread synchronizations:

- first, if we're breaking mutex on timeout – there is a problem of choosing the timeout. Have it too low, and we can end up with fake timeouts, and having it too high will cause frustrated users
- second, it implies that we're working EXACTLY according to the pattern above. In particular:
  - having more than one memcached object per "game world" is not allowed
  - "partially correct" writes of "game state" are not allowed either, even if they're intended to be replaced "very soon" under the same lock

In practice, these issues are rarely causing too much problems when using memcached for mutex-based pessimistic locking. On the other hand, as for memcached we'd need to simulate mutex over CAS, I still suggest optimistic locking (just because it is simpler and causes less memcached interactions).

Transactions. Classical DB transactions are useful, but dealing with concurrent transactions is really messy. All those transaction isolation levels (with interpretations subtly different across different databases), locks, and deadlocks are not a thing which you really want to think about.

Fortunately, Redis transactions are completely unlike classical DB transactions and are coming without all this burden. In fact, Redis transaction is merely a sequence of operations which are executed atomically. It means no locking, and an ability to split your "game world" state into several parts to deal with traffic. On the other hand, I'd rather suggest to stay away from this additional complexity as long as possible, using Redis transactions only as means of optimistic locking as described in [Redis.CAS]. Another way of utilizing capabilities of Redis transactions is briefly mentioned in "Web-Based Deployment Architecture: FSMs" section below.

Web-Based Deployment Architecture: FSMs

You may ask: how finite state machines (FSMs) can possibly be related to the web-based stuff? They seem to be different as night and day, don't they?

Actually, they're not. Let's take a look at both optimistic and pessimistic locking above. Both are taking the whole state, generating new state out of it, and storing this new state. But this is exactly what our FSM::process_event() function from Chapter V does! In other words, even for web-based architecture, we can (and IMHO SHOULD) write processing in an event-driven manner, taking state and processing inputs, producing state and issuing replies as a result.

As soon as we've done it this way, the question "Should we use optimistic locking or pessimistic one", becomes a deployment implementation detail

In other words, if we have an FSM-based (a.k.a. event-driven) game code, we can change the wrapping infrastructure code around it, and switch it from optimistic locking to pessimistic one (or vice versa). All this without changing a single line *within* any of FSMs!

Moreover, if using FSMs, we can even change from Web-Based Architecture to Classical one and vice versa without changing FSM code

If by any chance reading the whole "game world" state from cache becomes a problem (which it shouldn't, but you never know), it MIGHT still be solved via FSMs together with Redis-style transactions mentioned above. Infrastructure code (the one outside of FSM) may, for example, load only a part of the "game world" state depending on type of input request (while locking all the other parts of the state to avoid synchronization problems), and also MAY implement some kind on-demand exception-based state loading along the lines of on-demand input loading discussed in [[TODO]] section below.

Web-Based Deployment Architecture: Merits

Unlike the naïve approach above, Web-Based systems may work. Their obvious advantage (especially if you have a bunch of experienced web developers on your team) is that it uses familiar and readily-available technologies. Other benefits are also available, such as:

- easy-to-find developers
- simplicity and being relatively obvious (that is, until you need to deal with locks, see above)
- web servers are stateless (except for caching, see below), so failure analysis is trivial: if one of your web servers goes down, it can be simply replaced
- can be easily used both for the games with downloadable client and for browser-based ones

Web-Based Architecture (as well as any other one), of course, also has downsides, though they may or may not matter depending on your game:

- there is no way out of web-based architecture; once you're in – switching to any other one will be impossible. Might be not that important for you, but keep it in mind.
- it is pretty much HTTP-only (with an option to use Websockets); migration to plain TCP/UDP is generally not feasible.
- as everything will work via operations on the whole game state, different parts of your game will tend to be tightly coupled. Not a big problem if your game is trivial, but may start to bite as complexity grows.
- as the number of interactions between players and game world grows, Web-Based Architecture becomes less and less efficient (as distributed-mutex-locked accesses to retrieve whole game state from the back-end cache and write it back as a whole, don't scale well). Even medium-paced "synchronous" games such as casino multi-players, are usually not good candidates for Web-Based Architecture.
- you need to remember to keep all the accesses to game objects synchronized;

if you miss one – it will work for a while, but will cause very strange-looking bugs under heavier load.

- you'll need to spend A LOT of time meditating over your caching strategy. As the number of player grows, you're very likely to need a LOT of caching, so start designing your caching strategies ASAP. See above about peculiarities of caching when applied to games (especially on write-back part and mutexes), and make your own research.
- as the load grows, you will be forced to spend time on finding a good and really-working-for-you solution for that nasty web-server-never-releases-mutex problem mentioned above. While not that hopeless as ensuring consistency within pre-SQL DBF-like file-lock-based databases, expect quite a chunk of trouble until you get it right.

Still,

if your game is rather slow/asynchronous and inter-player interactions are simple and rather far between, Web-Based Architecture may be the way to go

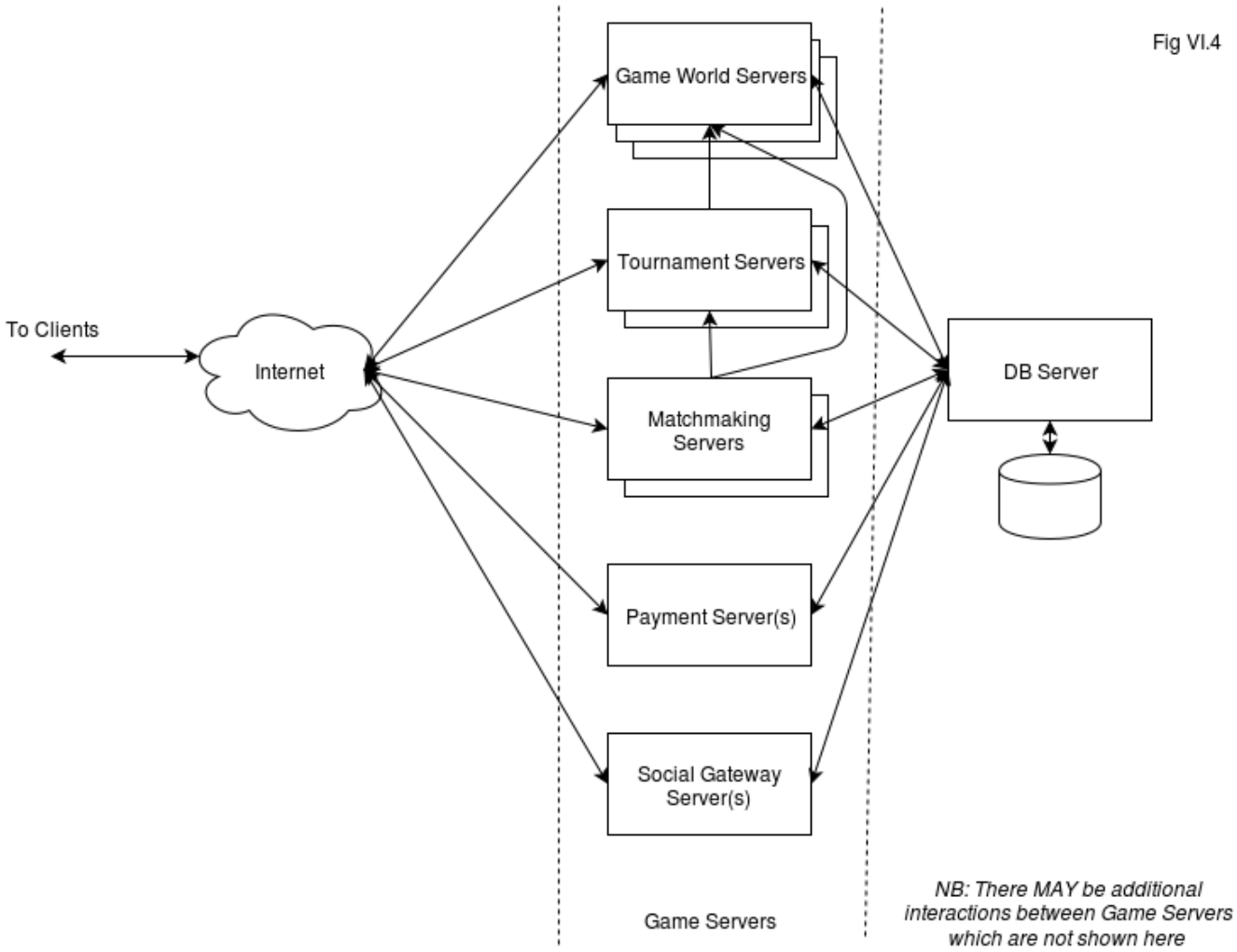
While Classical Architecture described below (especially with Front-End Servers added, see [[TODO]] section) can also be used for slow-paced games, implementing it yourself just for this purpose is a Really Big Headache and might be easily not worth the trouble if you can get away with Web-Based one. On the other hand,

even for medium-paced synchronous multi-player games (such as casino-like multi-player games) Web-Based Architecture is usually not a good candidate

(see above).

Classical Game Deployment Architecture

Fig VI.4 shows a classical game deployment diagram.



In this deployment architecture, clients are connected to Game Servers directly, and Game Servers are connected to a single DB Server, which hosts system-wide persistent state. Each of Game Servers MIGHT (or might not) have it own database (or other persistent storage) depending on the needs of your specific game; however, usually Game Servers store only in-memory states with all the persistent storage going into a single DB residing on DB Server.

Game Servers

Game Servers are traditionally divided according to their functionality, and while you can combine different types of functionality on the same box, there are often good reasons to avoid combining too many different things together.

Different types of Game Servers (more strictly – different types of functionality hosted on Game Servers) should be mapped to the entities on your Entities&Relationships Diagram described in Chapter II. You should do this mapping for your specific game yourself. However, as an example, let's take a look at a few of *typical* Game Servers (while as always, YMMV, these are likely to be present for quite a few games):

Game World Servers. Your game worlds are running on Game World Servers, plain and simple. Note that "Game World" here doesn't necessarily mean a "3D

game world with simulated physics etc.". Taking a page from a casino-like games book, "Game World" can be a casino table; going even further into realm of stock exchanges, "Game World" may be a stock exchange floor. Surprisingly, from an architecture point of view, all these seemingly different things are very similar. All of them represent a certain state (we usually name it "game world") which is affected by player's actions in real time, and changes to this state are shown to all the players.[2]

"Usually, when a player launches her client app, the client by default connects to one of Matchmaking Servers.

Matchmaking Servers. Usually, when a player launches her client app, the client by default connects to one of Matchmaking Servers. In general, matchmaking servers are responsible for redirecting players to one of your multiple game worlds. In practice, they can be pretty much anything: from lobbies where players can join teams or select game worlds, to completely automated matchmaking. Usually it is matchmaking servers that are responsible for creating new game worlds, and placing them on the servers (and sometimes even creating new servers in cloud environments).

Tournament Servers. Not always, but quite often your game will include certain types of "tournaments", which can be defined as game-related entities that have their own life span and may create multiple Game World instances during this life span. Technically, these are usually reminiscent of Matchmaking Servers (they need to communicate with players, they need to create Game Worlds, they tend to use about the same generic protocol synchronization mechanics, see Chapter [[TODO]] for details), but of course, Tournament Servers need to implement tournament rules of the specific tournament etc. etc.

Payment Server and Social Gateway Server. These are necessary to provide interaction of your game with the real world. While these server might look an "optional thing nobody should care about", they're usually playing an all-important role in increasing popularity of your game and monetization, so you'd better to account for them from the very beginning.

"Payment Server and

The very nature of Payment Servers and Social Gateway Server is to be "gateways to the real world", so they're usually exactly what is written on the tin: gateways. It means that their primary function is usually to get some kind of input from the player and/or other Game Servers, write something to DB (via DB Server), and make some request according to some-external-protocol (defined by payment provider or by social network). On the other hand, implementing them when you need to support multiple payment/social providers (each

Social Gateway Server are necessary to provide interaction of your game with the real world.

with their own peculiarities, you can count on it) – is a challenge; also they tend to change a lot due to requirements coming from business and marketing, changes in provider's APIs, need to support new providers etc. And of course, at least for payment servers, there are questions of distributed transactions between your DB and payment-provider DB, with all the associated issues of recovery from "unknown-state" transactions, and semi-manul reconciliation of reports at the end of month. As a result, these two seemingly irrelevant-to-gameplay servers tend to have their own teams after deployment; more details on payment servers will be discussed in Chapter [[TODO]].

One of the things these servers should do, is isolating Game World Servers and preferably Matchmaking Servers from the intimate details about specifics of the payment providers and social networks. In other words, Game World Servers shouldn't generally know about such things as "a guy has made a post of Facebook, so we need to give him bonus of 25% extra experience for 2 days". Instead, this functionality should be split in two: Social Gateway Server should say "this guy has earned bonus X" (with explanation in DB why he's got the bonus, for audit purposes), and Game World Server should take "this guy has bonus X" statement and translate it into 25% extra experience.

---

[2] restrictions may apply to which parts of the state are shown to which players. One such example is a server-side fog-of-war, that we'll discuss in Chapter [[TODO]]

Implementing Game Servers under QnFSM architecture

In theory, Game Servers can be implemented in whatever way you prefer. In practice, however, I strongly suggest to have them implemented under Queues-and-FSMs (QnFSM) model described in Chapter V. Among the other things, QnFSM provides very clean separation between different modules, enables replay-based debug and production post-mortem, allows for different deployment scenarios without changing the FSM code (this one becomes quite important for the server side), and completely avoids all those pesky inter-thread synchronization problems at logical level; see Chapter V for further discussion of QnFSM benefits.

Fig VI.5 shows a diagram with an implementation of a generic Game Server under QnFSM:

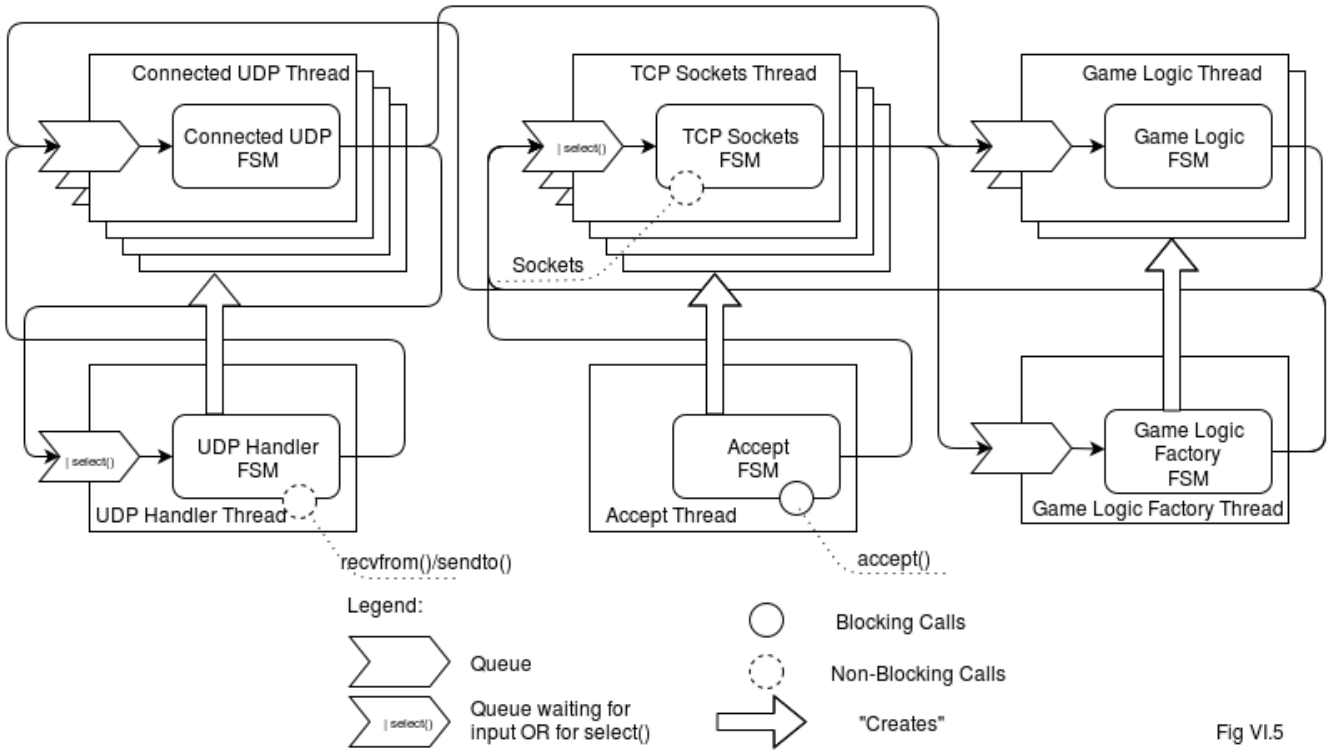


Fig VI.5

If it looks complicated at the first glance – well, it should. First of all, the diagram represents quite a generic case, and for your specific game (and at least at first stages) you may not need all of that stuff, we'll discuss it below. Second, but certainly not unimportant, writing anywhere-close-to-scalable server is not easy.

Now let's take a closer look at the diagram on Fig VI.5, going in an unusual direction from right to left.

Game Logic and Game Logic Factory. On the rightmost side of the diagram, there is the most interesting part – things, closely related to your game logic. Specifics of those Game Logic FSMs are different for different Game Servers you have, and can vary from "Game World FSM" to "Payment Processing FSM" with anything else you need in between. It is worth noting that while for most Game Logic FSMs you won't need any communications with the outside world except for sending/receiving messages (as shown on the diagram), for gateway-style FSMs (such as Payment FSM or Social Gateway FSM) you will need some kind of external API (most of the time they go over outgoing HTTP, though I've seen quite strange things, such as X.25); it doesn't change the nature of those gateway-style FSMs, so you still have all the FSM goodies (as long as you "intercept" all the calls to that external API, see Chapter V for details). [[TODO! – discussion on blocking-vs-non-blocking APIs for gateway-style FSMs]]

"When a Matchmaking server needs to create a new game world on server X, it sends a request to the Game Logic

Game Logic Factory is necessary to create new FSMs (and if

necessary, new threads) by an external request. For example, when a Matchmaking server needs to create a new game world on server X, it sends a request to the Game Logic Factory which resides on server X, and Game Logic Factory creates game world with requested parameters. Deployment-wise, usually there is only one instance of the Game Logic Factory per server, but technically there is no such strict requirement.

Factory which resides on server X, and Game Logic Factory creates game world with requested parameters.

TCP Sockets and TCP Accept. Going to the left of Game Logic on Fig VI.5, we can see TCP-related stuff. Here the things are relatively simple: we have classical accept() thread, that passes the accepted sockets to Socket Threads (creating Socket Threads when it becomes necessary).

The only really important thing to be noted here is that each Socket Thread[3] should normally handle more than one TCP socket; usually number of TCP sockets per thread for a game server should be somewhere between 16 and 128 (or "somewhere between 10 and 100" if you prefer decimal notation to hex). On Windows, if you're using WaitForMultipleObjects()[4], you're likely to hit the wall at around 30 sockets per thread (see further discussion in Chapter [[TODO]]), and this has been observed to work perfectly fine. Having one thread (even worse – two, one for recv() and another one for send()) per socket on the server-side is generally not advisable, as threads have substantial associated overhead (both in terms of resources, and in terms of context switches). In theory, multiple sockets per thread may cause additional latencies and jitter, but in practice for a reasonably well written code running on a non-overloaded server I wouldn't expect additional latencies and jitter of more than single-digit microseconds, which should be non-observable even for the most fast-paced games.

---

[3] and accordingly, Socket FSM, unless you're hosting multiple Socket FSMs per Socket Thread, which is also possible

[4] which IMHO provides the best balance between performance and implementation complexity (that is, if you need to run your servers on Windows), see Chapter [[TODO]] for further details

UDP-related FSMs. UDP (shown on the left side of Fig VI.5) is quite a weird beast; in some cases, you can use really simple things to get UDP working, but in some other cases (especially when high performance is involved), you may need to resort to quite heavy solutions to achieve scalability. The solution on Fig VI.5 is on the simpler side, so you MIGHT need to get into more complicated things to achieve performance/scalability (see below).

Let's start explaining things here. One problem which you [almost?] universally will

have when using UDP, is that you will need to know whether your player is connected or not. And as soon as you have a concept of "UDP connection" (for example, provided by your "reliable UDP" library), you have some kind of connection state/context that needs to be stored somewhere. This is where those "Connected UDP Threads" come in.

KISS principle

KISS is an acronym for 'Keep it simple, stupid' as a design principle noted by the U.S. Navy in 1960.

— Wikipedia —

So, as soon as we have the concept of "player connected to our server" (and we need this concept at least because players need to be subscribed to the updates from our server), we need those "Connected UDP Threads". Not exactly the best start from KISS point of view, but at least we know what we need them for. As for the number of those threads – we should limit the number of UDP connections per Connected UDP Thread; as a starting point, we can use the same ballpark numbers of UDP connections per thread as we were using for TCP sockets per thread: that is, between 16-128 UDP connections per thread.

UDP Handler Thread and FSM is a very simple thing – it merely gets whatever-comes-in-from-recvfrom(), and passes it to an appropriate Connected UDP Thread (as UDP Handler FSM also creates those Connected UDP Threads, it is not a problem for it to have a map of incoming-packet-IP/port-pairs to threads).

"You MAY find that your UDP Handler Thread becomes a bottleneck, causing incoming packets to drop

However, you MAY find that this simpler approach doesn't work for you (and your UDP Handler Thread becomes a bottleneck, causing incoming packets to drop while your server is not overloaded yet); in this case, you'll need to use platform-specific stuff such as recvmmsg(),[5] or to use multiple recvfrom()/sendto() threads. The latter multi-threaded approach will in turn cause a question "where to store this mapping of incoming-packet-IP/port-pairs to threads". This can be addressed either using shared state (which is a deviation from pure FSM model, but in this particular case it won't cause too much trouble in practice), or via separate UDP Factory Thread/FSM (with UDP Factory FSM storing the mapping, and notifying recvfrom() threads about the mapping on request, in a manner somewhat similar to the one used for Routing Factory FSM described in [[TODO]] section below).

[5] see further discussion on recvmmsg() in Chapter [[TODO]]

Websocket-related FSMs and HTTP-related FSMs (not shown). If you need to

support Websocket clients (or, Stevens forbid, HTTP clients) in addition to, or instead of TCP or UDP, this can be implemented quite easily. Basic Websocket protocol is very simple (with basic HTTP being even simpler), so you can use pretty much the same FSMs as for TCP, but implementing additional header parsing and frame logic within your Websocket FSMs. If you think you need to support HTTP protocol for a synchronous game – think again, as implementing interactive communications over request-response HTTP is difficult (and tends to cause too much server load), so Websockets are generally preferable over HTTP for synchronous games and are providing about-the-same (though not identical) benefits in terms of browser support and being firewall friendly; see further discussion on these protocols in Chapter [[TODO]]. For asynchronous games, HTTP (with simple polling) MAY be a reasonable choice.

CUDA/OpenCL/Phi FSM (not shown). If your Game Worlds require simulation which is very computationally heavy, you may want to use your Game World servers with CUDA (or OpenCL/Phi) hardware, and to add another FSM (not shown on Fig VI.5) to communicate with CUDA/OpenCL/Phi GPGPU. A few things to note in this regard:

- We won't discuss how to apply CUDA/OpenCL/Phi to your simulation; this is your game and a question "how to use massively parallel computations for your specific simulation" is utterly out of scope of the present book.
- Obtaining strict determinism for CUDA/OpenCL FSMs is not trivial due to potential inter-thread interactions which may, for example, change the order of floating-point additions which may lead to rounding-related differences in the last digit (with both results practically the same, but technically different). However, for most of gaming purposes (except for replaying server-side simulation forever-and-ever on all the clients), even this "almost-strict-determinism" may be sufficient. For example, for "recovery via replay" feature discussed in "Complete Recovery from Game World server failures: DIY Fault-Tolerance in QnFSM World" section below, results during replay-since-last-state-snapshot, while not guaranteed to be *exactly* the same, are not too likely to result in macroscopic changes which are too visible to players.
- Normally, you're not going to ship your game servers to your datacenter. Well, if the life of your game depends on it, you might, but this is a huuuge headache (see below, as well as Chapter [[TODO]] for further discussion)
    - As soon as you agree that it is not your servers, but leased ones or cloud ones (see also Chapter [[TODO]]), it means that you're completely dependent on your server ISP/CSP on supporting whatever you need.

CSP
Cloud Service Provider

    - Most likely, with 3rd-party ISP/CSP it will be Tesla or GRID GPU (both by NVidia), so in this case you should be ok with CUDA rather than OpenCL.
    - The choice of such ISPs which can lease you GPUs, is limited, and they

tend to be on an expensive side :-(. As of the end of 2015, the best I was able to find was Tesla K80 GPU (the one with 4992 cores) rented at $500/month (up to two K80's per server, with the server itself going at $750/month). With cloud-based GPUs, things weren't any better, and started from around $350/month for a GRID K340 (the one with 4×384=1536 total cores). Ouch!

- If you *are* going to co-locate your servers instead of leasing them from ISP[6], you should still realize that server-oriented NVidia Tesla GPUs (as well as AMD FirePro S designated for servers) are damn expensive. For example, as of the end of 2015, Tesla K80 costs around $4000(!); at this price, you get 2xGK210 cores, 24GB RAM@5GHz, clock of 562/875MHz, and 4992 CUDA cores. At the same time, desktop-class GeForce Titan X is available for about $1100, has 2 of newer GM200 cores, 12GB RAM@7GHz, clock of 1002/1089MHz, and 3072 CUDA cores. In short – Titan X gets you more or less comparable performance parameters (except for RAM size and double-precision calculations) at less than 30% of the price of Tesla K80. It might look as a no-brainer to use desktop-class GPUs, but there are several significant things to keep in mind:
    - the numbers above are *not* directly comparable; make sure to test your specific simulation with different cards before making a decision. In particular, differences due to RAM size a double-precision maths can be very nasty depending on specifics of your code

"In short – Titan X gets you more or less comparable performance parameters (except for RAM size and double-precision calculations) at less than 30% of the price of Tesla K80.

    - even if you're assembling your servers yourself, you are still going to place your servers into a 3rd-party datacenter; hosting stuff within your office is not an option (see Chapter [[TODO]])
        - space in datacenters costs, and costs a lot. It means that tower servers, even if allowed, are damn expensive. In turn, it usually means that you need a "rack" server.
        - Usually, you cannot just push a desktop-class GPU card (especially a card such as Titan X) into your usual 1U/2U "rack" server; even if it fits physically, in most cases it won't be able to run properly because of overheating. Feel free to try, and *maybe* you will find the card which runs ok, but don't expect it to be the-latest-greatest one; thermal conditions within "rack" servers are extremely tight, and air flows are traditionally very different from the desktop servers, so throwing in additional 250W or so with a desktop-oriented air flow to a non-GPU-optimized server isn't likely to work for more than a few minutes.

- IMHO, your best bet would be to buy rack servers which are specially designated as "GPU-optimized", *and ideally – explicitly supporting those GPUs that you're going to use.* Examples of rack-servers-supporting-desktop-class-GPUs range from[7] 1U server by Supermicro with up 4x Titan X cards,[8] to 4U boxes with up to 8x Titan X cards, and monsters such as 12U multi-node "cluster" which includes total of 10×6-core Xeons and 16x GTX 980, the whole thing going at humble $40K total, by ExxactCorp. In any case, before investing a lot to buy dozens of specific servers, make sure to load-test them, and *load-test a lot* to make sure that they won't overheat under many hours of heavy load and datacenter-class thermal conditions (where you have 42 such 1U servers with one lying right on top of each other, ouch!, see Chapter [[TODO]] for further details).

To summarize: if your game cannot survive without server-side GPGPU simulations – it can be done, but be prepared to pay *a lot more than you would expect based on desktop GPU prices*, and keep in mind that deploying CUDA/OpenCL/Phi on servers will take much more effort than simply making your software run on your local Titan X ☹ . Also – make sure to start testing on real server rack-based hardware as early as possible, you do need to know ASAP whether hardware of your choice has any pitfalls.

"If your game cannot survive without server-side GPGPU simulations – it can be done, but be prepared to pay *a lot more than you would expect based on desktop GPU prices*

---

[6] this potentially includes even assembling them yourself, but I generally don't recommend it

[7] I didn't use any of these, so I cannot really vouch for them, but at least you, IMHO, have reasonably good chances if you try; also make sure to double-check if your colocation provider is ready to host these not-so-mainstream boxes

[8] officially Supermicro doesn't support Titans, but their 1U boxes can be bought from 3rd-party VARs such as Thinkmate with 4x Titan X for a total of $10K, Titans included; whether it really works with Titans in datacenter environment 24×7 under your type of load – you'll need to see yourself

Simplifications. Of course, if your server doesn't need to support UDP, you won't need corresponding threads and FSMs. However, keep in mind that usually your connection to DB Server SHOULD be TCP (see "On Inter-Server Communications" section below), so if your client-to-server communication is UDP, you'll usually need to implement both. On the other hand, our QnFSM architecture provides a very good separation between protocols and logic, so usually you can safely start with a TCP-only server, and this will almost-certainly be enough to test your game intra-LAN (where packet losses and latencies are negligible), and implement UDP

support later (without the need to change your FSMs). Appropriate APIs which allow this kind of clean separation, will be discussed in Chapter [[TODO]].

On Inter-Server Communications

One of the questions you will face when designing your server-side, will be about the protocol used for inter-server communications. My take on it is simple:

even if you're using UDP for client-to-server communications, seriously consider using TCP for server-to-server communications

Detailed discussion on TCP (lack of) interactivity is due in Chapter [[TODO]], but for now, let's just say that poor interactivity of TCP (when you have Nagle algorithm disabled) becomes observable only when you have packet loss, and if you have non-zero packet loss within your server LAN – you need to fire your admins.[9]

On the positive side, TCP has two significant benefits. First, if you can get acceptable latencies without disabling Nagle algorithm, TCP is likely to produce much less hardware interrupts (and overall context switches) on the receiving server's side, which in turn is likely to reduce overall load of your Game Servers and even more importantly – DB Server. Second, TCP is usually much easier to deal with than UDP (on the other hand, this may be offset if you already have implemented UDP support to handle client-to-server communications).

---

[9] to those asking "if it is zero packet loss, why would we need to use TCP at all?" – I'll note that when I'm speaking about "zero packet loss", I can't rule out two packet lost in a day which can happen even if your system is really really well-built. And while a-few-dozen-microsecond additional delay twice a day won't be noticeable, crashing twice a day is not too good

QnFSM on Server Side: Flexibility and Deployment-Time/Run-Time Options.

When it comes to the available deployment options, QnFSM is an extremely flexible architecture. Let's discuss your deployment and run-time options provided by QnFSM in more detail.

Threads and Processes

First of all, you can have your FSMs deployed in different configurations depending on your needs. In particular, FSMs can be deployed as multiple-FSMs-per-thread, one-FSM-per-thread-multiple-threads-per-process, or one-FSM-per-process configurations (all this without changing your FSM code at all).[10]

"FSMs can be deployed as multiple-FSMs-per-thread, one-FSM-per-thread-multiple-threads-per-process, or one-FSM-per-process configurations (all this without changing your FSM code at all)

In one real-world system with hundreds of thousands simultaneous players but lightweight processing on the server-side and rather high acceptable latencies, they've decided to have some of game worlds (those for novice players) deployed as multiple-FSMs-per-thread, another bunch of game worlds (intended for mature players) – deployed as a single-FSM-per-thread (improving latencies a bit, and providing an option to raise thread priority for these FSMs), and those game worlds for pro players – as a single-FSM-per-process (additionally improving memory isolation in case of problems, and practically-unobservedly improving memory locality and therefore performance); all these FSMs were using absolutely very same FSM code, but it was compiled into different executables to provide slightly different performance properties.

Moreover, in really extreme cases (like "we're running a Tournament of the Year with live players"), you may even pin a single-FSM-per-thread to a single core (preferably the same where interrupts from you NIC come on this server) and to pin other processes to other cores, keeping your latencies to the absolute minimum.[11]

---

[10] Restrictions apply, batteries not included. If you have blocking calls from within your FSM, which is common for DB-style FSMs and some of gateway-style FSMs, you shouldn't deploy multiple-FSMs-per-thread

[11] yes, this will further reduce latencies in addition to any benefits obtained by simple increase of thread priority, because of per-core caches being intact

Communication as an Implementation Detail

With QnFSM, communication becomes an implementation detail. For example, you can have the same Game Logic FSM to serve both TCP and UDP. Not only it can come handy for testing purposes, but also may enable some of your players (those who cannot access your servers via UDP due to firewalls/weird routers etc.) to play over TCP, while the rest are playing over UDP. Whether you want this capability (and whether you want to match TCP players only with TCP players to

make sure nobody has an unfair advantage) is up to you, but at least QnFSM does provide you with such an option at a very limited cost.

Moving Game Worlds Around (at the cost of client reconnect)

Yet another flexibility option which QnFSM can provide (though with some additional headache, and a bit of additional latencies), is to allow moving your game worlds (or more generally – FSMs) from one server to another one. To do it, you just need to serialize your FSM on server A (see Chapter V for details on serialization), to transfer serialized state to a server's B Game Logic Factory, and to deserialize it there. Bingo! Your FSM runs on server B right from the same moment where it stopped running on server A. In practice, however, moving FSMs around is not that easy, as you'll also need to notify your clients about changed address where this moved FSM can be reached, but despite being an additional chunk of work, this is also perfectly doable if you really want it.

Online Upgrades

"Yet another two options provided by QnFSM, enable server-side software upgrades without stopping the server.

Yet another two options provided by QnFSM, enable server-side software upgrades while your system is running, without stopping the server.

The first of these options is just to start creating new game worlds using new Game Logic FSMs (while existing FSMs are still running with the old code). This works as long as changes within FSMs are minor enough so that all external inter-FSM interfaces are 100% backward compatible, and the life time of each FSM is naturally limited (so that at some point you're able to say that migration from the old code is complete).

The second of these online-upgrade options allows to upgrade FSMs while the game world is still running (via serialization – replacing the code – deserialization). This second option, however, is much more demanding than the first one, and migration problems may be difficult to identify. Therefore, severe automated testing using "replay" technique (also provided by QnFSM, see Chapter V for details) is strongly advised. Such testing should use big chunks of the real-world data, and should simulate online upgrades at the random moments of the replay.

On Importance of Flexibility

Quite often we don't realize how important flexibility is. Actually, we *rarely* realize how important it is until we run into the wall because of *lack* of flexibility. Deterministic FSMs provide *a lot* of flexibility (as well as other goodies such as post-mortem) at a relatively low development cost. That's one of the reasons why I

am positively in love with them.

DB Server

DB Server handles access to a database. This can be implemented using several very different approaches.

The first and the most obvious model is also the worst one. While in theory, it is possible to use your usual ODBC-style blocking calls to your database right from your Game Server FSMs, do yourself a favor and skip this option. It will have several significant drawbacks: from making your Game Server FSMs too tightly coupled to your DB to having blocking calls with undefined response time right in the middle of your FSM simulation (ouch!). In short – I don't know any game where this approach is appropriate.

"While in theory, it is possible to use your usual ODBC-style blocking calls to your database right from your Game Server FSMs, do yourself a favor and skip this option.

DB API and DB FSM(s)

A much better alternative (which I'm arguing for) is to have at least one FSM running on your DB server, to have your very own message-based DB API (expressed in terms of messages or non-blocking RPC calls) to communicate with it, and to keep all DB work where it belongs – on DB Server, within appropriate DB FSM(s). An additional benefit of such a separation is that you shouldn't be a DB guru to write your game logic, but you can easily have a DB guru (who's not a game logic guru) writing your DB FSM(s).

DB API exposed by DB Server's FSM(s), SHOULD NOT be plain SQL (which would violate all the decoupling we're after). Instead, your DB API SHOULD be specific to your game, and (once again) should be expressed in game terms such as "take PC Z and place it (with all it's gear) into game world #NN". All the logic to implement this request (including pre-checking that PC doesn't belong to any other game world, modifying PC's row in table of PCs to reflect the number of the world where she currently resides, and reading all PC attributes and gear to pass it back) should be done by your DB FSM(s).

In addition, all the requests in DB API MUST be atomic; no things such as "open cursor and return it back, so I can iterate on it later" are ever allowed in your DB API (neither you will really need such things, this stands in spite of whatever-your-DB-guru-may-tell-you).

As soon as you have this nice DB API tailored for your needs, you can proceed with writing your Game Server FSMs, without worrying about exact implementation of

your DB FSM(s).

Meanwhile, at the King's Castle…

As soon as we have this really nice separation between Game Server's FSMs and DB FSM(s) via your very own message-based DB API, in a sense, the implementation of DB FSM will become an implementation detail. Still, let's discuss how this small but important detail can be implemented. Here I know of two major approaches.

Single-connection approach. This approach is very simple. You have run just one FSM on your DB Server and process everything within one single DB connection:

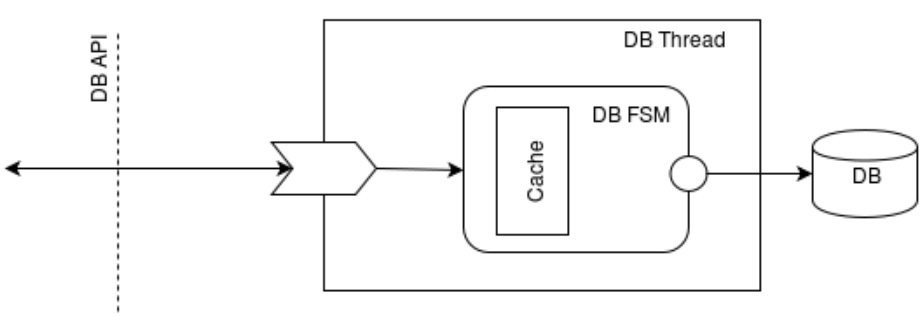


Fig VI.6

"Application-level cache has been observed to provide 10x+ performance improvement over DB cache even if all the necessary performance-related optimizations are made on the DB side

Here, there is a single DB FSM which has single DB connection (such as an ODBC connection, but there are lots of similar interfaces out there), which performs all the operations using blocking calls. A very important thing in this architecture is application-level cache, which allows to speed things up very considerably. In fact, this application-level cache has been observed to provide 10x+ performance improvement over DB cache even if all the necessary performance-related optimizations (such as prepared statements or even stored procedures) are made on the DB side. Just think about it – what is faster: simple hash-based in-memory search within your DB FSM (where you already have all the data, so we're speaking about 100 CPU clocks or so even if the data is out of L3 cache), or marshalling -> going-to-DB-side-over-IPC -> unmarshaling -> finding-execution-plan-by-prepared-statement-handle -> executing-execution-plan -> marshaling results -> going-back-to-DB-FSM-side-over-RPC -> unmarshaling results. In the latter case, we're speaking at least a few dozens of microseconds, or over 1e4 CPU clocks, over two orders of magnitude difference.[12] And with single connection to DB which is able to write data, keeping cache coherency is trivial. The main thing which gets cached for games is usually ubiquitous USERS (or PLAYERS) table, as well as some of small game-specific near-constant tables.

Despite all the benefits provided by caching, this schema clearly sounds as an heresy from any-DB-gal-out-there point of view. On the other hand, in practice it works surprisingly well (that is, as soon as you manage to convince your DB gal that you know what you're doing). I've seen such single-connection architecture[13] handling 10M+ DB transactions per day for a real-world game, and it were real transactions, with all the necessary changes, transactions, audit tables and so on.

Actually, at least at first stages of your development, I'm advocating to go with this single-connection approach.

It is very nice from many different points of view.

- First, it is damn simple.
- Second, there is no need to worry about transaction isolation levels, locks and deadlocks
- Third, it can be written as a real deterministic FSM (with all the associated goodies); moreover, this determinism stands (a) both if you "intercept calls" to DB for DB FSM itself, or (b) if we consider DB itself as a part of the FSM state, in the latter case no call interception is required for determinism.
- Fourth, the performance is very good. There are no locks whatsoever, the light is always green, so everything goes unbelievably smoothly. Add here application-level caching, and we have a winner! The single-connection system I've mentioned above, has had an *average* transaction processing time in below-1ms range (once again, with real-world transactions, commit after every transaction, etc.).

"There is no need to worry about transaction isolation levels, locks and deadlocks

The only drawback of this schema (and the one which will make DB people extremely skeptical about it, to put it very mildly) is an apparent lack of scalability. However, there are ways to modify this single-connection approach to provide virtually unlimited scalability[14] The ways to achieve DB scalability for this single-connection model will be discussed in Vol. 2.

One thing to keep in mind for this single-connection approach, is that it (at least if we're using blocking calls to DB, which is usually the case) is very sensitive to latencies between DB FSM and DB; we'll speak about it in more detail in Chapter [[TODO]], but for now let's just say that to get into any serious performance (that is, comparable to numbers above), you'll need to use RAID card with BBWC in write-

back mode[15], or something like NVMe, for the disk which stores DB log files (other disks don't really matter much). If your DB server is a cloud one, you'll need to look for the one which has low latency disk access (such things are available from quite a few cloud providers).

---

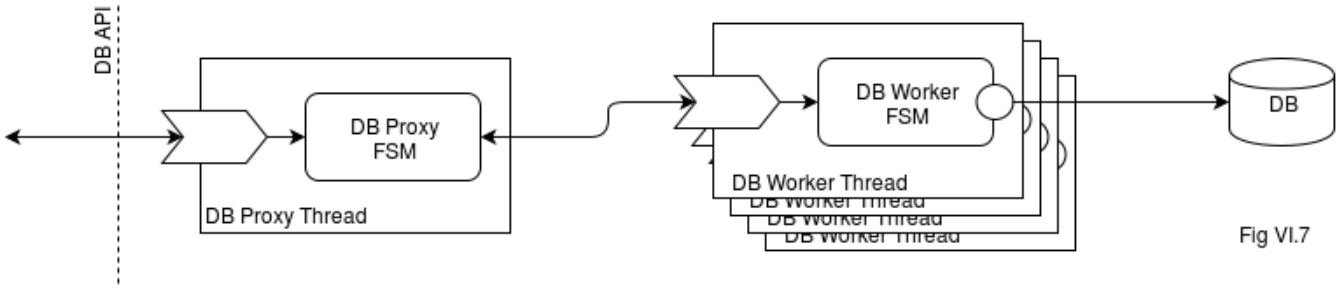
[12] with stored procedures the things become a bit better for DB side, but the performance difference is still considerable, not to mention vendor lock-in which is pretty much inevitable when using stored procedures

[13] with a full cache of PLAYERS table

[14] while in practice I've never went above around 100M DB transactions/day with this "single-connection-made-scalable" approach, I'm pretty sure that you can get to 1B pretty easily, and then it MAY become tough, as the number is too different from what-I've-seen so some unknown-as-of-now problems can start to develop. On the other hand, I daresay reaching these numbers is even more challenging with traditional multiple-connection approach

[15] don't worry, it is a perfectly safe mode for this kind of RAID, even for financial applications

Multiple-Connections approach. This approach is much more along the lines of traditional DB development, and is shown on Fig VI.7:



In short: we have one single DB-Proxy FSM (with the same DB API as discussed above),[16] which does nothing but dispatches requests to DB-Worker FSMs; each of these DB-Worker FSMs will keep its own DB connection and will issue DB requests over this connection. Number of these DB-Worker FSMs should be comparable to the number of the cores on your DB server (usually 2*number-of-cores is not bad starting number), which effectively makes this schema a kind of transaction monitor.

"The upside of

The upside of this schema is that it is inherently somewhat-scalable, but that's about it. Downsides, however, are numerous. The most concerning one is the cost of code maintenance in face of all those changes of logic, which are run in multiple connections. This inevitably leads us to a well-known but way-too-often-ignored discussion about transaction isolation levels, locks, and deadlocks at DB level. And if you don't know what it is – believe me, you Really don't

this schema is that it is inherently somewhat-scalable, but that's about it.

want to know about them. And updating DB-handling code when you have lots of concurrent access (with isolation levels above UR), is possible, but is extremely tedious. Restrictions such as "to avoid deadlocks, we must always issue all our SELECT FOR UPDATEs in the same order – the one written in blood on the wall of DB department" can be quite a headache to put it mildly.

Oh, and don't try using application-side caching for multiple-connections (i.e. even DB-Proxy SHOULD NOT be allowed to cache). While this is theoretically possible, re-ordering of replies on the way from DB to DB-Proxy make the whole thing way too complicated to be practical. While I've done such a thing myself once, and it worked without any problems (after several months of heavy replay-based testing), it was the most convoluted thing I've ever written, and I clearly don't want to repeat this experience.

But IMNSHO the worst thing about using multiple DB connections, is that while each of those DB FSMs can be made deterministic (via "call interception"), the whole DB Server cannot possibly be made deterministic (for multiple connections), period. It means that it may work perfectly under test, but fail in production while processing exactly the same sequence of requests.

Worse than that, there is a strong tendency for improper-transaction-isolation bugs to manifest themselves only under heavy load.

So, you can easily live with such a bug (for example, using SELECT instead of SELECT FOR UPDATE) quietly sitting in, but not manifesting itself until your Big Day comes, and then it crashes your site.[17] Believe me, you really don't find yourself in such a situation, it can be really (and I mean Really) unpleasant.

In a sense, working with transaction isolation levels is akin to working with threads: about the same problems with lack of determinism, bugs which appear only in production and cannot be reproduced in test environment, and so on. On the other hand, there are DB guys&gals out there who're saying that they can design a real-world multi-connection system which works under the load of 100M+ write transactions per day and never deadlocks, and I don't doubt that they can indeed do it. The thing which I'm not so sure about, is whether they really can maintain such quality of their system in face of new-code-required-twice-a-week, and I'm even less sure that you'll have such a person on your game team.

In addition, the scalability under this approach, while apparent, is never perfect (and no, those TPC-C linear-scalability numbers don't prove that linear scalability is achievable for real-world transactions). In contrast, single-connection-made-

scalable approach which we'll discuss in Vol. 2, can be extended to achieve perfect linear scalability (at least in theory).

---

[16] in particular, it means that we can rewrite our DB FSM from Single-connection to Multiple-connections without changing anything else in the system

[17] And it is not a generic "all the problems are waiting for the worst moment to happen" observation (which is actually purely perception), but a real deal. When probability of the problem depends on site load in a non-linear manner (and this is the case for transaction isolation bugs), chances of it happening for the first time exactly during your heavily advertised Event of the Year are huge.

DB Server: Bottom Line.

Unless you happen to have on your team a DB gal with real-world experience of dealing with locks, deadlocks, and transaction isolation levels for your specific DB under at least million-per-day DB write-transaction load – go for single-connection approach

If you do happen to have such a DB guru who vehemently opposes going single-connection – you can try multi-connection, at least if she's intimately familiar with SELECT-FOR-UPDATE and practical ways of avoiding deadlocks (and no, using RDBMS's built-in mechanism to detect the deadlock 10 seconds after it happens, is usually not good enough).

And in any case, stay away from any things which include SQL in your Game Server FSMs.

Failure Modes & Effects

When speaking about deployment, one all-important question which you'd better have an answer to, is the following: "What will happen if some piece of hardware fails badly?" Of course, within the scope of this book we won't be able to do a formal full-scale FMEA for an essentially unknown architecture, but at least we'll be able to give some hints in this regard.

FMEA

Failure mode and effects analysis (FMEA) was one of the first systematic techniques for failure analysis.

— Wikipedia —

Communication Failures

So, what can possibly go wrong within our deployment architecture? First of all, there are (not shown, but existing) switches (or even firewalls) residing between our servers; while these can be made redundant, their failures (or transient

software failures of the network stack on hosts) may easily cause occasional packet loss, and also (though extremely infrequently) may cause TCP disconnects on inter-server connections. Therefore, to deal with it, our Server-to-Server protocols need to account for potential channel loss and allow for guaranteed recovery after the channel is restored. Let's write this down as a requirement and remember until Chapter [[TODO]], where we will describe our protocols.

Server Failures

"Note that the stuff marked as 'High Availability', doesn't help with losing in-memory state: what we need to avoid losing in-memory state, is 'Fault-Tolerant' techniques.

In addition, of course, any of the servers can go badly wrong. There are tons of solutions out there claiming to address this kind of failures, but you should keep in mind that usually, the stuff marked as "High Availability", doesn't help with losing in-memory state: what you need *if* you want to avoid losing in-memory state, is "Fault-Tolerant" techniques (see "Server Fault Tolerance: King is Dead, Long Live the King!" section below).

Fortunately, though, for a reasonably good hardware (the one which has a reasonably good hardware monitoring, including fans, and at least having ECC and RAID, see Chapter [[TODO]] for more discussion on it), such fatal server failures are extremely rare. From my experience (and more or less consistently with manufacturer estimates), failure rate for reasonably good server boxes (such as those from one of Big Three major server vendors) is somewhere between "once-per-5-years" and "once-per-10-years", so if you'd have only one such server (and unless you're running a stock exchange), you'd be pretty much able to ignore this problem completely. However, if you have 100 servers – the failure rate goes up to "once or twice a month", which is unacceptable if such a failure leads to the whole site going down.

Therefore, at the very least you should plan to make sure that single failure of the single server doesn't bring your whole site down. BTW, most of the time it will be a Game World Server going down, as you're likely to have much more of these than the other servers, so at first stages you may concentrate on containment of Game World server failures. Also we can note that, counter-intuitively, failures of DB Server are not *that* important to deal with;[18] not because they have less impact (they do have much more impact), but because they're much less likely to happen that a failure of *one-of-Game-World-servers*.

---

[18] that is, beyond keeping a DB backup with DB logs being continuously moved to another location, see Chapter [[TODO]] for further discussion

Containment of Game World server failures

"If Game World server fails, it can be restarted from scratch, losing all the changes since last save-to-DB, but at least preserving previous results.

The very first (and very obvious) technique to minimize the impact of your Game World server failure on the whole site, is to make sure that your Game World reports relevant changes (without sending the whole state) to DB Server as soon as they occur. So that if Game World server fails, it can be restarted from scratch, losing all the changes since last save-to-DB, but at least preserving previous results. These saves-to-DB are the best to be done at some naturally arising points within your game flow.

For example, if your game is essentially a Starcraft- or Titanfall-like sequence of matches, then the end of each match represents a very natural save-to-DB point. In other words, if Game World server fails within the match – all the match data will be lost, but all the player standings will be naturally restored as of beginning of the match, which isn't too bad. In another example, for a casino-like game the end of each "hand" also represents the natural save-to-DB point.

If your gameplay is an MMORPG with continuous gameplay, then you need to find a way to save-to-DB all the major changes of the players' stats (such as "level has been gained", or "artifact has changed hands"). Then, if the Game Server crashes, you may lose the current positions of PCs within the world and a few hundred XP per player, but players will still keep all their important stats and achievements more or less preserved.

Two words of caution with regards to save-to-DB points. First,

For synchronous games, don't try to keep the whole state of your Game Worlds in DB

"If you disrupt the game-event-currently-in-progress for more than 0.5-2

Except for some rather narrow special cases (such as stock exchanges and some of slow-paced and/or "asynchronous" games as defined in Chapter I), saving all the state of your game world into DB won't work due to performance/scalability reasons (see discussion in "Taming DB Load: Write-Back Caches and In-Memory States" section above). Also keep in mind that even if you would be able to perfectly preserve the current state of the game-event-currently-in-progress (with game event being "match", "hand", or an "RPG fight") without killing your DB, there is another very big practical problem of psychological rather than technical nature. Namely, if you disrupt the game-event-

minutes, for almost-any synchronous multi-player game you won't be able to get the same players back, and will need to rollback the game event anyway.

currently-in-progress for more than 0.5-2 minutes, for almost-any synchronous multi-player game you won't be able to get the same players back, and will need to rollback the game event anyway.

For example, if you are running a bingo game with a hundred of players, and you disrupt it for 10 minutes for technical reasons, you won't be able to continue it in a manner which is fair to all the players, at the very least because you won't be able to get all that 100 players back into playing at the same time. The problem is all about numbers: for two-player game it might work, for 10+ – succeeding in getting all the players back at the same time is extremely unlikely (that is, unless the event is about a Big Cash Prize). I've personally seen a large commercial game that handled the crashes in the following way: to restore after the crash, first, it rolled forward its DB at the DB level to get perfectly correct current state, and then it rolled all the current game-events back at application level, exactly because continuing these events wasn't a viable option due to the lack of players.

Trying to keep all the state in DB is a common pitfall which arises when the guys-coming-from-single-player-casino-game-development are trying to implement something multiplayer. Once again: don't do it. While for a single-player casino game having state stored in DB is a big fat Business Requirement (and is easily doable too), for multi-player games it is neither a requirement, nor is feasible (at least because of the can't-get-the-same-players-together problem noted above). Think of Game World server failure as of direct analogy of the fire-in-brick-and-mortar-casino in the middle of the hand: the very best you can possibly do in this case is to abort the hand, return all the chips to their respective owners (as of the beginning of the hand), and to run out of the casino, just to come back later when the fire is extinguished, so you can start an all-new game with all-new players.

The second pitfall on this way is related to DB consistency issues and DB API.

Your DB API MUST enforce logical consistency

For example, if (as a part of your very own DB API) you have two DB requests, one of which says "Give PC X artifact Y", and another one "Take artifact Y from PC X", and are trying to report an occurrence of "PC X took over artifact Y from PC XX" as two separate DB requests (one "Take" and one "Give"), you're risking that in case of Game World server failure, one of these two requests will go through, and the other one won't, so artifact will get lost (or will be duplicated) as a result. Instead of using these two requests to simulate "taking over"

"You should have a special

occurrence, you should have a special DB request "PC X took over artifact Y from PC XX" (and it should be implemented as a single DB transaction within DB FSM); this way at least the consistency of the system will be preserved, so whatever happens – there is still exactly one artifact. The very same pattern MUST be followed for passing around anything of value, from casino chips to artifacts, with any other goodies in between.

DB request "PC X took over artifact Y from PC XX" (and it should be implemented as a single DB transaction within DB FSM)

Server Fault Tolerance: King is Dead, Long Live the King!

If you want to have your servers to be really fault-tolerant, there are some ways to have your cake and eat it too.

However, keep in mind, that all fall-tolerant solutions are complicated, costly, and in the games realm I generally consider them as an over-engineering (even by my standards).

Fault-Tolerant Servers: Damn Expensive

Historically, fault-tolerant systems were provided by damn-expensive hardware such as [Stratus] (I mean their hardware solutions such as ftServer; see discussion on hardware-vs-software redundancy in Chapter [[TODO]]) and [HPIntegrityNonStop] which have everything doubled (and CPUs often quadrupled(!)) to avoid all single points of failure, and these tend do work very well. But they're usually way out of game developer's reach for financial reasons, so unless your game is a stock exchange – you can pretty much forget about them.

Fault-Tolerant VMs

Fault-Tolerant VMs (such as VMWare FT feature or Xen Remus) are quite new kids on the block (for example, VMWare FT got beyond single vCPU only in 2015), but they're already working. However, there are some significant caveats. *Take everything I'm saying about fault-tolerant VMs with a really good pinch of salt, as all the technologies are new and evolving, and information is scarce; also I admit that I didn't have a chance to try these things myself* 🙁 .

When you're using a fault-tolerant VM, the Big Picture looks like this: you have two commodity servers (usually right next to each other), connect them via 10G Ethernet, run VM on one of them (the "primary" one), and when your "primary" server fails, your VM magically reappears on the "secondary" box. From what I can see, modern Fault-Tolerant VMs are using one of two technologies: "virtual lockstep" and "fast checkpoints".

Unfortunately, each of them has its own limitations.

> "Modern Fault-Tolerant VMs are using one of two technologies: 'virtual lockstep' and 'fast checkpoints'. Unfortunately, each of them has its own limitations.

Virtual Lockstep: Not Available Anymore?

The concept of virtual lockstep is very similar to our QnFSM (with the whole VM treated as FSM). Virtual lockstep takes one single-core VM, intercepts all the inputs, passes these inputs to the secondary server, and runs a copy VM there. As any other fault-tolerant technology, virtual lockstep causes additional latencies, but it *seems* to be able to restrict its appetite for additional latency to a sub-ms range, which is acceptable for most of the games out there. Virtual lockstep is the method of fault-tolerance vSphere prior to vSphere v6 was using. The downside of virtual lockstep is that it (at least as implemented by vSphere) wasn't able to support more that one core. For our QnFSMs, this single-core restriction wouldn't be *too much* of a problem, as they're single-threaded anyway (though balancing FSMs between VMs would be a headache), but there are lots of applications out there which are still heavily-multithreaded, so it was considered an unacceptable restriction. As a result, vSphere, starting from vSphere 6, has changed their fault-tolerant implementation from virtual lockstep to checkpoint-based implementation. *As of now, I don't know of any supported implementations of Virtual Lockstep* ☹ .

Checkpoint-Based Fault Tolerance: Latencies

To get around the single-core limitation, a different technique, known as "checkpoints", is used by both Xen Remus and vSphere 6+. The idea behind checkpoints is to make a kind of incremental snapshots ("checkpoints") of the full state of the system and log it to a safe location ("secondary server"). As long as you don't let anything out of your system before the coming-later "checkpoint" is committed to a secondary server, all the calculations you're making meanwhile, become inherently unobservable from the outside, so in case of "primary" server failure, it is not possible to say whether it didn't receive the incoming data at all. It means that for the world outside of your system, your system (except for the additional latency) becomes almost-indistinguishable[19] from a real fault-tolerant server such as Stratus (see above). In theory, everything looks perfect, but with VM checkpoints we *seem* to hit the wall with checkpoint frequency, which defines the minimum possible latency. On systems such as VMWare FT, and Xen Remus, checkpoint intervals are measured in dozens of milliseconds. If your game is ok with such delays – you're fine, but otherwise – you're out of luck ☹ . For more details on checkpoint-based VMs, see [Remus].

Saving for latencies (and the need to have 10G connections between servers, which is not that big deal), checkpoint-based fault tolerance has several significant advantages over virtual lockstep; these include such important things as support

for multiple CPU cores, and N+1 redundancy.

---

[19] strictly speaking, the difference can be observed as some network packets may be lost, but as packet loss is a normal occurrence, any reasonable protocol should deal with transient packet loss anyway without any observable impact

Complete Recovery from Game World server failures: DIY Fault-Tolerance in QnFSM World

If you're using FSMs (as you should anyway), you can also implement your own fault-tolerance. I should confess that I didn't try this approach myself, so despite looking very straightforward, there can be practical pitfalls which I don't see yet. Other than that, it should be as fault-tolerant as any other solution mentioned above, *and* it should provide good latencies too (well in sub-ms range).

As any other fault-tolerant solution, for games IMHO it is an over-engineering, but if I'd feel strongly about the failures causing per-game-event rollbacks, this is the one I'd try first. It is latency friendly, it allows for N+2 redundancy (saving you from doubling the number of your servers in case of 1+1 redundancy schemas), and it plays really well alongside our FSM-related stuff.

The idea here is to have separate Logging Servers logging all the events to all the FSMs residing on your Game World servers; then, you will essentially have enough information on your Logging Servers to recover from Game World server failure. More specifically, you can do the following:

- have an additional Logging Server(s) "in front of Game Servers"; these Logging Server(s) perform two functions:
    - log all the messages incoming to all Game Server FSMs
        - these include: messages coming from clients, messages coming from other Game Servers, and messages coming from DB Server
        - moreover, even communications between different FSMs residing on the same Game Server, need to go via Logging Server and need to be logged
    - timestamp all the incoming messages
- all your Game Server FSMs need to be strictly-deterministic
    - in particular, Game Server FSMs won't use their own clocks, but will use

timestamps provided by Logging Servers instead

- In addition, from time to time each of Game Server FSMs need to serialize its whole state, and report it to Logging Server

- then, we need to consider two scenarios: Logging Server failure and Game Server failure (we'll assume that they never fail simultaneously, and such an event is indeed extremely unlikely unless it is a fire-in-datacenter or something)

    - if it is Logging Server which fails, we can just replace it with another (re-provisioned) one; there is no game-critical data there

    - if it is Game Server which fails, we can re-provision it, and then roll-forward each and every FSM which was running on it, using last-reported-state and logs-saved-since-last-reported-state saved on the Logging Server. Due to the deterministic nature of all the FSMs, the restored state will be exactly the same as it was a few seconds ago[20]

        - at this point, all the clients and servers which were connected to the FSM, will experience a disconnect

        - on disconnect, the clients should automatically reconnect anyway (this needs to account for IP change, what is a medium-sized headache, but is doable; in [[TODO]] section we'll discuss Front-End servers which will isolate clients from disconnects completely)

        - issues with server-to-server messages should already be solved as described in "Communication Failures" subsection above

"if it is Game Server which fails, we can re-provision it, and then roll-forward each and every FSM which was running on it

In a sense, this "Complete Recovery" thing is conceptually similar to EventProcessorWithCircularLog from Chapter V (but with logging residing on different server, and with auto-rollforward in case of server failure), or to a traditional DB restore-and-log-rollforward.

Note that only hardware problems (and software bugs outside of your FSMs, such as OS bugs) can be addressed with this method; bugs within your FSM will be replayed and will lead to exactly the same failure 🙁 .

Last but not least, I need to re-iterate that I would object any fault-tolerant schema for most of the games out there on the basis of over-engineering, though I admit that there might be good reasons to try achieving it, especially if it is not too expensive/complicated.

---

[20] or, in case of almost-strictly-deterministic FSMs such as those CUDA-based ones, it will be almost-exactly-the-same

[[TODO!]] DIY VIrtual-Lockstep

Classical Game Deployment Architecture: Summary

To summarize the discussion above about Classical Game Deployment Architecture:

- It works
- It can and should be implemented using QnFSM model with deterministic FSMs, see discussion above for details
- Your communication with DB (DB API) SHOULD use game-specific requests, and SHOULD NOT use any SQL; all the SQL should be hidden behind your DB FSM(s)
- Your first DB Server SHOULD use single-connection approach, unless you happen to have a DB guy who has real-world experience with multi-connection systems under at least millions-per-day write(!) transaction loads
  - Even in the latter case, you SHOULD try to convince him, but if he resists, it is ok to leave him alone, as long as external DB API is still exactly the same (message-based and expressed in terms of whatever-your-game-needs). This will provide assurance that in the extreme case, you'll be able to rewrite your DB Server later.

[[To Be Continued…

This concludes beta Chapter VI(a) 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 VI(b), "Modular Architecture: Server-Side. Throwing in Front-End Servers.]]

[–] References

[Lightstreamer] http://www.lightstreamer.com/
[Redis.CAS] http://redis.io/topics/transactions#cas
[Zubek2016] Robert Zubek, "Private communications with"
[Zubek2010] Robert Zubek, "Engineering Scalable Social Games", GDC2010

[Stratus] "Stratus Technologies", Wikipedia
[HPIntegrityNonStop] "HP Integrity NonStop", Wikipedia
[Remus] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« ***Chapter V(d). Modular Architecture: Client-Side. Client Arch***...

Chapter VI(b). Server-Side Architecture. Front-End Servers a... »

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

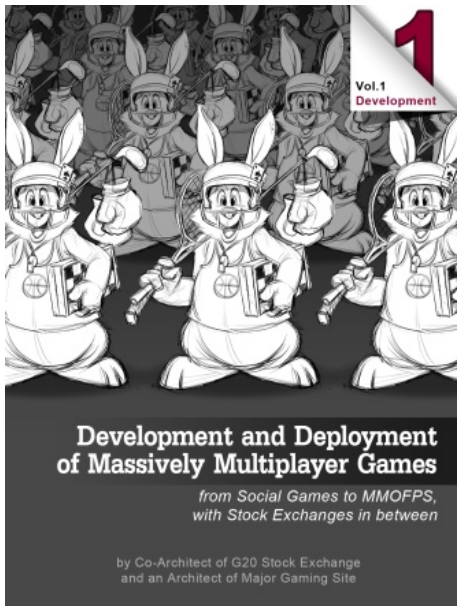
Copyright © 2014-2016 ITHare.com

Chapter VI(b). Server-Side Architecture. Front-End Servers and Client-Side Random Load Balancing

posted December 28, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

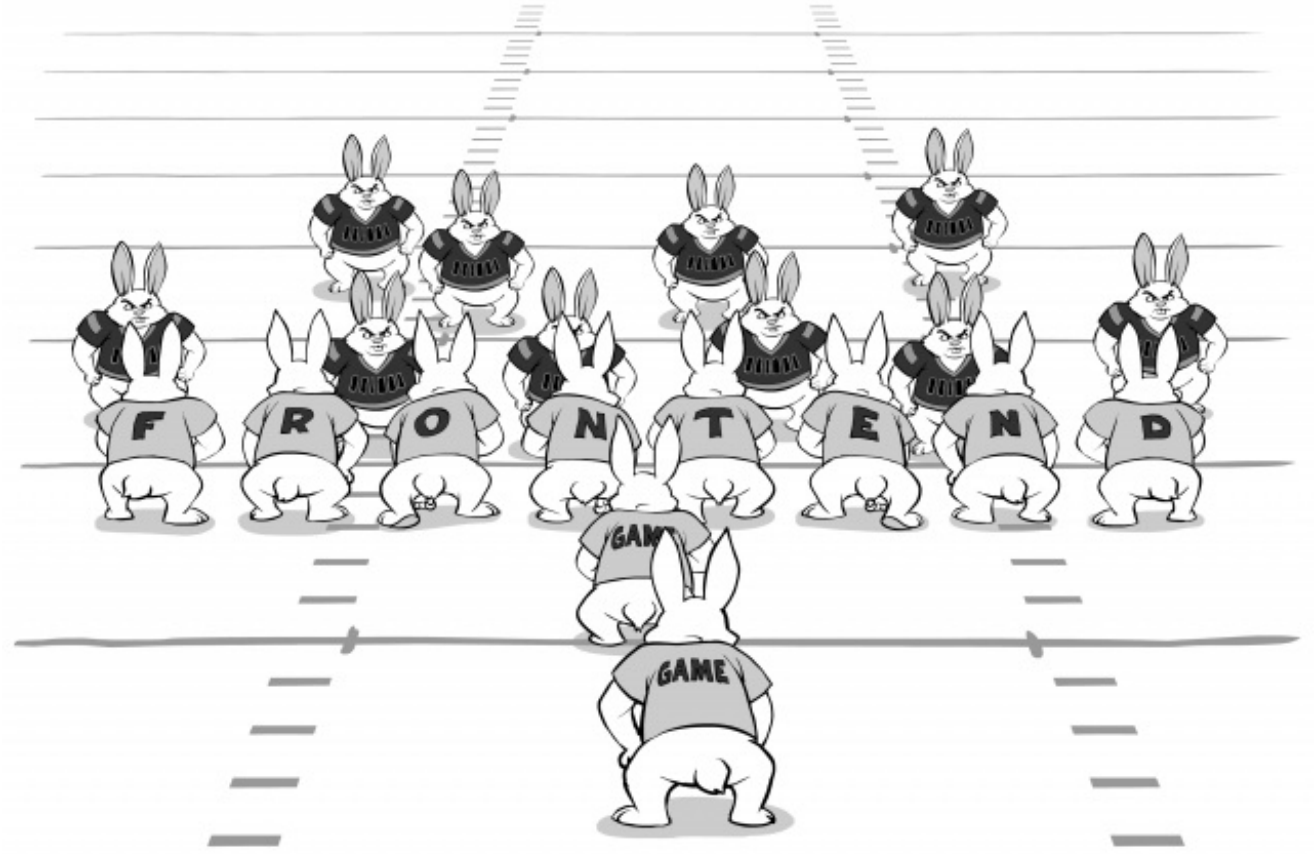
[[*This is Chapter VI(b) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further 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.]]

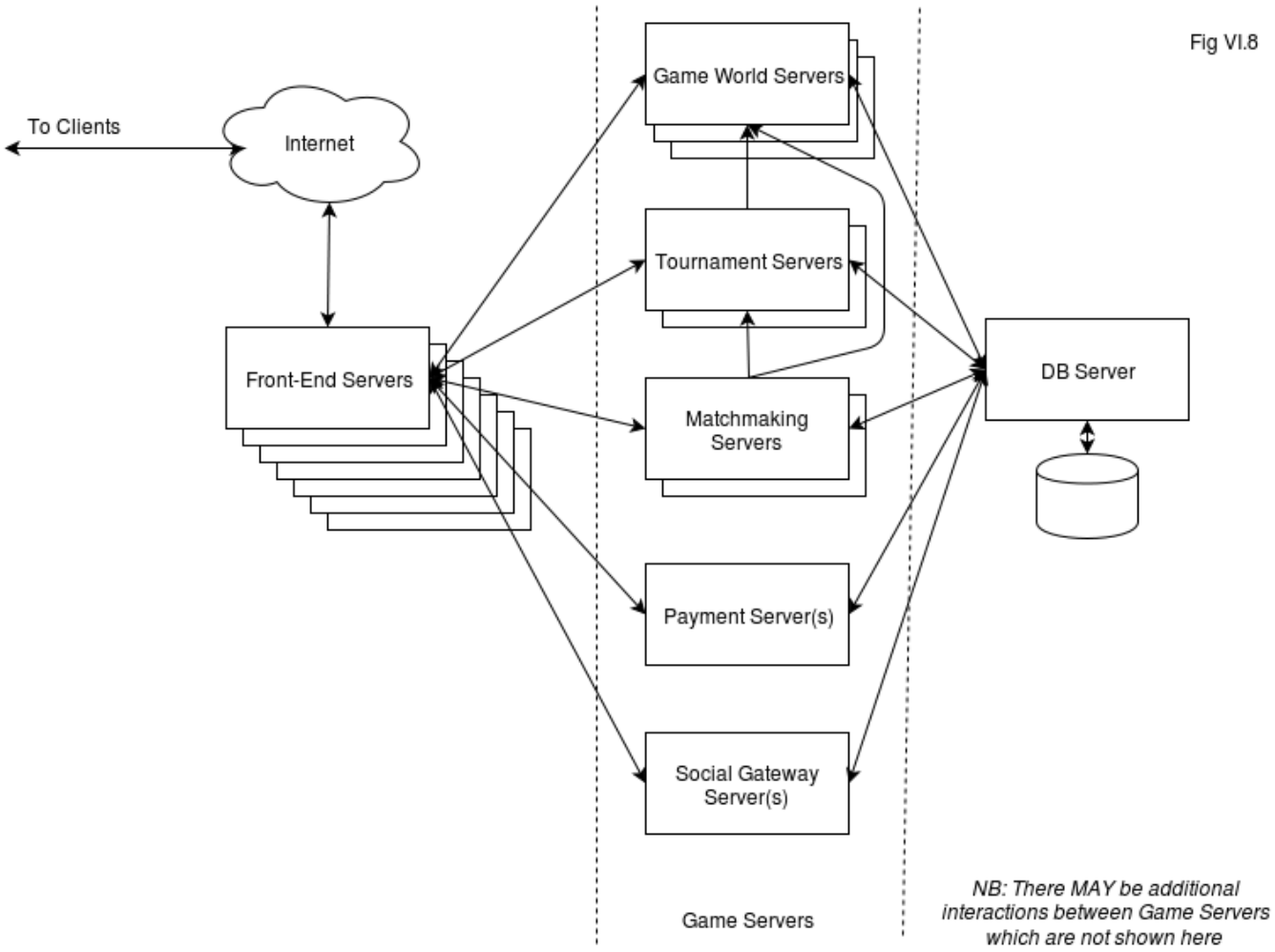


Enter Front-End Servers

[Enter Juliet] Hamlet: Thou art as sweet as the sum of the sum of Romeo and his horse and his black cat! Speak thy mind! [Exit Juliet]
— a sample program in Shakespeare Programming Language —



Our Classical Deployment Architecture (especially if you do use FSMs) is not bad, and it will work, but there is still quite a bit of room for improvement for most of the games out there. More specifically, we can add another row of servers in front of the Game Servers, as shown on Fig VI.8:



As you see, compared to the Classical Deployment Architecture (see Fig VI.4 above) we've just added a row of Front-End Servers in front of our Game Servers. These additional Front-End Servers are intended to deal with all the communication stuff when it comes from the clients. All those pesky "whether the player is connected or not" questions (including keep-alive handing where applicable, see Chapter [[TODO]] for details on keep-alives), all that client-to-server encryption (if applicable), with all those keys etc., all those rather more-or-less strange reliable-UDP protocols (again, if applicable), and of course, routing messages between the clients and different Game Servers – all the communication with clients is handled here.

In addition, usually these Front-End servers store a copy of relevant Game Worlds when it is necessary, and are acting as "concentrators" for the game world updates; i.e. even if a Game Server has 100'000 people watching some game (like final of some tournament or something), it will need to send updates only to a few Front-End servers, and Front-End servers will take care of data distribution to all the 100'000 people. This ability comes really handy when you have some kind of Big Final game, with thousands of people willing to watch it (and you don't really need to make it a video broadcast, which is not-so-convenient for existing players and damn expensive, but you can do it right within your client).

"In addition, usually these Front-End servers store a copy of

More on it below, and implementation of this "concentrator" paradigm is discussed in more detail in Chapter [[TODO]].

We'll discuss the implementation of our Front-End servers a bit later, but for now let's note that most importantly,

relevant Game Worlds when it is necessary, and are acting as "concentrators" for the game world updates

Front-End Servers MUST be easily replaceable without significant inconveniences to players

That is, if any of Front-End Servers fails for whatever reason – the most a player should see, is a disconnect for a few seconds. While still disruptive, it is very much better than scenarios such as "the whole game world went down and we need to restore it from backup". In other words, whenever Front-End server crashes for whatever reason, all the clients who were connected there, need to detect the crash (or even worse, "black hole") and automagically reconnect to some other Front-End server; in this case all the player can see, is a momentarily disconnect (which is also a nuisance, but is much better than to see your game hang).

Front-End Servers: Benefits

Whenever we're adding another layer of complexity, there is always a question "Do we really need it?" From what I've seen, having easily replaceable Front-End Servers in front of your Game Servers is very valuable and provides quite a few benefits. More specifically:

- Front-End Servers take some load off your Game Servers, while being easily replaceable
  - it means that you can have less Game Servers
    - this, combined with the observation that Front-End Servers are easily replaceable, means that you improve reliability of your site as a whole; instances when some of your Game World Servers go down, will occur more rarely (!)
  - having a copy of relevant game world(s) on your Front-End Servers, takes even more load off your Game Servers, and makes Game Server load independent on the number of observers
- you can use really cheap boxes for your Front-End Servers; strictly speaking, you don't even need ECC and RAID for them (and you certainly do need them for your Game Servers). As noted above, Front-End Servers are easily replaceable, so if one goes down – its load is automagically redistributed among the others (see Chapter [[TODO]] for further details). If you're going to

"you can use

deploy into the cloud – you may want to consider cheaper offers for your front-end servers (even if they're coming from different CSP).[1]

really cheap boxes for your Front-End Servers

- they allow your client to have a single connection point to the whole site; benefits of this approach include better control over player's "last mile" so that priorities between different data streams can be controlled, eliminating difficult-to-analyze "partial connections", and hiding more implementation details of your site from the hostile world outside; more on single client connection in Chapter [[TODO]]
- they allow for trivial client-side load balancing (no hardware load balancers needed, etc. etc.), more discussion on the load balancing below in "On Client-Side Load Balancing and Law of Big Numbers" section below
- having a copy of relevant game world(s) on your Front-End Servers allows to have virtually unlimited number of observers who want to watch some of the games being played on your site (such as a Big Final or something[2]) Best of all, this will happen *without affecting game server's performance (!).* Moreover, usually you won't need to organize anything for your Big Final, the system (if built properly) can take care of it itself, in (roughly) the following manner:
  - whenever somebody comes to watch a certain game, his client requests this game from the Front End Server
  - if Front End Server doesn't have a copy of the requested game, it requests it from the relevant Game Server, alongside with updates to the game world state
  - from this point on, Front End Server will keep an "in-sync" copy of the game world, providing it (with updates) to all the clients which have requested it
  - it means that from this point on, even if you have 100'000 observers watching some game on this Game Server, all the additional load is handled by your Front-End Servers, without affecting your Game Server
  - for further details, see Chapter [[TODO]].
- Front-End Servers allow for better security later on (acting essentially as a kind of DMZ, see Chapter [[TODO]] for details).

---

[1] keep in mind that you still need top-notch connectivity

[2] and as Big Finals are a good way to attract attention, this does provide you an edge over your competitors, etc. etc.

Front-End Servers: Latencies and Inter-Player Latency Differences

"You can have processing time of your Front End server application-layer of the order of single-digit microseconds.

As for the negative side of having Front End Servers, I can think only of two such drawbacks. The first one is additional latency introduced by your Front End Server. More specifically, we're speaking about the time which is necessary for the packet incoming from a client at application layer, to get processed by your Front End Server, to go into TCP stack on Front End Server side,[3] to get out of TCP stack on Game Server side, and to reach application layer in your Game Server (plus the time necessary to go in the opposite direction).

Let's take a look at this additional latency. From my experience, if you're using a reasonably good communication layer library, you can have processing time of your Front End server application-layer of the order of single-digit microseconds.[4] Then, we have an end-to-end TCP connection from your Front End Server to your Game Server; latencies of such a connection (over 10GB Ethernet) have been measured at around 8 µs [Larsen2007]. Adding these two delays together and multiplying it by two to get RTT, would mean that we're still staying well below 100 µs. However, there are some further considerations (such as switch delays, differences between different operating systems, differences between games, etc.) which make me uncomfortable to say that you will have no problem achieving 100 µs delay (i.e. either you may, or you may not). On the other hand, I am ready to say that if you're careful enough with your implementation, reducing the delay introduced by Front-End Servers, down to 1ms is achievable in all but most weird cases.

To summarize:

- if additional latency of around 1 millisecond is ok for you – don't worry about additional latencies and go for Front-End Servers; this certainly covers all genres with the only potential exception being MMOFPS
- if additional latency you can live with, is well below 1 millisecond (which is difficult for me to imagine as it is still over an order of magnitude less that 1/60 sec frame update time, but in MMOFPS world pretty much anything can happen) – think about it a bit more and try to find out (ideally – experimentally) what kind of latency you can achieve in practice; if your experiments show that latencies are indeed unacceptable, you MIGHT need to drop those Front-End Servers because of the latency they're introducing[5]
- YMMV, no warranties of any kind, batteries not included

The second (IMHO more theoretical, but as usual, YMMV) potential issue with having Front-End Servers would arise if some of your Front-End Servers are overloaded (or they're running using significantly different hardware), so those

players connected to less-loaded Front-End Servers, will have lower latencies, and therefore will have an advantage.

On the one hand, I didn't see situations where it makes any practical difference in real-world deployments (i.e. as I've seen it, if some of the Front-End Servers are overloaded, it means that most of the other ones are already at 90%+ of capacity, which you should avoid anyway; see [[TODO!]] section for further discussion of load balancing). On the other hand, YMMV and in theory you might get hit by such an effect (though I certainly don't see it coming into play for anything but MMOFPS).

"If/when such inter-player latency becomes a real problem, you MAY need to implement some kind of affinity for players of certain Game Worlds to certain Front-End servers

If such inter-player latency differences become the case (and only when/if it becomes a real problem), you MAY need to implement some kind of affinity for players of certain Game Worlds to certain Front-End servers (more on affinity in "On Affinity" section below). However, keep in mind that large-scale affinity tends to remove most of the benefits provided by Front-End Servers, so if you feel that you're going to implement affinity for each-and-every-game – you'll probably be better without Front-End Servers (implementing affinity only for a small percentage of your games, such as "high profile tournaments" will cause less trouble, see "On Affinity" section below for further discussion).

---

[3] yes, I'm arguing for TCP connections for inter-server communications in most cases, see "On Inter-Server Communication" section above. On the other hand, UDP is also possible if you really really prefer it.
[4] note that this might become a non-trivial exercise, see further discussion in Chapter [[TODO]]. On the other hand, I've done it myself.
[5] in theory, you may also want to experiment with something like Infiniband, which BTW would fit nicely in overall QnFSM architecture with communications neatly isolated from the rest of the code, but most likely it won't be worth the trouble

Client-Side Random Balancing and Law of Big Numbers

As soon as you have several Front-End servers where your clients are coming, you have a question "how to ensure that all the Front-End Servers are loaded equally", i.e. a typical load balancing question. Load balancing in general is quite a big topic at least over last 20 years. Three most common techniques out there are the following: DNS Round-Robin, Client-Side Random Balancing, and Server-Side (usually hardware-based) Load Balancers. With the industry producing those hardware boxes behind the last one, there is no wonder that it becomes more and

more popular at least in the enterprise world. Still, let's take a closer look at these load balancing solutions.

DNS Round-Robin

DNS round-robin is based on a traditional DNS requests. Whenever a client requests address frontend.yoursite.com to be resolved into IP address, a DNS request is sent (this stands with or without DNS round-robin) to your (or "your DNS provider's") DNS server. If DNS server is configured for DNS round-robin, it returns *different* IP addresses to different DNS requests, in a round-robin fashion[6] hence the name.

> "one of these returned IPs can get cached by a Big Fat DNS server, and then get distributed to many thousands of clients

DNS Round-Robin, when applied to balancing browsers across different web servers, has two major disadvantages. First of all, there is a problem with caching DNS servers along the path of the request (which is a very standard part of DNS handling). That is, even if your server is faithfully returning all your IPs in a round robin fashion, one of these returned IPs can get cached by a Big Fat DNS server (think Comcast or AT&T), and then get distributed to many thousands of clients; in this case distribution of your clients across your servers will be skewed towards that "lucky" IP which got cached by the Big Fat DNS server ☹ . The second problem with using DNS round-robin for web servers, if that if one of your servers is down, usual web browser won't try another server on the list, so usually in web server realm round-robin DNS doesn't provide server fault tolerance.

Fortunately, as we DO have a client, we can solve both these problems very easily. Moreover, these techniques will also work for your browser-based games (that is, *after* you've got your JS loaded and it started execution).

---

[6] strictly speaking, it is a little bit more complicated than that, as DNS packets contain a list of servers, but as virtually everybody out there ignores all the entries in returned packet except for the very first one, it is more or less equivalent to returning only one IP per request – that is, unless you have your own client which can do the choice itself, see "Client-Side Balancing"

Client-Side Random Balancing

To improve on DNS round-robin, a very simple idea can be used. We won't rotate anything on the server side; instead, we will distribute *exactly the same* list of servers to all the clients. This list may be hardcoded into your clients (and that's what I've used personally with big success), or the list can be distributed via DNS as a simple list of IPs for desired name (and retrieved on client via getaddrinfo() or equivalent). Which way to prefer – doesn't matter to us now, but we'll discuss relevant issues in Chapter [[TODO]].

"Client simply takes random item from the IP list, and tries connecting to this randomly chosen IP.

As soon as the client gets the list of IPs, everything is very simple. Client simply takes random item from the IP list, and tries connecting to this randomly chosen IP. If connection attempt is unsuccessful (or connection is lost, etc.) – client gets another random item from the list and tries connecting again.

One note of caution – while you don't really need a cryptographic-quality random generator to choose the IP from the list, you DO want to avoid situations when your random number generator (the one used for this purpose) is essentially just some function of coarse-grained time. One Really Bad example would be something like

```
1  int myrand() {//DON'T DO THIS!
2    srand(time(0));
3    return rand();
4  }
```

In such a case, if you get mass disconnect (and as a result all your players will attempt to reconnect at about the same time), your IP distribution will likely get skewed due to too few differences between the clients trying to get their IP addresses; if all the clients attempt to connect within 5 seconds, with such a bad *myrand()* function you'll get at most 5 different IPs (less if you're unlucky). Other than such extremely bad cases, pretty much any RNG should be fine for this purpose. Even a trivial linear congruential generator, seeded with time(0) *at the moment when the program was launched* (and NOT at the moment of request, as in example above), should do in practice, though adding some kind of milliseconds or some other randomly looking or client-specific data to the mix is advisable "just in case".

Law of Large Numbers

According to the law, the

Client-Side Random Balancing: a Law of Large Numbers, and comparison with DNS Round-Robin

Unlike DNS round-robin (which in theory provides "ideal" balancing), client-side random balancing relies on the statistical Law of Large Numbers to achieve flat distribution of clients between the servers. What the law basically says is

> **average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed.**
> — *Wikipedia* —

that for independent measurements, the more experiments you're performing – the more flat distribution you'll get. [[TODO!: add stuff about binomial distribution, and an example]]

In practice, despite being "non-ideal" in theory, client-side random balancing achieves much more flat distribution than DNS round-robin. The reason for it is two-fold. First, as soon as the number of clients is large (hundreds and up), client-side random balancing becomes sufficiently flat for practical purposes (and if your system is provisioned for thousands of players, and only a few have came yet – the distribution won't be too flat, but the inequality involved won't be able to hurt, and the balance will improve as the number grows). On the positive side, however, client-side random balancing doesn't suffer from DNS caching issue described above. Even if you're using DNS to distribute IP lists (and this list gets cached) – with client-side balancing *all the IP lists circulating in the system are identical by design,* so caching (unlike with DNS round-robin) doesn't change client distribution at all.

To summarize: personally, I would be very cautious to use DNS Round-Robin for production load balancing. On the other hand, I've seen Client-Side Random Balancing to work extremely well for a game which grew from a few hundreds of simultaneous players into hundreds of thousands; it worked without any problems whatsoever, providing almost-perfect balancing all the time. That is, if the average load across the board was 50%, you could find some servers at 48% and some at 52%, but not more than that.[7]

As for the second disadvantage mentioned above for DNS Round-Robin as applied to web browsers (which was inability of most of the browsers to provide fault tolerance in case when one of the servers crashes) – this evaporates as soon as we have the whole list on the client-side, can detect failure, and can select another item from the list.

---

[7] this, of course, stands only when you have run your servers identically for sufficient time; if one of the servers has just entered service, it will take some hours until it reaches the same load level than the others. If really necessary, this effect can be mitigated, though mitigation is rather ugly and I've never seen it necessary in practice

Server-Side Load Balancers

An approach which is very different from both round-robin DNS and client-side

random balancing, is to use server-side load balancers. Load balancer is usually an additional box, sitting in front of your servers, and doing, as advertised, load balancing.

"These additional balancing capabilities are usually completely unnecessary for games (where Law of Large Numbers tends to stand very firmly)

Server-side load balancers do have significantly more balancing capabilities with regards to scenarios when different clients cause very different loads (so that server-side balancers can work even if the Law of Large Numbers doesn't work anymore). However, on the one hand, these additional balancing capabilities are usually completely unnecessary for games (where Law of Large Numbers tends to stand very firmly), and on the other hand, such load balancer boxes tend to be damn expensive (double that if you want redundancy, and you certainly want it), they do not allow inter-datacenter balancing and fault tolerance (by design), and they introduce additional not-so-well-controlled latencies.[8]

Oh, and BTW – when speaking about redundancy and the cost of their boxes, quite a few hardware manufacturers will tell you "hey, you can use our balancer in active/active configuration, so you won't waste anything!". Well, while you *can* indeed use many server-side load balancers in active/active configuration, you still MUST have at least one redundant box to handle the load if one of those boxes fails. In other words, if all you have is two boxes in active/active configuration, when both are working, overall load on each of them MUST be well below 50%, there is no way around it if you want redundancy.

As a result of all the considerations above, for game load-balancing purposes I have never seen any practical uses for server-side load balancer boxes (as always, YMMV and batteries are not included). Even if you're using Web-Based Deployment Architecture (in the way described above), you should be able to stay away from them (though YMMV even more).

---

[8] most of load balancers are designed to balance web sites where anything below 100ms is pretty much nothing, so at the very least make sure to discuss and measure the latency (under your kind of load!) before buying such a box

Balancing Summary

From my experience, client-side random balancing (aimed towards front-end servers) worked really good, and I've never seen any reasons to use something different. Round-robin DNS is almost universally inferior to client-side balancing, and hardware-based server-side balancers are too complicated and expensive,

usually without any real reason to use them in gaming environment. As note above, one exception when you MAY need server-side balancers, is if you're using Web-Based Deployment Architecture.

One last word about load balancing: it *is* possible to use more than one of the methods listed here (and it might even work for you); however, implications of such combined use of more than one method of load balancing, are way too convoluted to discuss them in this book.

Front-End Servers as a CDN

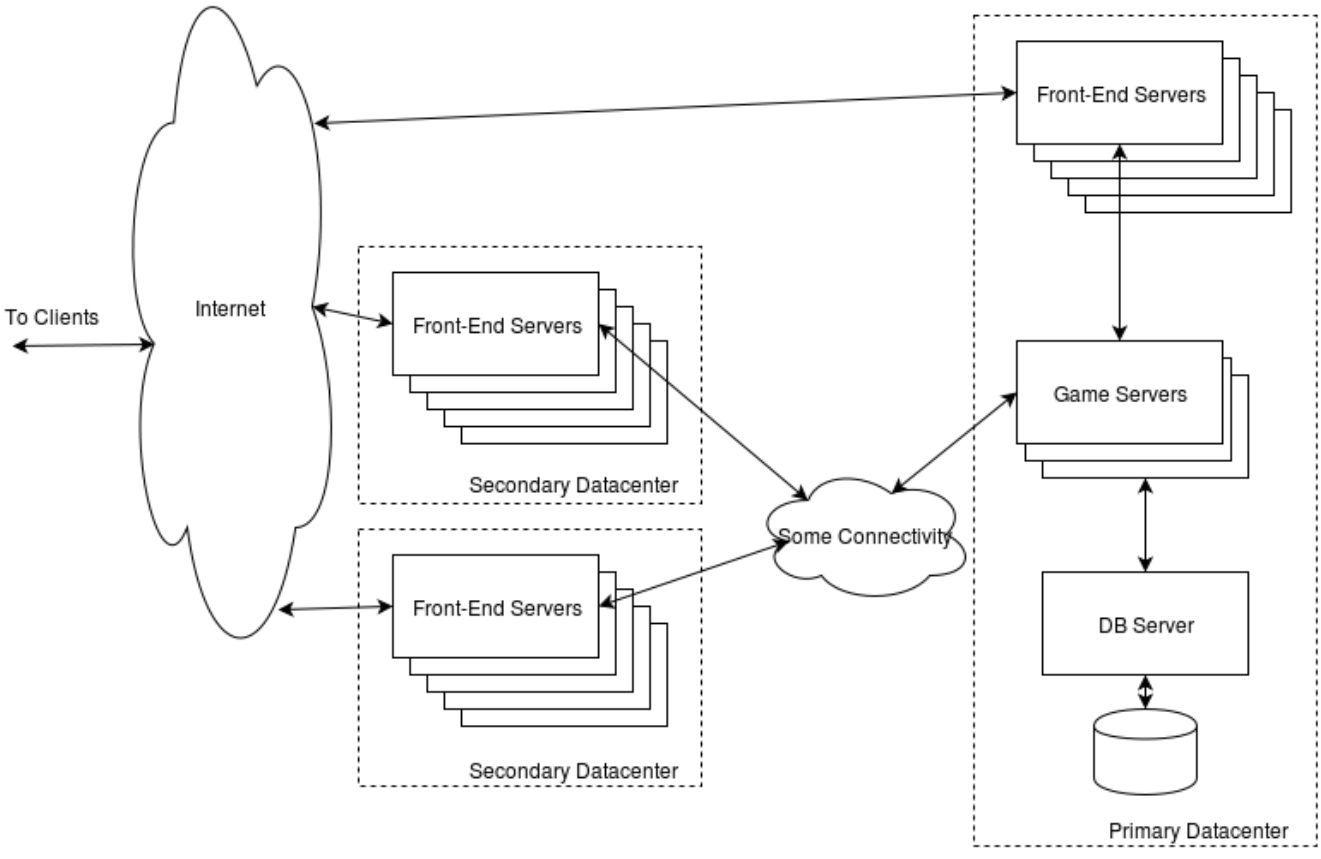
It is possible to use Front-End Servers as a kind of CDN (or even use them to build your own CDN). Even if you're running all your Game Servers from one single datacenter, for certain kinds of games it might be a good idea to have your Front-End Servers sitting in different datacenters (and acting as different "entry points" to your clients), as shown on Fig VI.9:

CDN

A content delivery network or content distribution network (CDN) is a globally distributed network of proxy servers deployed in multiple data centers

— Wikipedia —

Fig VI.9



RARELY WORTH THE TROUBLE

The idea here is pretty much like the one behind classical CDN: to reduce latencies

for end-users. On the other hand, we need to note that

unlike classical CDN, the content with our game-sorta-CDN is not static, so gain in latencies is possible only because of better peering, with gains usually being in single-digit milliseconds

There is still a different reason to use such deployment architectures – in case if you want to protect yourself from Internet connectivity in your primary datacenter going down (provided that "Some Connectivity" survives); in practice, if you have a decent datacenter, it should never happen. More precisely – your datacenter WILL occasionally experience transient faults of around 1.5-2 minutes long (typical BGP convergence time), so if you're looking for excuses to use this nice diagram on Fig VI.9 *and* your client can detect the fault and redirect to a different datacenter significantly faster than that, it MAY make some difference to your players.

Implementation-wise, there are several considerations for such CDN-like multi-datacenter Front-End Server configurations:

- you MUST have very good connectivity between your data centers ("some connectivity" on Fig. VI.9). At the very least, you should have inter-ISP peering explicitly set by both of your ISPs (to each other) to ensure the best data flow for this critical path
  - strictly speaking, "some connectivity" does not necessarily need to be Internet-based; you often can save additional few milliseconds by getting something like "dedicated" Frame-Relay between your datacenters, but this will likely cost you in the range of tens of thousands per month 😕 .
- traffic on "some connectivity" can be an order (or even two) of magnitude lower than that going to the clients due to Front-End Servers acting as "concentrators"
- you SHOULD account for secondary datacenter to go down (in particular, in case of inter-datacenter connectivity going down). The simplest way to deal with it is to have enough capacity in your primary datacenter (both traffic-wise and CPU-wise) to handle *all* of your clients, but this tends to be expensive. As an alternative, shutting down some activities in case of such a failure may be possible depending on specifics of your game.

"CDN-like arrangements of Front-End Servers MAY save some of your players a few milliseconds in

Bottom line for CDN-like arrangements. CDN-like arrangements of Front-End Servers may save some of your players a few milliseconds in latency (that is, if you have a really

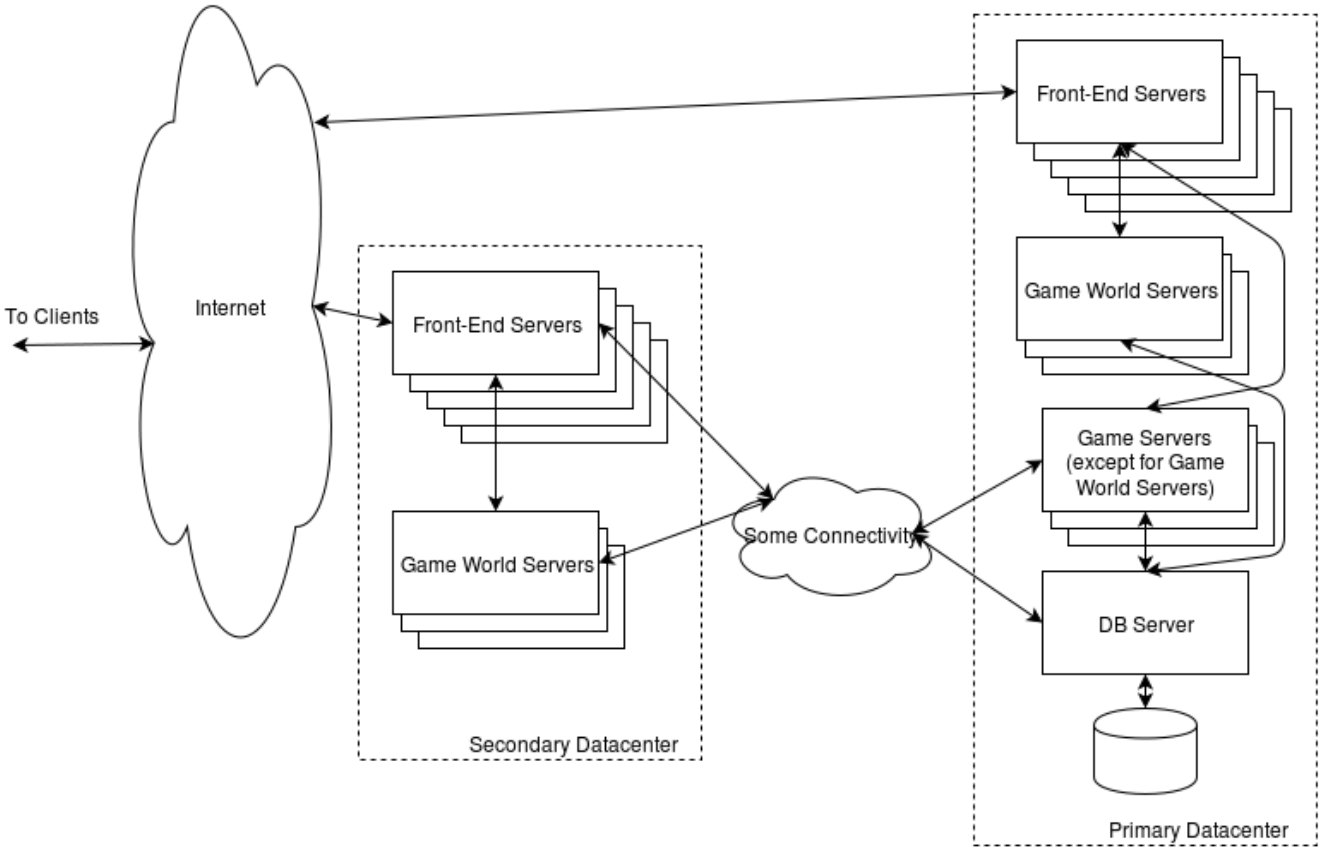
good connection between datacenters), which in turn may allow to level the field a bit with regards to latency. From my experience, it was hardly worth the trouble (because you cannot really improve MUCH in terms of latency, as the packets still need to go all the way to the Game Server and back), but keep the possibility in mind. For example, it may come handy in some really strange scenarios when you're legally required to keep your game servers in a strange location (hey casino guys!) where you simply don't have enough bandwidth to serve your clients directly.

latency. From my experience, it was hardly worth the trouble

Front-End Servers + Game Servers as a kinda-CDN

On the other hand, if you're really concerned about latencies, it is usually much better to bring your Game World Servers closer to players (while leaving DB Server behind), as shown on Fig VI.10:

Fig VI.10



Here, we're moving the most time-critical stuff (which is usually your Game World Servers) towards the end-user, providing significantly better latencies to those players who're in the vicinity of corresponding datacenter. Maintaining such infrastructure is quite a Big Headache, but is doable, so if you're really concerned about latencies – you may want to deploy in such a manner. A word of caution – if going this way, you will end up with "regional servers", which have their own share of troubles (you'll need to ensure that clients in the region go only to the relevant Front-End Servers, security on inter-datacenter connections becomes quite an

issue, etc., etc.); once again – it is doable, but go this way only if you *really* need it.

On Affinity

In some cases, you may decide that you need to have a kind of "affinity" so that some specific players (usually those playing in a specific game world) are coming to specific Front-End Servers.

"The things will go smoothly as long as the number of the game worlds which use affinity is small.

Note when we're speaking about our Front-End Servers, "affinity" is quite different from classical affinity (usually referred to as "persistence" or "stickiness") used on load balancers for web servers. In the web world persistence/stickiness is about having the same client coming to the same server (to deal with sessions and per-client caches). For our Front-End Servers, however, affinity has a very different motivation, and is usually about Front-End-Server-to-game-world affinity (for players or for players+observers) rather than client-to-server affinity (see "Front-End Servers: Latencies and Inter-Player Latency Differences" section above for one reason where you MIGHT need such affinity).

Technically, implementing Front-End-Server-to-game-world-affinity is not *that* difficult, but the real problems will start *after* you deploy your affinity. In short – the things will go smoothly as long as the number of the game worlds which use affinity is small. On the other hand, as soon as you have a significant chunk of your players connected using the affinity rules, you will find that achieving reasonable load balance between different Front-End Servers becomes difficult 🙁 . When there is no affinity, the balance is near-perfect just because of the Law of Large Numbers; as you're introducing the affinity rules, you're starting to skew this near-perfectly-flat distribution, and the more players are affected by affinity, the more you're deviating from the ideal distribution, so managing those rules while achieving load balance can become a Big Fat Challenge.

Bottom line: avoid affinity as long as possible (and most likely you will be able to get away without it).

Front-End Servers: Implementation

Now let's discuss ways how our Front-End Servers can be implemented. As mentioned above, the key property of our Front-End Servers is that they're easily replaceable in case of failure. To achieve this behavior,

you MUST ensure that there is NO original game-

world state on any of your Front-End Servers
In other words, Front-End Servers should have only a replica of the original game-world state, with the original game-world state kept by Game Servers

There is no need to worry too much about it if you're using a generic subscriber/publisher (or state replication) kind of stuff, but be extremely careful if you're introducing any custom logic to your Front-End Servers, because you may lose the all-important "easily replaceable" property above. See Chapter [[TODO]] for further discussion of this potential issue.

Front-End Servers: QnFSM Implementation

One implementation of the Front-End Server implemented under pure Queues-and-FSMs architecture (see Chapter V for details on QnFSM, state machines, and queues) is shown on Fig VI.10:

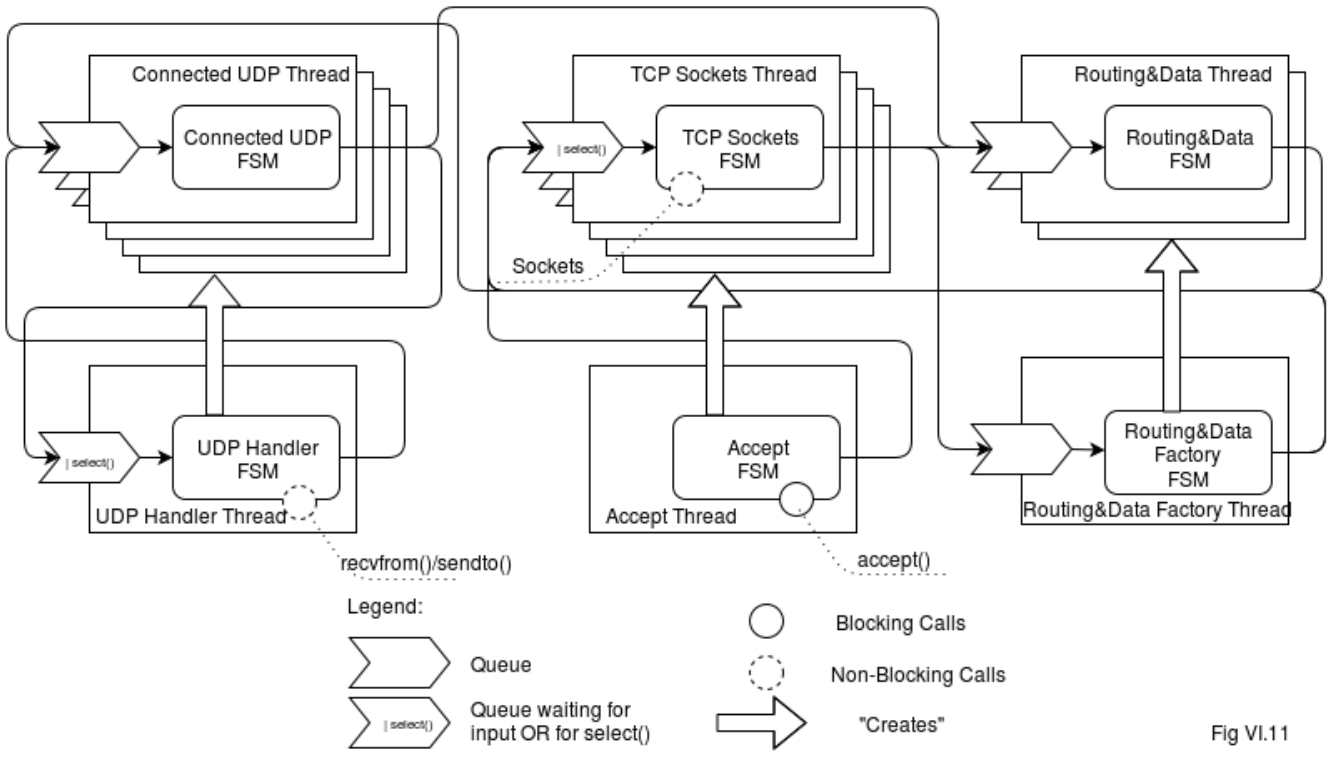


Fig VI.11

Here, we have TCP- and UDP-related threads similar to those described in "Implementing Game Servers under QnFSM architecture" section above with regards to Game Servers, and one or more of Routing&Data Threads (with at least one Routing&Data FSM each), which are responsible for routing of all the packets, and for caching the data (such as "game world" data). Let's discuss these routing-related FSMs in a bit more detail.

Routing&Data FSMs. Each of Routing&Data FSMs has its own data that it handles (and updates if applicable). For example, one such Routing&Data FSM may contain

a state of one game world. Other Routing&Data FSMs may handle routing of the point-to-point packets from players to (and from) one specific Game Server. Further details of the data types handled by Routing&Data FSMs will be discussed in Chapter [[TODO]], but generally there will be three different types of Routing&Data FSMs:

- generic connection handlers (to handle point-to-point communications including player input and server-to-server connections)
- generic publisher/subscriber handlers (to cache and handle generic but structured data such as a list of available games, if players are allowed to select the game)
- specific game world handlers (to cache and handle game world data if the required functionality doesn't fit into generic handler). In many cases you'll be able to live without specific game world handlers, but if you want to implement some kind of server-side filtering, like server-side fog-of-war to avoid sending data to those players who shouldn't see it (so no hack of the client can possibly lift fog-of-war) – specific game world handlers become a necessity.

"It is possible (and often advisable) to have more than one Routing&Data FSM within single Routing&Data Thread

It is possible (and often advisable) to have more than one Routing&Data FSM within single Routing&Data Thread to reduce unnecessary load due to an exceedingly high number of threads (and unnecessary thread context switches). How to combine those Routing&Data FSMs into specific threads – depends on your game significantly, but usually generic connection handlers are extremely fast and all of them can be combined in one thread. As for generic publisher/subscriber and specific game world handlers, their distribution into different threads should take into account typical load and allowed latencies. The rule of thumb is (as usual) the following: the more FSMs per thread – the more latency and the less thread-related overhead; unfortunately, the rest depends too much on specifics of your game to discuss it here.

Routing&Data Factory Thread. Routing&Data Factory Thread is responsible for creating Routing&Data Threads (and Routing&Dara FSMs), according to requests coming from TCP/UDP threads. A typical life cycle of Routing&Data FSM may look as follows:

- One of TCP/UDP FSMs needs to route some message (or to provide synchronization to some state), and realizes that it has no data on Routing&Data FSM, which it needs to route the message to, in its own cache.
- TCP/UDP FSM sends a request to Routing&Data Factory FSM

- Factory FSM creates Routing&Data Thread (with an appropriate Routing&Data FSM)
- Factory FSM reports ID of the Queue, where the messages towards appropriate Routing&Data FSM should be sent, back to the requesting TCP/UDP Thread
    - TCP/UDP FSM (the one mentioned above) sends the message to the appropriate Queue (using ID rather than pointer to enable deterministic "recording"/"replay", see Chapter V for details).
- Whenever the Routing&Data FSM is no longer necessary for its purposes, TCP/UDP FSM reports it to the Factory FSM
    - if it was the last TCP/UDP FSM which needs this Routing&Data FSM, Factory FSM may instruct appropriate Routing&Data Thread to destroy the Routing&Data FSM

Routing&Data FSMs in Game Servers and Clients

I need to confess that personally I am *positively in love* these Routing&Data FSMs. I *love* them so much that I usually have not only on Front-End Servers, but also on Game Servers, and on Clients too; while they're not strictly necessary there (and are not shown on appropriate diagrams to avoid unnecessary clutter), they did help me to simplify things quite a bit, making all the communications very uniform. Still, it is pretty much your choice if you want to have Routing&Data stuff on your Game Servers and/or Clients.

Front-End Servers Summary

To summarize the section on Front-End Servers:

"As a rule of thumb, Front-End Servers are a Good Thing™.

- As a rule of thumb, Front-End Servers are a Good Thing™. In particular:
    - they take the load off your Game Servers
        - which often makes the system cheaper (as Front-End Servers are cheap)
        - and also improves overall system reliability (as Front-End Servers are easily replaceable)
    - they facilitate single client connection (which is generally a good thing to have, see Chapter [[TODO]] for further discussion)
    - they facilitate client-side load balancing
    - they allow to handle 100'000+ observers for your Big Event easily (actually, the sky is the limit)
    - their drawbacks are pretty much limited to the additional latency, and this additional latency is firmly in sub-millisecond range

- Client-side load balancing usually is the best one for games
  - one potential exception is Web-Based Deployment Architectures, where you MAY need server-side balancers
  - large-scale affinity is to be avoided
- CDN-like arrangements are possible, but not without caveats
- Front-End Servers can (and IMHO SHOULD) be implemented in QnFSM architecture, as described above

[[To Be Continued…

This concludes beta Chapter VI(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 VI(c), "Modular Architecture: Server-Side. Eternal Windows-vs-Linux Debate.]]

[–] References

[Larsen2007] Steen Larsen, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter VI(a). Server-Side MMO Architecture. Naïve, Web-Ba…*

MMOG Server-Side. Eternal Linux-vs-Windows Debate »

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

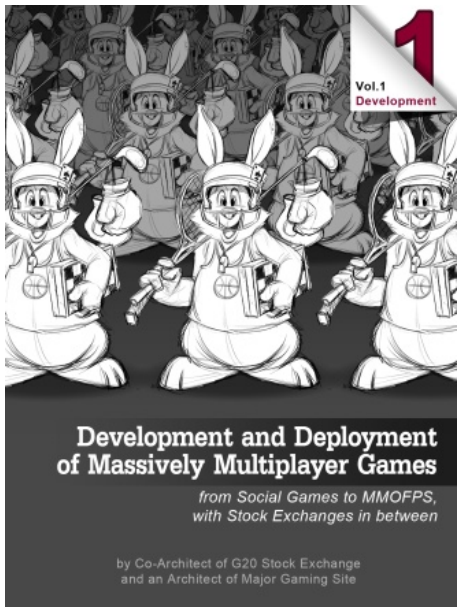
Copyright © 2014-2016 ITHare.com

MMOG Server-Side. Eternal Linux-vs-Windows Debate

posted January 4, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter VI(c) 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.]]



Operating Systems

Please don't expect to find anything new in this section, especially in the context of "which OS is the best one out there". It is merely a summary of well-known things as they apply to MMOG server-side.

For the client-side, operating system is normally a big fat Business Requirement, which means that we as developers don't have much choice about it. If we need to support Android, iOS and Windows on the client-side – we just need to shut up and do it, plain and simple. With operating system for the server-side, situation is usually different – as nobody on the business side of things really cares (or at least SHOULD NOT care) about which OS is used to run our servers, it is more or less a developer's choice. What MAY (and actually SHOULD) interest business guys/gals though, is time-to-market and the cost of running servers, more on it below.

When it comes to server-side operating systems, there are actually only two realistic choices: Windows and Linux[1] (or "Linux and Windows", depending on your preferences, we'll discuss this in a moment). While in theory you can run an OS X server, or can dream about trying a 32-core SPARC M7 under Solaris, or (like myself) be eager to get your hands on the latest greatest POWER8 box, in practice all we'll ever get (except, maybe, for stock exchange guys) is x64 box with either Windows or Linux. And while there is nothing wrong about x64, it still often leaves us feel a bit sad about all those existing-but-never-available opportunities.

Leaving sentimental feelings aside, we need to take a look at two real contenders: Linux/BSD and Windows. Unfortunately, over the course of serveral last ~~centuries~~ decades, any attempt to to take such a look has invariably lead to almost-religious wars.

---

[1] For the purposes of our discussion, we'll consider BSD as a flavour of Linux (which is admittedly a sacrilege, but Linux programming and *BSD programming at our application level are that similar, that with a few narrow exceptions such as epoll/kqueue, we can pretty much ignore the difference until actual deployment).

New Generation Chooses Cross-Platform! Well, at least it SHOULD...

One thing you should seriously consider before choosing one single OS as your development target, is "whether you can make your program cross-platform

instead". In general, I strongly support cross-platfrom programs, even on the server-side, for several reasons:

- we don't need to go into Linux-vs-Windows debate right here, making it a deployment-time issue rather than development-time issue. Not only having cross-platform code postpones the debate, but also it makes the debate much less heated, as the cost of mistake at deployment-time is orders of magnitude lower
- cross-platform programs are, well, cross-platform, which gives you deployment-time freedom
  - for example, if you find that for the purposes of your game the latest greatest TCP stack from Linux (or Windows) works significantly better (see "Other Technical Differences (kernel scheduler, TCP stack, etc)" section below) – you can switch without that much hassle
  - moreover, you can have some servers on Windows and some on Linux at the same time (optimizing different audiences according to different parameters)
- cross-platform programming helps to keep dependencies in check
- cross-platform programs tend to have better structured codebases (I attribute it to better discipline, so it is not inherent to cross-platform programs, but a correlation)
- cross-platform programming help to test your code better. It has been observed that running a program which was considered perfectly error-free, on a different platform, helps to reveal quite a few subtle bugs which have never manifested themselves on original platform (but were sitting there, just waiting for the right moment to kick in).

How to achieve a holy grail of cross-platform code, is a separate story, which we'll discuss in [[TODO]] section below. For now, let's just make a note that going cross-platform does not necessarily mean going JVM (Python, Erlang, pick your poison), and that C++ can also be made perfectly cross-platform, so at least don't write it off on these grounds. On the other hand, let's keep in mind that outside of deterministic FSMs (and for pretty much any programming language), the best we can possibly hope for, is "run once – test everywhere", and "testing everywhere" takes time ☹. Which, in turn, makes convincing managers going cross-platform route quite difficult (that is, unless you're using Java/Python/...), so you *may* need to choose your OS even if you would like to avoid it in the first place.

Eternal Windows-vs-Linux Debate

I realize that for the analysis below, I will be hit hard by zealots from both sides. On the other hand, as choosing server-side OS is an important part of the overall MMO exercise, I need to provide at least some observations in this regard, so I have no choice other than to brace myself and be prepared to all the punches from both Windows and Linux fans (with an occasional hit by BSD/Solaris proponents).

"I have no choice other than to brace myself and be prepared to all the punches from both Windows and Linux fans

Now, we can forget about the boring cross-platform stuff, and to concentrate on the classical Linux-vs-Windows flame war. BTW, most of the arguments routinely raised in such flame wars, do have some merit behind them, with the tricky part being to estimate applicability and impact of these arguments within the specific context. Let's take a closer look at some of them (only in the context of the server-side specifically for games):

Open-Source

The practical argument here goes along the lines of "if you ever have a problem, you'll be able to fix it". However, being a game developer, I don't think it is realistic to expect that you'll be able to fix anything in Linux kernel (or, Linus forbid, driver). If you've done it before – of course, being able to fix things in kernel becomes an all-important argument, but otherwise – don't hold your breath over it.

Stability/Reliability

There are a lot of horror stories about Windows being unstable/unreliable, including (in)famous migration of London Stock Exchange from Windows to Linux in 2009. [http://www.itwire.com/opinion-and-analysis/the-linux-distillery/28359-london-stock-exchange-gets-the-facts-and-dumps-windows-for-linux] My personal experience, however, doesn't support this observation. In short – from what I've seen, if all you're using from Windows, is Windows kernel (without any fancy COM components or .NET) – Windows has been observed work perfectly fine (more on disabling unnecessary software in Chapter [[TODO]]). Add anything large on top of a bare Windows kernel – and if you're not careful enough, you're entering much riskier waters, to put it mildly. Pretty much the same goes for Linux, but as Linux doesn't try to cover everything-under-the-sun as a part of operating system, you can usually choose which software to use, more freely. Still, from my experience, if you're careful enough, it is more or less a tie between Linux and post-9x Windows in the stability realm.

Security

> "Personally, I would agree that Linux is somewhat more secure (that is, if you're exercising at least basic caution and are not running your web server under root account).

Another quite popular argument is that Linux is more secure than Windows (what Microsoft vehemently objects, mostly on the basis of the number of reported bugs, which is a very convenient metrics for a closed-source company). Personally, I would agree that Linux is somewhat more secure (that is, if you're exercising at least basic caution and are not running your web server under root account).

I tend to attribute it to the fact that Linux in general is more modular than Windows, so disabling unnecessary parts is easier (and it is these unnecessary parts that cause most trouble). While this is partially offset by an atrocious *nix permission system (with suid bit abuses being responsible of a substantial chunk of successful real-world attacks), being highly modular still helps even in this department. Also SE Linux, despite all the shortcomings, does provide an additional layer of protection.

On the other hand, it is clear that you do need a highly qualified and security-aware admin to run any operating system securely. Just one very recent real-world breach example involved default Amazon EC2 Linux image to run Apache under root (and while SE Linux was running, SE policies didn't prevent attacker from taking the server over).[2] In short: it wasn't a problem of Linux as such, but a problem of Linux being misconfigured. However, it leads us to an all-important bottom line:

Each server is only as secure as its admin

If you have highly qualified admins, then I'd probably prefer Linux from security perspective, but in practice security advantage over Windows will likely to be negligible (that is, if you're using only "bare" Windows kernel, while disabling everything else, see above).

---

[2] if you don't understand why running your services under root account is a problem – wait until Chapter [[TODO]], we'll briefly discuss it there

Fast Network Packet Processing

If your game is a very latency-sensitive, all chances are that you'll need to use UDP (see Chapter [[TODO]] for further discussion). And when you're using UDP, you may easily run into your recvmsg() thread (or even recvmmsg() thread) becoming a bottleneck. One of the ways to deal with it in a cross-platform way, is to try

multiple threads calling recvmsg() on the very same (non-blocking) socket, which has been reported to work pretty good (which has been briefly described in "UDP-related FSMs" section above). However, if this doesn't help, you're pretty much out of cross-platform options. It means using rather obscure and little-known platform-specific APIs, which may include the following.

[[TODO!: Linux netmap/DPDK, Windows RIO]]

[[TODO!: Interrupt balancing: Linux RSS/RPS/RFS]]

Other Technical Differences (kernel scheduler, TCP stack, etc)

There are quite a few debates out there related to comparisons between Linux and Windows kernel schedulers and network stacks. In short – at least for games, the differences between them are negligible. A tiny bit more detailed analysis follows.

Regarding kernel/thread schedulers – note that for the game you certainly want to keep your CPU utilization low (even for social games having CPU utilization at 100% is certainly not a good idea), and thread queue – as short as possible. It means that there should always be a free CPU in the system, which is ready to process incoming packet.[3] It means that the scheduler (almost) always doesn't really have a choice which thread to schedule – *all* threads which are not waiting, will run, as there are (almost) always sufficient CPUs to run them. In practice, I don't know of any significant differences between Windows and Linux schedulers when applied to games; moreover, the difference was non-observable in practice even in the days of Linux O(n) scheduler.[4]

NUMA

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor.

— Wikipedia —

One closely related topic is related to so-called NUMA scheduling. The thing here is the following. In production, you're very likely to use 2-socket x64 servers, which are NUMA for the last 10 years or so. And for NUMA,[5] it is very important performance-wise to keep your threads' physical memory on the same socket (NUMA node) where your thread is running (otherwise memory accesses will need to go across the QPI/Hypertransport, which is slow compared to local memory accesses). The topic of keeping NUMA locality when scheduling, is still very much in active development (see, for example, [Corbet2013]), and does have a potential to bring significant benefits for applications (due to removal of unnecessary round-trips via QPI/Hypertransport). However, the last time I've seen (at least somewhat appropriate) comparison, I wasn't able to notice the difference between Windows and Linux in this regard (which might or might not be because of FSM-oriented architecture, which tends to exhibit very good memory locality and may be easier to handle

by NUMA schedulers). In short – jury is still out on Windows-vs-Linux NUMA scheduling, it may or may not affect your game (though IMHO the differences are not going to be drastic, at least not for long). Good description of NUMA on Linux can be found in [Lameter2013]. A bit more on practical suggestions related to manipulating NUMA-related things from application level will be mentioned in Chapter [[TODO]].

As for the TCP stack: all the TCP stacks out there start with the same RFC793 (yes, that's 1981 and still not obsolete); of course, there are several dozens RFCs on top of the basics described there, and sets of these RFCs and their defaults vary, but deep inside it is still pretty much the same thing (and even first several layers on top of it, such as Nagle's algorithm or SACK, are pretty much the same). Most of the differences between TCP stacks discussed out there, are actually about using different defaults/settings for TCP stack, which result in different throughput under different conditions (especially TCP performance over long fat pipes can be significantly different); however, these things, while interesting and important for video- and file-services, are not directly applicable to games, where average packet size is around 40-80 bytes (that's including 20 bytes of IP header). When it comes to latencies, network stack doesn't affect UDP latencies much, and TCP latencies will depend on lots of things, including TCP stack on the client side (not to mention that if you're into single-digit millisecond latencies, using TCP is probably not the best idea). One thing which *may* affect those games working over TCP, is a choice of TCP congestion algorithm (with Windows Server 2008+ using NewReno, and recent Linux reportedly using CUBIC); however, as of now, I don't have any information which demonstrates any advantage of any of them TCP-latency-wise (that is, with usual mixed-bag of clients, consisting of PCs and mobile phones); on the other hand, it is an area where development is still very much ongoing, so further changes are likely. Also note that as we cannot control client and there are tons of different clients with different TCP settings out there, any theoretical analysis becomes extremely complicated; the best we can do – is to try both in a real-world environment (the one with thousands of clients) and see whether there are any differences. Which makes yet another reason to have your code cross-platform.

When it comes to IPC (which you need to implement inter-FSMs interactions), both systems are very much the same. We'll discuss it in more detail in Chapter [[TODO]], but the rule of thumb is always the same: if you want it to be really fast – use shared memory, all the other mechanisms are inherently slower. Fortunately, shared memory is available on both Windows and Linux. If you don't care too much about achieving topmost available speed on the same machine – all common other methods (such as pipes and sockets) are readily available on both these platforms; and for our purposes, you won't need more than that. Fancy stuff such as completion ports and APC, *may in theory* provide some difference, but in practice for FSM-based architectures, it wasn't observed to provide any advantage (see also Chapter [[TODO]] for details). In short – IPC-wise, you will have quite a difficulty to find significant difference between Linux and Windows.[6]

As for file systems – for your Front-End Servers and Game Servers they don't really matter. Amount of file I/O on Front-End Servers and Game Servers should be kept negligible, mostly reading executables and configuration files; under these conditions all the differences between JFS, ZFS, ext4, and NTFS, won't play any significant role.

To summarize – technically (and from games perspective) both Windows and Linux kernel (and network stack) are doing really good job and (drivers aside) you're quite unlikely to observe significant differences because of these things. While you *may* see the some difference, if migrating from Windows to Linux or vice versa on the same hardware, experiences when migrating different server boxes will most likely be different, and I tend to attribute them (mostly) not to OS's as such, but rather to drivers, whose quality varies greatly. One potential exception is TCP congestion algorithm (that is, for TCP-based games), but its effects on games are yet to be seen.

---

[3] in practice, it is more complicated, as depending on the hardware, interrupt coming from NIC can be processed only on a dedicated CPU, which complicates things. However, this is normally not an OS restriction, but a hardware restriction, so there isn't much which can be done about it

[4] I also don't know of attempts to use different Linux schedulers for games, but based on reasoning above, I have my doubts whether they will make any difference

[5] I'm speaking about classical NUMA, with a node per socket

[6] ok, local sockets tend to be a tad slower on Windows than on Linux, but if you're really after speed, you still need to use shared memory, so it becomes pretty much a moot issue

C++ Compilers

If speaking about C++, a question of compiler becomes quite important. If you're going Windows route, your obvious choice would be MSVC, and for Linux it is probably GCC or LLVM/CLang. When comparing MSVC to GCC, GCC (especially GCC 4.8 and up) tends to produce better-quality code, which may amount (in practice) to as much as 5-10% overall performance difference.[7] This can be accounted for as 5-10% increase in number of servers you need to run your game; alternatively, you *may* try using MinGW (which is essentially GCC for Windows, I didn't try it myself, and can provide no warranties of any kind in this regard).

If comparing LLVM/CLang to GCC, in practice the difference (as of beginning of 2016) is pretty much negligible.

---

[7] individual functions can be *much* faster, but *on average* and taking into account such things as context switches and associated very severe cache misses (both

being inevitable on game servers), it is not that much as it may seem from "pure calculation" benchmarks

Is it Enough to Decide?

All the arguments above are repeated ad infinitum on the Internet, and as you see, I personally tend to favor Linux, but honestly, I don't really see that these arguments are sufficient to make a decision for our game servers (except, maybe, in some rare cases for interrupt-related stuff, see "Fast Network Packet Processing" section above). In practice, the real deal is usually about the following two reasons.

Free as in "Free Beer"

If your estimates show that you may need dozens and hundreds of servers – then the price of the license starts to hurt in a really bad way. And don't listen to those who say "Hey, RedHat license is about the same price as the Windows one, so it doesn't really matter"; in a price-conscious environment, you will likely use Debian, CentOS, or some other perfectly free distro, and will stay away from paying anything for Linux (except, maybe, for your DB server). And guess what – with zero price of free distros, there is absolutely no way for Windows to beat them price-wise, and even matching it looks very unlikely in foreseeable future.

> **"With zero price of free distros, there is absolutely no way for Windows to beat them price-wise, and even matching it looks very unlikely in foreseeable future.**

TCO wars

TCO
Total cost of ownership (TCO) is a financial estimate intended to help buyers and owners determine the direct and indirect costs of a product or system.
— Wikipedia —

At some point around 10 years ago, Microsoft has pushed an argument that despite license costs, a long-term cost of ownership (known as TCO) is lower for Windows (mostly due to higher salaries of Linux guys). This argument was one of the cornerstones of Microsoft's highly controversial "Get the Facts" campaign. I certainly and clearly don't agree with Microsoft on TCO, and am of a very firm opinion that at least for not-too-small datacenter-hosted systems, pretty much regardless of how you calculate it, costs of Linux boxes will be lower. Fortunately, there are quite a few bits of research out there, which confirm my experience a.k.a. gut feeling in this regard. These start (surprisingly) from a Microsoft-sponsored(!) IDC report back from 2002 [IDC]; while Microsoft has made a lot of buzz about Windows TCO advantage found by this report, it usually conveniently omitted that for web servers Linux TCO

was found to be lower (and our game servers are much more similar to web servers than to handling file or print jobs). Other studies supporting the same point of view include a report by Cybersource [Cybersource] and an IBM-sponsored report by RFG [RFG]. The latter one is especially interesting not only because it is exactly about application servers, and not only because it found Linux being 40% less expensive than Windows in the long run, but also because it has found that Linux admins, while more expensive, on average are able to handle more servers than their opposite numbers on the Windows side. To be honest, I need to mention that there are other reports which do claim that Microsoft TCO has an advantage, but also being honest, I need to say that I am not buying their arguments, agreeing with PCWorld's take on Linux-vs-Windows TCP for servers: "There's no beating Linux's total cost of ownership, since the software is generally free… The overall TCO simply can't be beat." [PCWorld]

To summarize the long text above:

Cost-wise, for game servers Linux is likely to provide a Significant Advantage

The importance of this observation, however, depends heavily on the number of servers you expect to run; if servers costs (not including traffic costs!) are going to be negligible, the whole line of argument about the server costs becomes much less important. More on it in "It is All about Money :-(" subsection below.

On ISPs and Windows-vs-Linux Cost

"For cheaper servers, the difference between Windows and Linux can eat as much as 50% of the server rental price (though for those servers

If you by any chance think "hey, we will rent servers from ISP anyway, so license costs won't matter", you're deadly wrong. Yes, you will most likely rent servers from ISPs (see Chapter [[TODO]] for details), but ISPs (no real surprise here) need to factor in the license price into their server rental price. As of the beginning of 2016, kind of typical price difference between CentOS two-socket "workhorse" server and the-same-hardware server with Windows Standard,[8] was roughly between $35/month and $50/month. For cheaper servers, the difference between Windows and Linux can eat as much as 50% of the server rental price (though for those servers which are more or less optimal price-performance-wise observed difference was closer to 20-30%). And with cloud providers, it won't get any better: an instance which costs $52/month with Linux, went up to $77/month with Windows (that's almost 50% on top of Linux (!)).

Time To Market: Familiarity to your Developers

If your game is computationally intensive, and you can support only a thousand players per server (and therefore, if your game is a success, you will need hundreds of servers to run your game), costs become a very important factor, difficult to fight with. In such cases, there is IMHO only one consideration that can trump lower costs for Linux boxes. This one is time to market for your game.

which are more or less optimal price-performance-wise observed difference was closer to 20-30%).

In other words, if you don't have anybody on the team who has ever developed anything for Linux, it is usually a good enough reason to use Windows on the server-side (and yes, it will work, provided that you're careful enough). It is not that to exploit lower cost of Linux boxes you need *all* of your developers to be Linux gurus (after all, you're much better when you can keep your FSMs "pure" anyway, and being "pure" pretty much implies being cross-platform), but if the whole your team has *zero* Linux experience – it will probably qualify as a valid reason to use Windows (that is, if you've already calculated the associated price tag and are ok with it).

An additional (and quite similar) time-to-market-related pro-Windows argument arises if your game is PC-only (or PC-and-Xbox-only). In this case, if you keep your server under Windows, you can have the same code running on server and client quite easily. While such logic has a grain of truth in it, personally I don't really like this line of reasoning. First of all, there isn't that much code to share to start with (it is mostly about the framework which runs FSMs, Communications- and Routing-related FSMs, and client-side prediction if applicable). Second, your FSMs need to be "pure" and cross-platform anyway (see above). Third, even the code outside of FSMs can be made cross-platform without going into vendor-lock-in stuff rather easily. And last but not least, having the same code run on different platforms, while taking additional time, allows to test your code better, improving overall code quality.

It is All about Money 🙁

At the end of the day, if your team consists primarily (but not exclusively) of Windows developers, *and* your game is computationally intensive enough to support only thousands (or even worse – hundreds) of players per server (and you can count on income per player being very limited), you're facing quite a difficult decision.

Usually, under such circumstances time-to-market considerations will override lower server costs, so it is all about the balance of Windows-vs-Linux guys and gals on your team. On the other hand, it is clearly a Business Decision which needs to be made by Business People and is outside of scope of this book. Our job as developers is just to warn business-minded people that Windows servers are going to cost more

than their Linux counterparts (and that server/cloud rental difference can be as large as 50%, though likely to be more in around 20-30%; note that these numbers *do not* include traffic, which will be the same regardless of the platform); the rest is not our decision anyway.

> "Usually, under such circumstances time-to-market considerations will override lower server costs

On the other hand, if your estimates show that you can handle a hundred thousands players per server – it looks unlikely that license costs will eat too much of your budget either way, so in this case you may be able to use Linux or Windows, whichever-platform-looks-better-for-you. The whole thing is all about numbers, pure and simple.

Mixed Bags

In the context of the discussion above, a logical question arises: "Can we develop our servers for Windows to get it faster, and migrate to Linux later to save costs?" The answer is "yes, you can, but you need to be extremely vigilant to avoid unnecessary dependencies". In general, QnFSM model with deterministic FSMs stimulates cross-platform development, so it might be not that difficult, but you still should remember about your intention to migrate later (this, for example, pretty much excludes using fancy-but-Windows-specific things such as completion ports; not that you really need them anyway for FSM-based architecture, see Chapter [[TODO]] for details).

It is also possible to run both Windows and Linux servers on the server-side not just as a part of migration from one to another one, but because of different reasons. Just to give an idea how it may happen: you may need to integrate with a payment provider, that requires you to use DLL, available only on Windows.[9] Ok, you can have that-provider's-FSM on a different server running under Windows, while having everything else running under Linux. Been there, done that.

---

[8] No "Essentials" edition was observed as a rental option, probably because of license restrictions

[9] and while I hate such providers, throwing them away is not my decision to make 🙁

Linux-vs-Windows: Time to Decide

To summarize my arguments above:

- if you want to use Linux because you're familiar with it – you're fine regardless of number of servers you'll need
- if you want to use Windows because you're familiar with it – take a look at the number of servers you expect to be using
  - the price of Windows license is far from negligible (making up to 50% of the rental cost of the server, though usually the price difference is more in 20-30% range), so it can make a significant difference for your ongoing costs after you launch the game
  - in this case, you may want to develop for Windows first (to speed time-to-market), and to migrate to Linux later
    - extreme vigilance to avoid being inadvertedly locked-in is required (see Chapter IV for details). On the other hand, FSMs tend to make dependency fighting simpler.
- if you're in doubt – use Linux, it is safer that way[10]

"if you want to use Linux because you're familiar with it – you're fine regardless of number of servers you'll need

Things to Keep in Mind: Windows

When developing for a specific platform, there are always platform-specific things which you need to keep in mind. For Windows my own favorite list of DO's and DON'Ts goes as follows (note that this is a language-agnostic list, for C++-specific stuff see Chapter [[TODO]]):

- DO fight 3rd-party dependencies. Unnecessary dependencies tend to make Windows less stable, less secure, the code less manageable, etc. Refer to Chapter IV, "DIY vs Re-use: In Search of Balance" for details on "what to DIY and what to re-use".
- DO fight 3rd-party dependencies. Re-use MUST NOT be taken lightly, and extreme vigilance is required.
- DO fight 3rd-party dependencies. In spades. While all the developers are prone to taking some "nice" 3rd-party component and to using it without telling anybody, from my experience Windows developers are more likely to do it than Linux ones.
- DON'T use .NET-based stuff unless absolutely necessary. .NET in production will cause you quite a lot of trouble. If you want to use .NET as your own platform – well, at least you (I hope) know why you're using it, and will be able to configure it to minimize the impact. If you're programming in not-a-.NET-language, running .NET

"DO fight 3rd-party dependencies. In spades.

unless absolutely necessary, is a recipe for several different disasters (ranging from security problems to run-away 3rd-party not-really-necessary .NET component eating all-the-available-resources).
    - Stay away from web services (that is, unless you're into Web-Based Architecture), at the very least for time-critical pieces. In general, any technology that has a blocking RPC interface, should be avoided, as blocking inter-process (and even worse, inter-server) calls don't fit well into our perfectly-non-blocking no-unnecessary-context-switching highly-optimized FSMs, and will cause significant performance degradation compared to them.
- Stay away from COM.[11] COM components have two pretty bad properties. First, it is yet another technology based on blocking RPC calls (see above about them). Second, if you're using COM for your own components – it is quite silly (ok, unless you're using Visual Basic), and if you're using it for 3rd-party components – it is a 3rd-party dependency, you should fight as stated above. Consider an offense of using DCOM as just an aggravated form of the offense of using COM.

---

[10] "safer" here can be interpreted in several different ways: from "a little bit safer security-wise" to "safer in case if your profits are much lower than expected, so price of the servers becomes more critical".
[11] yes, I know lots of people consider COM long-dead; unfortunately, it is not

Things to Keep in Mind: Linux

Linux also has it's fair share of DO's and especially DON'Ts. My favourite ones are as follows:

- DO fight 3rd-party dependencies. While from my experience, the danger of 3rd-party dependencies is lower for Linux than for Windows, it still exists.
- DON'T program for one single distribution. Your code should be generic enough to allow jumping around different distros; there is no reason to depend on package manager or exact directory structure. If you need these badly, move this kind of stuff into config files (or into rarely-executed shell scripts), so your admins can adjust directories if necessary.
    - As long as we're speaking about Linux (not including BSD), all you really need to use on your Game Server is Linux kernel and glibc. Both will be very much the same for all the distros (with the only difference being kernel/glibc version).

"DON'T program for one single distribution.

    - If considering *BSD family, they are somewhat different, but as long as

you're using POSIX APIs (and that covers 99% of what you'll really want in practice[12]), the differences are negligible

- DON'T use shell scripts for frequently-performed tasks. While an occasional shell script to install your daemon is fine, invoking shell 1000 times a second is rarely a good idea.
    - Pretty much the same goes for cron – DON'T try to get around cron's 1-minute restriction by playing tricks such as running 60 cron jobs every minute, with the first job waiting for one second, the second one waiting for another second, and so on.
- DON'T think that threads are much faster than processes on Linux (at least not that much as they are on Windows). And BTW, it is not that threads on Linux are slow, it is that process creation (fork()) is fast. On the other hand, you may still want to use threads if you're after cross-platform development.

---

[12] the remaining 1% includes things such as epoll/kqueue

Things to Keep in Mind: All Platforms

In addition, there are a few things to remember about, which apply regardless of the platform you're developing for:

- DON'T use platform-specific APIs within your FSMs (see below about using them outside of FSMs). Leaving aside a few narrow exceptions, your FSMs need to stay "pure" (see Chapter V for discussion of the associated benefits), and platform-specific APIs is #1 enemy of the code being "pure".
- DO consider cross-platform code even outside FSMs. The whole QnFSM can be written in a fully cross-platform manner.[13] Even if you find platform-specific optimizations, it is better to have a purely cross-platform version (at the very least, to have a baseline to compare your optimizations against).[14]

"DO consider cross-platform code even outside FSMs.

---

[13] been there, done that

[14] I've seen quite a few "platform-optimized" versions which were actually slower than cross-platform ones, and even more platform-optimized stuff which was exactly on par with the cross-platform one

[[To Be Continued...

This concludes beta Chapter VI(c) 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 VI(d), "Modular Architecture: Server-Side. Programming Languages.]]

[–] References

[Corbet2013] Jonathan Corbet, "NUMA scheduling progress"
[Lameter2013] Christoph Lameter, "NUMA (Non-Uniform Memory Access): An Overview"
[IDC] "Windows 2000 Versus Linux in Enterprise Computing"
[Cybersource] "Linux vs Windows. Total Cost of Ownership Comparison"
[RFG] "TCO for Application Servers: Comparing Linux with Windows and Solaris"
[PCWorld] Katherine Noyes, "Five Reasons Linux Beats Windows for Servers"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« ***Chapter VI(b). Server-Side Architecture. Front-End Servers a...***

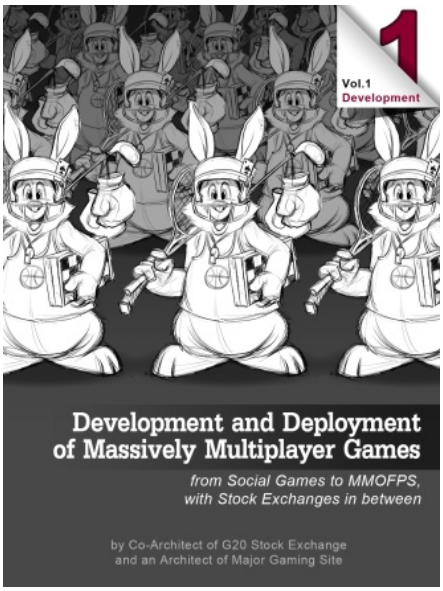
Asynchronous Processing for Finite State Machines/Actors:... »

Filed Under: Distributed Systems, Programming, System Architecture
Tagged With: game, Linux, multi-player, server, Windows

Copyright © 2014-2016 ITHare.com

Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between)

posted January 11, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

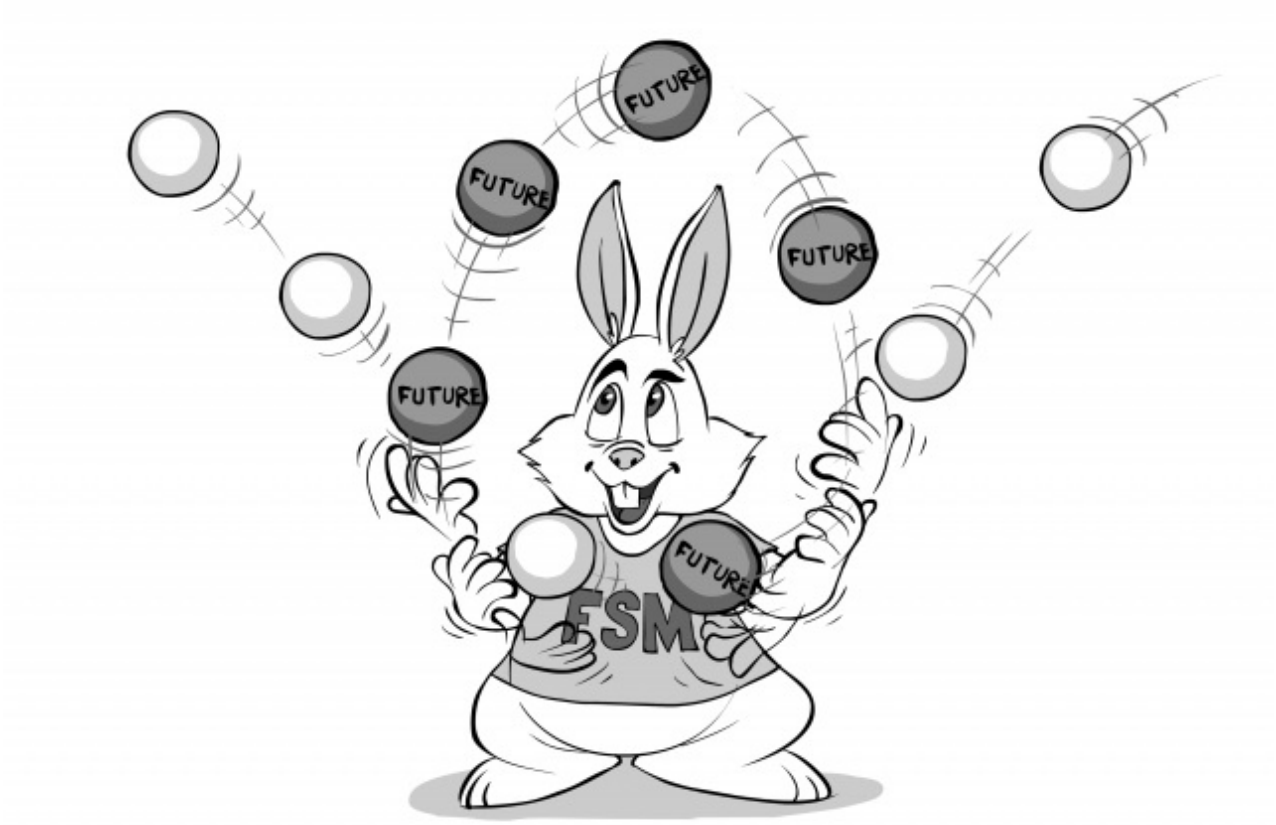


[[*This is Chapter VI(d) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further 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.]]

[[I was planning the next part of Chapter VI to be about server-side programming languages, but have found that to speak about them, it would be better to describe a bit more about FSMs and an important part of them – futures and exception-related FSM-specific stuff. My apologies for this change in plans, and I hope that the part about server-side programming languages will be the next one]]

When programming Finite State Machines (FSMs, with Erlang/Akka-style Actors, or more generally – non-blocking event-driven programs, being very close) in a really non-blocking manner, two practical questions arise: "how to deal with communications with the other A in a non-blocking way", and "what to do with timed actions".



For the purposes of this section, we'll use C++ examples; however, leaving aside syntax, most of the reasoning here will also apply to any other modern programming language (with an obvious notion that the part on functional-style implementation will need support for lambdas); one obvious example is JavaScript as it is used in Node.js (more on it below).

Also, for the purpose of our examples, we assume that we have some kind of IDL compiler (more on in in Chapter VII), which takes function definitions and produces C++ stubs for them. The idea behind an IDL is to have all the inter-FSM communications defined in a special Interface Definition Language (see examples below), with an IDL compiler producing stubs (and relevant marshalling/unmarshalling code) for our programming language(s). IDL serves two important purposes: first, it eliminates silly-but-annoying bugs when manual marshalling is done differently by sender and receiver; second, it facilitates cross-language interactions.

IDL

Interface definition language (IDL) is a specification language used to describe a software component's application programming interface (API). IDLs describe an interface in a language-independent way, enabling

Take 1. Naïve Approach: Plain Events (will work, but is Plain Ugly)

Both inter-FSM communication and timed actions can be dealt with without any deviation from FSM/Actor model, via introducing yet another couple of input events. Let's say that we have a non-blocking RPC call from FSM A to a FSM B, which returns a value. RPC call translates into a message coming from

FSM A to FSM B (how it is delivered, is a different story, which will be discussed in Chapter [[TODO]]). FSM B gets this message as an input event, processes it, and sends another message to FSM A. FSM A gets this message as an input event, and performs some actions (which are FSM-specific, so FSM writer needs to specify them).

In a similar manner, whenever we're scheduling a timer, it is just a special timer event which will be delivered by FSM framework (="the code outside of FSM") to FSM more or less around requested time.

> **communication between software components that do not share one language**
>
> — *Wikipedia* —

First, let's consider a very simple example. Let's say our Game World FSM needs to report that our player has gained level, to DB (so that even if our Game World crashes, the player won't lose level, see "Containment of Game World server failures" section above for further discussion). In this case, our IDL may look as follows:

```
1  void dbLevelGained(int user_id, int level);
2    //ALL RPC calls are NON-BLOCKING!!
```

After this IDL is compiled, we may get something like:

```
1  //GENERATED FROM IDL, DO NOT MODIFY!
2  int dbLevelGained_send(FSMID fsm_id, int user_id, int level);
3    //sends a message to fsm_id
4    //returns request id
```

Then, calling code in FSM A may look like this:

```
1  dbLevelGained(db_fsm_id,user_id,level);
```

So far, so simple, with no apparent problems in sight. Now, let's see what happens in a more elaborated "item purchase" example. Let's say that we want to show player the list of items available for purchase (with items for which he has enough money on the account, highlighted), allow her to choose an item, get it through DB (which will deduct item price from player's account and add item to his DB inventory), and add the item to the game world.

Don't worry if you think that the code in Take 1 is ugly. It is. Skip to OO-based and function-based versions if this one affects your sensibilities

To do this, our IDL will look as follows:

```
int dbGetAccountBalance(int user_id);
list<StoreItem> dbGetStoreItems();
void dbBuyItemFromAccount(int user_id, ITEMID item);
  //MUST be a separate call to ensure data integrity without external locking,
  //  see "Containment of Game World server failures" subsection for discussion

int clientSelectItemToBuy(list<StoreItem>,int current_balance);
```

After this IDL is compiled, we may get something like:

```
//GENERATED FROM IDL, DO NOT MODIFY!
#define DB_GET_ACCOUNT_BALANCE 123
#define DB_GET_STORE_ITEMS 124
#define DB_BUY_ITEM_FROM_ACCOUNT 125
#define CLIENT_SELECT_ITEM_TO_BUY 126

int dbGetAccountBalance_send(FSMID fsm_id, int user_id);
  //sends a message, returns request_id
pair<bool,int> dbGetAccountBalance_recv(Event& ev, int request_id);
  //return.first indicates if incoming message matches request_id
int dbGetStoreItems_send(FSMID fsm_id);
pair<bool,list<StoreItem>> dbGetStoreItems_recv(Event& ev, int request_id);
int dbBuyItemFromAccount_send(FSMID fsm_id, int user_id, ITEMID item);
pair<bool,bool> dbBuyItemFromAccount_recv(Event& ev, int request_id);

int clientSelectItemToBuy_send(FSMID fsm_id, const list<StoreItem>& items,
  int current_balance);
pair<bool,int> clientSelectItemToBuy_recv(Event& ev, int request_id);
```

And, our code in FSM A will look like the following (this is where things start getting ugly):

```
//WARNING: SEVERELY UGLY CODE AHEAD!!
void MyFSM::process_event(Event& ev) {
  switch( ev.type ) {
    case SOME_OTHER_EVENT:
      //...
      //decided to make a call
      int request_id = dbGetAccountBalance_send(db_fsm_id,user_id);
      account_balance_requests.push(pair<int,int>(request_id,user_id));
        //account_balance_requests is a member of MyFSM
        //need it to account for multiple users requesting purchases
        //  at the same time
      //...
      break;

    case DB_GET_ACCOUNT_BALANCE:
      for(auto rq:account_balance_requests) {
        auto ok = dbGetAccountBalance_recv(ev, rq.first);
        if(ok.first) {
          int user_id = rq.second;
          int balance = ok.second;
          //got account balance, let's get list of items now
          int request_id2 = dbGetStoreItems_send(db_fsm_id);
          store_items_requests.push(
```

```
        pair<int,pair<int,int>>(request_id2,pair<int,int>(user_id,balance)));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case DB_GET_STORE_ITEMS:
     for(auto rq:store_item_requests) {
      auto ok = dbGetStoreItems_recv(ev, rq.first);
      if(ok.first) {
       pair<int,int> user_id_and_balance = rq.second;
       list<StoreItem>& items = ok.second;
       //got everything client needs, let's send it to client now
       int request_id3 = clientSelectItemToBuy_send(user_fsm_id,
        items,user_id_and_balance.second);
       client_select_items_to_buy_requests.push(
        pair<int,int>(request_id,user_id));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case CLIENT_SELECT_ITEM_TO_BUY:
     for(auto rq:store_item_requests) {
      auto ok = clientSelectItemsToBuy_recv(ev, rq.first);
      if(ok.first) {
       int user_id = rq.second;
       ITEMID selected_item = ok.second;
       //got client selection, let's try buying now
       int request_id4 = dbBuyItemFromAccount_send(db_fsm_id,
        user_id,selected_item);
       buy_item_requests.push(pair<int,pair<int,ITEMID>>(
        request_id,pair<int,int>(user_id,selected_item)));
       break;
      }
      MY_ASSERT(false,"Cannot happen");
       //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
     }
     break;

   case DB_BUY_ITEM_FROM_ACCOUNT:
     for(auto rq:store_item_requests) {
      auto ok = dbBuyItemFromAccount_recv(ev, rq.first);
      if(ok.first) {
       pair<int,ITEMID> user_id_and_item = rq.second;
       bool item_ok = ok.second;
       //got DB confirmation, let's modify our game world now
       players[user_id].addItem(user_id_and_item.second);
       //phew
       break;
      }
      MY_ASSERT(false,"Cannot happen");
```

```
79          MY_ASSERT(false, "Cannot happen");
80            //throws an exception, more on MY_ASSERT in Chapter[[TODO]]
81          }
82          break;
83        }
84      }
```

If you feel that this code has been beaten with an ugly stick – that's because it is

Over 60 lines of code with only about 5 being meaningful (and the rest being boilerplate stuff) is pretty bad. Not only it takes a lot of keystrokes to write, but it is even worse to read (what really is going on is completely hidden within those tons of boilerplate code). And it is very error-prone too, making maintenance a nightmare. If such a thing happens once for all your 1e6-LOC game – that's ok, but you will need these things much more than once. Let's see what can we do to improve it.

LOC

Lines of Code is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code

— Wikipedia —

Take 2. OO-Style: Less Error-Prone, but Still Unreadable

In OO-style, we will create a Callback class, will register it with our FSM, and then it will be FSM framework ("the code outside of FSMs") dealing with most of the mechanics within. Rewriting our "item purchase" example int OO-style will change the whole thing drastically. While IDL will be the same, both generated code and calling code will look very differently. For OO-style asynchronous calls, stub code generated from IDL may look as follows:

```
1   //GENERATED FROM IDL, DO NOT MODIFY!
2   void dbGetAccountBalance_send(FSM* fsm, /* new */ Callback* cb,
3     FSMID target_fsm_id, int user_id);
4     //sends a message, calls cb->process_callback() when done
5   int dbGetAccountBalance_parsereply(Event& ev);
6
7   void dbGetStoreItems_send(FSM* fsm, /* new */ Callback* cb, FSMID target_fsm_id);
8   list<StoreItem> dbGetStoreItems_parsereply(Event& ev);
9   void dbBuyItemFromAccount_send(FSM* fsm, /* new */ Callback* cb,
10    FSMID target_fsm_id, int user_id, ITEMID item);
11  bool dbBuyItemFromAccount_parsereply(Event& ev);
12  void clientSelectItemToBuy_send(FSM* fsm, /* new */ Callback* cb,
13    FSMID target_fsm_id, const list<StoreItem>& items, int current_balance);
14  ITEMID clientSelectItemToBuy_parsereply(Event& ev);
```

And our calling code may look as follows:

```
1   //LESS ERROR-PRONE THAN TAKE 1, BUT STILL UNREADABLE
2   //TO BE AVOIDED IF YOUR COMPILER SUPPORTS LAMBDAS
3   class BuyItemFromAccountCallback : public Callback {
```

```
class BuyItemFromAccountCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
  ITEMID item;

  public:
  BuyItemFromAccountCallback(MyFSM* fsm_,int user_id_, ITEMID item_)
  : fsm(fsm_),user_id(user_id_), item(item_)
  {
  }
  void process_callback(Event& ev) override {
    bool ok = dbBuyItemFromAccount_parsereply(ev);
    if(ok)
      fsm->players[user_id].addItem(user_id_and_item.second);
  }
};
class ClientSelectItemToBuyCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;

  public:
  ClientSelectItemToBuyCallback(MyFSM* fsm_,int user_id_)
  : fsm(fsm_),user_id(user_id_)
  {
  }
  void process_callback(Event& ev) override {
   ITEMID item = clientSelectItemToBuy_parsereply(ev);
   dbBuyItemFromAccount_send(fsm,
     new BuyItemFromAccountCallback(fsm,user_id,item),
     fsm->getDbFsmId(), user_id, item);
  }
};
class GetStoreItemsCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
  int balance;

  public:
  GetStoreItemsCallback(MyFSM* fsm_,int user_id_, int balance_)
  : fsm(fsm_),user_id(user_id_), balance(balance_)
  {
  }
  void process_callback(Event& ev) override {
    list<StoreItem> items = dbGetStoreItems_parsereply(ev);
    clientSelectItemToBuy_send(fsm,
      new ClientSelectItemToBuyCallback(fsm, user_id),
      fsm->getClientFsmId(user_id), items, balance);
  }
};

class GetAccountBalanceCallback : public Callback {
  private:
  MyFSM* fsm;
  int user_id;
```

```
59      int user_id;
60
61    public:
62    GetAccountBalanceCallback(MyFSM* fsm_,int user_id_)
63    : fsm(fsm_), user_id( user_id_ )
64    {
65    }
66    void process_callback(Event& ev) override {
67      int balance = dbGetAccountBalance_parsereply(ev);
68      dbGetStoreItems_send(fsm,
69        new GetStoreItemsCallback(fsm,user_id, balance), fsm->getDbFsmId() );
70    }
71  };
72
73  void MyFSM::process_event(Event& ev){
74    switch( ev.type ) {
75      case SOME_OTHER_EVENT:
76        //...
77        //decided to make a call
78        dbGetAccountBalance_send(this,
79          new GetAccountBalanceCallback(this, user_id), db_fsm_id, user_id);
80        //...
81         break;
82    }
83  }
```

This one is less error-prone than the code in Take 1, but is still very verbose, and poorly readable. For each meaningful line of code there is still 10+ lines of boilerplate stuff (though it is easier to parse it out while reading, than for Naïve one).

In [Facebook] it is named “callback hell”. Well, I wouldn’t be that categoric (after all, there was life before 2011), but yes – it is indeed very annoying (and poorly manageable). If you don’t have anything better than this – you might need to use this kind of stuff, but if your language supports lambdas, the very same thing can be written in a much more manageable manner.

Take 3. Lambda Continuations to the rescue! Callback Pyramid

As soon as we get lambda functions (i.e. more or less since C++11), the whole thing becomes much easier to write down. First of all, we could simply replace our classes with lambda functions. In this case, code generated from the very same IDL, may look as follows:

```
//GENERATED FROM IDL, DO NOT MODIFY!
void dbGetAccountBalance(FSM* fsm, FSMID target_fsm_id, int user_id,
  std::function<void(int)> cb);
  //sends a message, calls cb when done

void dbGetStoreItems(FSM* fsm, FSMID target_fsm_id,
  std::function<void(const list<StoreItem>&)> cb);
void dbBuyItemFromAccount(FSM* fsm, FSMID target_fsm_id, int user_id, ITEMID item
  std::function<void(book ok)> cb);
void clientSelectItemToBuy(FSM* fsm, FSMID target_fsm_id,
  const list<StoreItem>& items, int current_balance,
  std::function<void(ITEMID item)> cb);
```

And calling code might look as follows:

```
//inside MyFSM::process_event():
//...
//decided to make a call
dbGetAccountBalance(this,db_fsm_id,user_id,
  [=](int balance) {
    //this lambda is a close cousin of
    //  Take2::GetAccountBalanceCallback
    //  You may think of lambda object created at this point,
    //    as of Take2::GetAccountBalanceCallback
    //    automagically created for you
    dbGetStoreItems(this,db_fsm_id,
      [=](const list<StoreItem>& items) {
        //this lambda is a close cousin of
        // Take2::GetStoreItemsCallback
        clientSelectItemToBuy(this,user_fsm_id,items,balance,
            //here, 'this', 'user_fsm_id', and 'balance' are 'captured'
            //  from the code above
          [=](ITEMID item) {
            //this lambda is a close cousin of
            //  Take2::ClientSelectItemToBuyCallback
            dbBuyItemFromAccount(this,db_fsm_id,user_id,item_id,
              [=](bool ok) {
                //this lambda is a close cousin of
                //  Take2::BuyItemFromAccountCallback
                if(ok) {
                  players[user_id].addItem(item_id);
                }
              }
            );
          }
        );
      }
    );
  }
);
```

Compared to our previous attempts, such a “callback pyramid” is indeed a big relief. Instead of previously observed 50+ lines of code for meaningful 5 or so (with meaningful ones scattered around), here we have just about 2 lines of overhead per

each meaningful line (instead of previous 10(!)), and also all our meaningful lines of code are nicely gathered in one place (and in their right order too). Phew. With all my dislike to using lambdas just for the sake of your code being "cool" and functional, this is one case when using lambdas makes very obvious sense (despite the syntax looking quite weird).

In fact, this code is very close to the way Node.js programs handle asynchronous calls. Actually, as it was mentioned in Chapter V [[TODO!: mention it there]] the whole task we're facing with our QnFSMs (which is "event-driven programming with a completely non-blocking API") is almost exactly the same as the one for Node.js, so there is no wonder that the methods we're using, are similar.

On Continuations

"Don't even think of converting *all* of your code to a so-called Continuation-Passing-Style

Those lambdas we're using here, are known as "continuations". In general, "continuation" is a thing, which says what we should do when we reach certain point within our logical flow. To make our FSMs (and Node.js) work – continuations are the only feasible way to do it (in fact, our Take 1 and Take 2 also implemented continuations, albeit in an unusual way).

However, don't even think of converting *all* of your code to a so-called Continuation-Passing-Style[1] (the one with an explicit prohibition for any function to return any value, instead each and every function taking additional function parameter to be called with would-be return value). Full conversion to continuation-passing-style will make your code significantly less readable, and will hit your performance too. Think of our "callback pyramid" code above not as a final proof of lambdas being the-utlimate-solution-to-all-your-problems, but as of a useful pattern, which can be used to simplify coding *in this specific scenario*.

Exceptions

Now, as we got rid of those ugly Take 1 and Take 2 (where any additional complexity would make them absolutely incomprehensible), we can start thinking about adding exceptions to our code. Indeed, we can add exceptions to the "callback pyramid", by adding (to each of RPC stubs and each of the lambdas) another lambda parameter to handle exceptions (corresponding to usual 'catch' statement). Keep in mind that to provide usual try-catch semantics (with topmost-function exception handler catching all the stuff on all the levels), we need to pass this 'catch' lambda downstream:

```
dbGetAccountBalance(this,db_fsm_id,user_id,
 [=](int balance, std::function<void(std::exception&)> catc) {
  dbGetStoreItems(this,db_fsm_id,
   [=](const list<StoreItem>& items, std::function<void(std::exception&)> catc) {
    clientSelectItemToBuy(this,user_fsm_id,items,balance,
     [=](ITEMID item, std::function<void(std::exception&)> catc) {
      dbBuyItemFromAccount(this,db_fsm_id,user_id,item_id,
       [=](bool ok, std::function<void(std::exception&)> catc) {
        if(ok) {
         players[user_id].addItem(item_id);
        }
       }
      ,catc);
     }
    ,catc);
   }
  ,catc);
 },
 [=](std::exception&) {//'catch'
  //do something
 }
);
```

As we can see, while handling exceptions with 'callback pyramid' is possible, it certainly adds to boilerplate code, and also starts to lead us towards the Ugly Land ☹ .

Limitations

For the 'callback pyramid' above, I see two substantial limitations. The first one is that adding exceptions, while possible, adds to code ugliness and impedes readability (see example above).

The second limitation is that with 'callback pyramid' it is not easy to express the concept of "wait for more than one thing to complete" , which leads to unnecessary sequencing, adding to latencies (which may or may not be a problem for your purposes, but still a thing to keep in mind).

On the other hand, as soon as we have lambdas, we can make another attempt to write our asynchronous code, and to obtain the code which is free from these two limitations.

> "with 'callback pyramid' it is not easy to express the concept of 'wait for more than one thing to complete' , which leads to unnecessary sequencing, adding to

---

[1] which is the first thing Google throws at you when you're typing in "node.js continuation"

Take 4. Futures

While lambda-based 'callback pyramid' version is indeed a Big Fat Improvement over our first two takes, let's see if we can improve it further. Here, we will use a concept known as "futures" (our FSMFuture is similar in concept, but different in implementation, from std::future, boost::future, and folly::Future, see "Similarities and Differences" section below for discussion of differences between the these). In our interpretation, "future" is a value which is already requested, but not obtained yet. With such "futures", IDL-generated code for the very same "item purchase" example, may look as follows :

latencies (which may or may not be a problem for your purposes, but still a thing to keep in mind).

```
FSMFuture<int> dbGetAccountBalance(FSM* fsm, FSMID db_fsm_id, int user_id);
FSMFuture<list<StoreItem>> dbGetStoreItems(FSM* fsm, FSMID db_fsm_id);
FSMFuture<void> dbBuyFromAccount(FSM* fsm, FSMID db_fsm_id,
  int user_id, ITEMID item);

FSMFuture<ITEMID> clientSelectItemToBuy(FSM* fsm, FSMID client_fsm_id,
  list<StoreItem>, int current_balance);
```

And the calling code will look along the lines of:

```
//inside MyFSM::process_event():
//...
//decided to make a call
FSMFuture<int> balance = dbGetAccountBalance(this, db_fsm_id ,user_id);
  //sends non-blocking RPC request
FSMFuture<list<StoreItem>> items = dbGetStoreItems(this, db_fsm_id);
  //sends non-blocking RPC request

// all further calls don't normally do anything right away,
//   just declaring future actions
//   to be performed when the results are ready

//declare that we want to wait for both non-blocking RPC calls to complete
FSMFutureBoth<int,list<StoreItem>> balance_and_items(this, balance, items);

//declare what we will do when both balance and items are ready
FSMFuture<ITEMID> clientSelection = balance_and_items.then(
  [=]() {
    return clientSelectItemToBuy(this, user_fsm_id,
      balance_and_items.secondValue(), balance_and_items.firstValue());
  }
).exception(
    //NOTE that when we're attaching exception handler to a future,
    //  FSMFuture implementation can also apply it
    //    to all the futures 'downstream'
    //    unless it is explicitly overridden
  [=]() {
    //handle exception
  }
);

//declare what we will do when
FSMFuture<bool> purchase_ok = clientSelection.then(
  [=]() {
    return dbBuyFromAccount(this, db_fsm_id, user_id, clientSelection.value());
  }
);

purchase_ok.then(
  [=]() {
    players[user_id].addItem(clientSelection.value());
  }
);
```

While being a bit more verbose than lambda-based “call pyramid” version, at least for me personally it is more straightforward and more readable. Also, as a side bonus, it allows to describe scenarios when you need two things to continue your calculations (in our example – results of dbGetAccountBalance() and dbGetStoreItems()) quite easily, and without unnecessary sequencing which was present in all our previous versions. In other words, the future-based version as written above, will issue two first non-blocking RPC requests in parallel, and then will wait for both of them before proceeding further (opposed to all previous versions issuing the same calls sequentially and unnecessary losing on latency). While writing the same parallel logic within the previous takes is possible (except maybe for Take 3), it would result in a

code which is too ugly to deal with and maintain at application level; with futures, it is much more straightforward and obvious.

Similarities and Differences

All the different takes are similar

It should be noted that for our "item purchase" example (and actually any other sequence-of-calls scenario), all our versions are very similar to each other, with most of the differences being about "syntactic sugar". On the other hand, when faced with code from Take 1, and equivalent one from Take 4, I would certainly prefer the latter one 🙂 .

> **"Performance-wise, the differences between different code versions discussed above will be negligible for pretty much any conceivable scenario.**

Performance-wise, the differences between different code versions discussed above will be negligible for pretty much any conceivable scenario. Consistently with Erlang/Akka/Node.js approaches, our unit of processing is always a message/event. Events as such roughly correspond to context switches, and context switches are quite expensive beasts (for x64 – very roughly of the order of 10'000 CPU clocks, that is, if we account for cache reloads, YMMV, batteries not included).[2] So, even if we're using our non-blocking RPCs to off-load some calculations to different threads (and not for inter-server communications, where the costs are obviously higher), the costs of each message/event processing are quite high, and things such as dynamic dispatching or even dynamic allocations won't be large enough to produce any visible performance difference.

Differences from std::future etc.

Traditionally, discussions about asynchronous processing are made in the context of "Off-loading" some calculations into a different thread, doing some things in parallel, and waiting for the result (at the point where it becomes necessary to calculate things further). This becomes particularly obvious when looking at std::future: among other things, it has get() method, which waits until the future result is ready (the same goes for boost::future, and folly::Futures have wait() method which does pretty much the same thing). As our FSMs in QnFSM model are completely non-blocking, we are not allowed to have things such as std::future::get() or folly::Future::wait().

Whenever an std::future (or folly::Future) completes computation, it reports back to original future object via some kind of inter-thread notification. Also for this kind of futures, the code in callbacks/continuations MAY (and usually will) be called from a different thread, which means that callbacks are normally not allowed to interact with the main thread (except for setting a value within the future).

Similarities to Node.js

In contrast, our asynchronous processing is based on the premise that whenever a future is available, it is delivered as a yet another message to FSM::process_event(). It stands for all our four different versions of the code (with the differences, while important practically, being more of syntactic nature). As a consequence, all versions of our code guarantee that all our callbacks (whether lambda or not) will always be called from the same thread,[3] which means that

we are allowed to use FSM object and all it's fields from all our callbacks (lambdas or not) and without any thread synchronization[4]

It allows to handle much more sophisticated scenarios than that of linear calculation/execution. For example, if in our "item purchase" example there is a per-world limit of number of items of certain type, we MAY add the check for number of items which are already present within our world, into processing of clientSelectItemToBuy reply, to guarantee that the limit is not exceeded. If there is only one item left, but there are many clients willing it, all of them will be allowed to go up until to clientSelectItemToBuy, but those who waited too long there, will get an error message at the point after clientSelectItemToBuy returns. All this will happen without any thread synchronization.

"All of this will happen without any thread synchronization.

From this point of view, our futures (as well as the other our Takes on the asynchronous communications) are more similar to Node.js approach (where all the callbacks are essentially made within the same thread, so no thread synchronizations issues arise).

Alternatively, we can see Take 3 and Take 4 as a quite special case of coroutines/fibers. In such interpretation, we can say that there is an implicit coroutine "yield" before each RPC call in a chain, which allows other messages to be processed while we're waiting for the reply. Still, when a reply comes back, we're back in the same context and in the same thread where we were before this implicit "yield" point.

---

[2] context switches being that expensive is one reason why off-loading micro-operations to other threads doesn't work (in other words: don't try to off-load lone "int a+ int b" to a different thread, it won't do any good)

[3] strictly speaking, in some fairly unusual deployment scenarios there can be exceptions to this rule, but no-synchronization needed claim always stands

[4] which is why we have significant simplification for our lambda version compared to the one in [Facebook]

On serializable lambdas in C++

To have all the FSM goodies (like production post-mortem etc.), we need to be able to serialize those captured values within lambdas (this also applies to FSMs). For most of the languages out there, pretty much everything is serializable, including lambda objects, but for C++, serializing lambda captured values is not easy 🙁 .

The best way of doing it which I currently know, is the following:

- write and debug the code written as in the examples above. It won't give you things such as production post-mortem, or full FSM serialization, but they're rarely needed at this point in development (if necessary, you can always go via production route described below, to get them)
- add prefix such as SERIALIZABLELAMBDA before each such lambda; define it to an empty string (alternatively, you may use specially formatted comment, but I prefer empty define as more explicit)
- have your own pre-processor which takes all these SERIALIZABLELAMBDAs and generates code similar to that of in Take 2, with all the generated classes implementing whatever-serialization-you-prefer (and all the generated classes derived from some base *class SerializableLambda* or something). Complexity of this pre-processor will depend on the amount of information you provide in your SERIALIZABLELAMBDA macro:
  - if you write it as SERIALIZABLELAMBDA(int i, string s), specifying all the captured variables with their types once again, then your pre-processor becomes trivial
  - if you want to write it as SERIALIZABLELAMBDA w/o parameters, it is still possible, but deriving those captured parameters and their types can be not too trivial
  - which way to go, is up to you, both will work
- in production mode, run this pre-processor before compiling
- in production mode, make sure that RPC functions don't accept std::function (accepting *class SerializableLambda* instead), so that if you forget to specify SERIALIZABLELAMBDA, your code won't compile (which is better than if it compiles, and fails only in runtime)

TL;DR for Asynchronous Communications in FSMs

- We've discussed in detail asynchronous RPC calls, but handling of timer-related messages can be implemented in a very similar way
- As our FSMs are non-blocking, being asynchronous becomes the law (exactly as for Node.js)
- You will need IDL (and IDL compiler) one way or another (more on it in Chapter [[TODO]])

- Ways of handling asynchronous stuff in FSMs are well-known, but are quite ugly (see Take 1 and Take 2)
- With introduction of lambdas, it became much better and simpler to write and understand (see Take 3 and Take 4)
- Futures can be seen as an improvement over "call pyramid" use of lambdas (which is consistent with findings in [Facebook])
    - in particular, it simplifies handling of "wait-for-multiple-results-before-proceeding" scenarios
    - FSM futures, while having the concept which is similar to std::future and folly:Future, are not identical to them
        - in particular, FSM futures allow interaction with FSM state from callbacks without any thread synchronization
- To get all FSM goodies in C++, you'll need to implement serializing lambdas, see details above

FSMs and Exceptions

One more FSM-related issue which was uncovered until now, is related to subtle relations between FSMs and exceptions. Once again, most of our discussion (except for the part marked "C++-specific") will apply to most programming languages, but examples will be given in C++.

Validate-Calculate-Modify Pattern

One very important practical pattern for FSMs, is Validate-Calculate-Modify. The idea behind is that most of the time, when processing incoming event/message within our FSM, we need to do the following three things:

- **Validate.** check that the incoming event/message is valid
- **Calculate.** calculate changes which need to be made to the state of our FSM
- **Modify.** Apply those calculated changes.

"Effects of any exception happening before Modify stage are trivial: as we didn't modify anything, any exception will

This pattern has quite a few useful applications; however, the most important uses are closely related to exceptions. As long as we don't modify state of our FSM within Validate and Calculate stages, effects of any exception happening before Modify stage are trivial: as we didn't modify anything, any exception will lead merely to ignoring incoming message (without any need to rollback any changes, as there were none; handling of on-stack allocations depends on the programming language and is discussed below), which exactly what is necessary most of the time (and this has some other interesting uses, see "Exception-based Determinism" section below). And Modify stage is usually

trivial enough to avoid vast majority of the exceptions.

Enforcing const-ness for Validate-Calculate-Modify (C++-specific)

lead merely to ignoring incoming message, without any need to rollback any changes, as there were none

To rely on "no-rollback-necessary" exception property within Validate-Calculate-Modify pattern, it is important to enforce immutability of FSM state before Modify stage. And as it was noted in [[GDC2015 – TODO!]], no rule is good if it is not enforced. Fortunately, at least in C++ we can enforce immutability relatively easily (that is, for reasonable and non-malicious developers). But first, let's define our task. We want to be able to enforce const-ness along the following lines:

```
1  void MyFSM::process_event(Event& ev) {
2    ///VALIDATE: 'this' is const
3    //validating code
4
5    //CALCULATE: 'this' is still const
6    //calculating code
7
8    //MODIFY: 'this' is no longer const
9    //modifying code
10 }
```

To make it work this way, for C++ I suggest the following (reasonably dirty) trick:

```
1  void MyFSM::process_event(Event& ev) const {
2    //yes, process_event() is declared const(!)
3
4    //VALIDATE: 'this' is enforced const
5    //validating code
6    //CALCULATE: 'this' is still enforced const
7    //calculate code
8
9    //MODIFY:
10   MyFSM* fsm = modify_stage_fsm();
11     //modify_stage_fsm() returns const_cast<MyFSM*>(this)
12
13   //modifying code
14   //  uses 'fsm' which is non-const
15 }
```

"Depending on

While not 100% neat, it does the trick, and prevents from accidental writing to FSM state before modify_stage_fsm() is called (as compiler will notice modifying const *this* pointer, and will issue an error). Of course, one can call modify_stage_fsm() at the very beginning of the process_event() negating all the protection (or use one of several dozens another ways to bypass const-ness), but we're assuming that you do want to benefit from such a split, and will honestly avoid bypassing protection as long

as it is possible.

Note that depending on your game and FSM framework, post_message() function (the one which posts messages to other FSMs) may be implemented either as as a non-const function (then you'll need to call it only after modify_stage_fsm()), or as a const function (and then it can be called before modify_stage_fsm()). To achieve the latter, your FSM framework need to buffer all the messages which were intended to be sent via post_message() (NOT actually sending them), and to post them after the process_event() function successfully returns (silently dropping them in case of exception).

your game and FSM framework, post_message() function may be implemented either as as a non-const function (then you'll need to call it only after modify_stage_fsm or as a const function (and then it can be called before modify_stage_fsm

Now to the goodies coming out of such separation.

Exceptions before Modification Stage are Safe, including CPU exceptions

There are certain classes of bugs in your code which are very difficult to test, but which do occasionally happen. Some of them are leading to situations-which-should-never-happen (MYASSERTs throwing exception, see Chapter [[TODO]] for further discussion), or even to CPU exceptions (dereferencing NULL pointer and division-by-zero being all-time favourites).

If you're following the Validate-Calculate-Modify pattern, then all such exceptions (that is, if you can convert CPU exception into your-language-exception, see Chapter [[TODO]] for details for C++) become safe, in a sense that offending packet is merely thrown away, and your system is still in a valid state, ready to process the next incoming message. Yes, in extreme cases it may lead certain parts of your system to hang, but in practice most of the time the impact is very limited (it is much better to have a crazy client to hang, than your whole game world to hang, to terminate, or to end up in an inconsistent state).

This resilience to occasional exceptions has been observed to be THAT important in practice, that I think it alone is sufficient to jump through the hoops above, enforcing clean separation along Validate-Calculate-Modify lines.

Exception-based Determinism

One of the ways to achieve determinism which was mentioned in Chapter V with description postponed until later, is exception-based determinism.

Let's consider the following scenario: your FSM MIGHT need some non-determinstic data, but chances for it happening are fairly slim, and requesting it for each call to process_event() would be a waste. One example of such a thing is random data from physical RNG. Instead of resorting to "call interception" (which is not the cleanest method available, and also won't work well if your RNG source is slow or on a different machine), you MAY implement determinism via exceptions. It would work along the following lines:

- RNG_data becomes one of the parameters to process_event(), but is normally empty.
  - Alternatively, you MAY put it alongside with current_time to TLS, see Chapter V for details
- if, by any chance, you find out that you need RNG_data during your CALCULATE stage with RNG_data being empty – you throw a special exception NeedRNGData
  - as your VALIDATE and CALCULATE stages didn't change FSM state, there is nothing to rollback within the state
  - on-stack variable handling will be different for C++ and garbage-collected languages:
    - for C++, as long as you're always using RAII/std::unique_ptr<> for all on-stack resources (which you should for C++ anyway), all such objects will be rolled back automagically without any additional effort from your side
    - for garbage-collected languages, all on-stack objects will be cleaned by garbage collector

RAII

Resource Acquisition Is Initialization is a programming idiom used in several object-oriented languages, most prominently C++, but also D, Ada, Vala, and Rust.

— Wikipedia —

- on receiving such an exception, the framework outside of FSM will obtain RNG_data, and then will call MyFSM::process_event() once again, this time providing non-empty RNG_data
- this time, your code will go along exactly the same lines until you're trying to use RNG_data, but as you already have non-empty RNG_data, you will be able to proceed further this time.

Bingo! You have your determinism in a clean way, without "call interception" (and all because of clean separation between Validation-Calculation-Modification).

FSM Exception Summary

To summarize my main points about FSM and exceptions:

- Validate-Calculate-Modify is a pattern which simplifies life after deployment significantly (while it is not MUST-have, it is very-nice-to-have)
  - if you're following it, enforcing it is a Good Thing(tm)

- Following it will allow you to safely ignore quite a few things-you-forgot-about without crashing (don't overrely on it though, it is not a silver bullet)
- It also allows to achieve determinism without "call interception" via using exception-based Determinism in some practically important cases
- What are you waiting for? Do It! 😉

[[To Be Continued…

This concludes beta Chapter VI(d) 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 VI(d), "Modular Architecture: Server-Side. Programming Languages.]]

[–] References

[Facebook] Hans Fugal, "Futures for C++11 at Facebook"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG Server-Side. Eternal Linux-vs-Windows Debate*

MMOG Server-Side. Programming Languages »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: asynchronous, finite state machine, game, multi-player

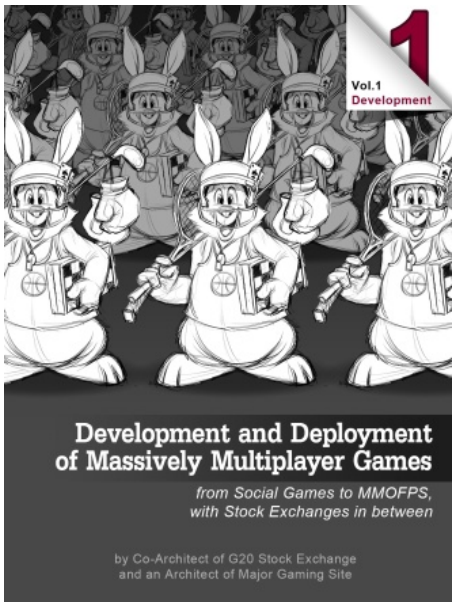
Copyright © 2014-2016 ITHare.com

MMOG Server-Side. Programming Languages

posted January 18, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter VI(e) 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.*

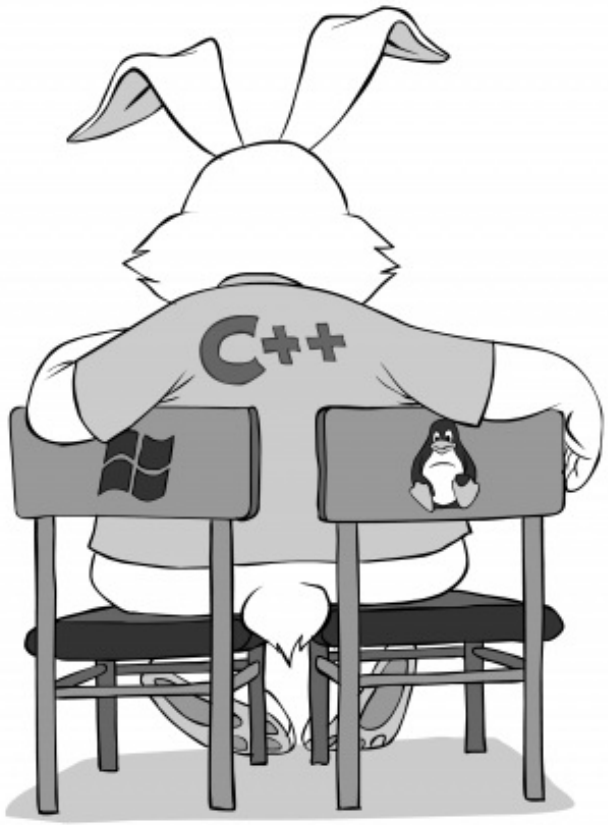
To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



Going Cross-Platform

In one of the previous sections we've discussed choosing a platform for your MMOG servers. However, one of the first things I've noted was that you should certainly consider developing cross-platform code. In fact, this is what I am usually doing (that is, if I can get past management, which is usually supported by a bunch of fellow developers who neither know, nor don't want to learn anything but their-favorite thing). But let's see what going cross-platform means from the programming languages point of view.

Cross-platform C++



Actually, my personal favorite for cross-platform development, is cross-platform C++. To those having any doubts: yes, C++ can be made cross-platform, I've done it myself on numerous occasions. It works even better when you have your code restricted to event-driven side-effect-free processing (a.k.a. deterministic finite-state-machines (FSMs), see Chapter V for details). For our current discussion, one thing is important about FSMs: as soon as your FSM becomes deterministic, it doesn't really have any significant interaction with the system, so it is "pure logic" (a.k.a. "moving bits around", and is pretty much like "pure" functions from functional programming). And "pure logic" is inherently cross-platform (that is, as long as you keep it "pure").

On the other hand, to keep your logic "pure", you'll need to make quite significant effort, and to be extremely vigilant when it comes to platform-specific dependencies (see also relevant discussion in Chapter V). This is especially true for C++.

Note that for some out-of-FSM pieces of code, you MAY want to use platform-specific stuff as an optimization. Usually, it works as follows (*yes, I know it is really old news for all the seasoned C++ cross-platform developers, but believe me or not, there are lots of C++ developers out there who don't know it, especially hardcore zealots of Windows-specific development)*:

"Note that for some out-of-FSM pieces of

- You develop a perfectly cross-platform version, which uses only cross-platform APIs. It doesn't really matter

whether cross-platform API is a part of official C++ standard, more important question is whether it is really implemented across the board. In practice, there are several big sets of APIs which we can safely consider cross-platform:

code, you MAY want to use platform-specific stuff as an optimization.

- C++11 standard (C++14 is still only partially supported across the board), including std:: library
- Most of C Standard Library (see discussion on it's limitations in Chapter [[TODO]])
- boost:: library
- Berkeley sockets (while it is not strictly 100% cross-platform, for practical purposes it is very close)
- Note that POSIX standard stuff (the one which is not a part of C library) is generally NOT cross-platform. Notable example: fork() which is missing under Windows
    - Moreover, some Windows functions which look like their POSIX counterparts and have the same signatures, exhibit different behavior. One notable example includes Microsoft _exec*() family of functions, which has very different semantics from POSIX exec*() .

- You launch it, and it works for a while
- Then, you realize that performance of your cross-platform code can be improved for one specific platform. Just as one example – your cross-platform version implemented inter-thread queues-with-select() (see Chapter V for the rationale behind these queues, which are waiting either for somebody pushing something into the queue, or for data arriving to one of the sockets) via sockets+anonymous-pipe, and you realized that under Windows WaitForMultipleObjects()-based version will work faster.
    - Ok, you're rewriting relevant piece of code (keeping all the external interfaces of this piece intact), and placing it under an ugly (but still working perfectly fine) #ifdef MY_DEFINE_WINDOWS_ONLY (and relevant portion of the cross-platform code under #ifndef MY_DEFINE_WINDOWS_ONLY). Bingo! You have your Windows-specific version running under Windows, and your cross-platform version running everywhere else.

Bottom line: C++ can be made cross-platform. For further details, see Chapter [[TODO]].

Cross-platform Languages

…the purpose of Newspeak was not only to provide a medium of expression for the world-view and mental habits proper to the devotees of IngSoc, but to make all other modes of thought impossible.

Another way to achieve cross-platform code is to use one of the cross-platform languages, such as Java, Python, C#, or Erlang.

From cross-platform point of view, these languages have one significant advantage over cross-platform C++: *most* of their APIs are already cross-platform, so they don't provide you *that much* opportunities to deviate into platform-specific stuff. While going platform-specific is still possible (via JNI/Python ctypes/PInvoke or unmanaged code/...), it is usually more difficult with cross-platform languages.

This "going platform-specific being more difficult" is actually the main advantage of cross-platform languages when going cross-platform

In other words, the problem with C/C++ is that they're providing you more freedom with going platform-specific (and yes, having more freedom is not always a good thing). The way cross-platform languages are doing it, can be seen as an (almost) enforcement of a self-imposed rule that "everything should be cross-platform".

Now let's consider these languages against our "baseline" cross-platform C++.

Pros (compared to C++)

"Almost all cross-platform programming languages I know are garbage-collected

- Almost all cross-platform programming languages I know[1] are garbage-collected.
  - It means less time spent on memory management during development, which in turn means faster time-to-market. On the other hand, I will argue that for an FSM model (especially in gaming context, where memory allocations are often discouraged as too expensive), memory management is rudimentary either way, so the difference will be negligible (that is, provided that you have at least one seasoned C++ developer who knows how the things should be done at lower levels, and provided that you are using std::unique_ptr<>).
  - It means no pointers, and no bugs related to misuse of pointers (and, Ritchie save us, pointer arithmetic). Note that once again, we're in the realm of having too much freedom causing trouble (and once again, it is only a question of self-discipline to avoid using them, as references do just fine 90% of the time, and reference-like use of pointers will fill the rest).
- As noted above, keeping your code cross-platform requires much less efforts

in Java/Python/… than in C++.

- Learning curve. C++ learning curve is steep. It is not too bad if you're staying within limits of the FSM, but reading a book on C++ can easily be overwhelming (especially books which start with discussing interesting-but-not-really-important-and-rarely-used-things such as "how to overload operators" and multiple inheritance).
- Good C++ developers are few and far between, not to mention they're very expensive. For most of the languages above (except for Erlang) finding a good developer is usually significantly easier.

Cons (compared to C++)

When speaking about deficiencies of the cross-platform programming languages, several things come to mind (note that while the list of cons is longer than that of pros, it doesn't mean that cross-platform languages are inherently worse; it is just that these cons are not as well-known as cons, so I'm spending more time elaborating on them):

"Almost all cross-platform programming languages I know are garbage-collected

- Almost all cross-platform programming languages I know[1] are garbage-collected. This means that they tend to suffer from two problems:
  - the first problem is memory bloat (if you have any doubts that such a problem exists – take a look at Eclipse or at OpenHAB). I tend to attribute this apparent bloat to the following. While garbage-collected languages eliminate so-called "syntactic memory leaks" (pieces of memory which *cannot possibly* be used), they cannot possibly eliminate "semantic memory leaks" (pieces of memory which *can* be used, but won't be used, ever) [NoBugs2012]. And those "semantic memory leaks" for garbage-collected languages tend to be worse than for manually memory managed languages such as C++, because of "we don't need to care about memory leaks" mentality, and because garbage collectors are obligated to stay on the absolutely safest side, keeping in memory everything that has a slightest chance to be used (i.e. everything *reachable*). Of course, memory bloat for garbage-collecting languages can be managed (there is nothing difficult in explicitly assigning null to a reference); however, whether after doing it they will still provide that much speedup in development time over C++ – is not obvious to me.
    - On the other hand, it should be noted that for FSM-based development (which usually implies states of rather limited size), the problem of "semantic memory leaks" is usually not too bad (based on the same reasoning why manual

Semantic Garbage

Semantic garbage cannot

memory management is usually not that much of a problem for FSM-based development), and fixing them isn't *too* difficult.

> **be automatically collected in general, and thus cause memory leaks even in garbage-collected languages.**
>
> — *Wikipedia* —

- The second problem is garbage collector's infamous "stop the world" (mis)feature. In short – to perform garbage collection, most of GCs out there need to "stop the world" (i.e. to stop *all the threads(!)* within the same VM) for some time. For most of the applications, it is not a problem (as delays even of a hundred milliseconds are so short that your application won't really notice them). However, if we're speaking about a fast-paced game such as an MMOFPS, these delays are known to cause lots of trouble. Even worse, when you run into such things, it is usually too late to rewrite your whole code, which leads to really ugly workarounds such as "let's not run garbage collector at all for a while" (then, if your game event, such as match, is long enough, you can easily eat all the server RAM and even more). While it doesn't mean that GC languages cannot possibly work with MMOFPS, I'd suggest to be very cautious in this regard, and to research how big "stop-the-world" pauses are for the GC used by your target VM (also note that it is about VM, and not about language, so, say, the same C# may exhibit very different behaviour under CLR and Mono).
    - As a mitigating measure, it is possible to reduce the time of "stopping the world" effect (at the cost of some performance loss); see, for example, "Concurrent Mark-and-Sweep" and "G1" garbage collectors for JVM, and <gcConcurrent>/SustainedLowLatency for CLR; they run a large portion of GC processing without "stopping the world" (so only a small part of GC loop needs to be run in the "stop the world" mode). From what I know, these GCs (at the cost of minor overall performance penalty) bring pauses down to single-ms range even for large heaps, which makes it "good enough" even for MMOFPS; as usual, YMMV, batteries not included. For Mono, there is a supposedly similar GC flag *concurrent-sweep*, though I have no information how small the "stop-the-world" pauses are when Mono GC runs with this flag.
    - As another mitigation technique (which, at least in theory, may also work as a compliment to concurrent collectors), it is possible to reduce "stop the world" time by splitting your system into separate VMs (such as JVM or CLR VM[2]) and each VM will run a separate GC. This tends to help because the smaller your "world" is, the less time garbage collector will need to run, so the less time "stop the world" will take. The technique actually flies extremely good with FSMs (as FSMs, at least our FSMs and Erlang/Akka Actors, are share-nothing, they can be easily put into separate VMs). In the extreme case, you may even end up with running one VM for each of "game world"

FSMs. There is a price of it, however, and it is related to the overhead brought by each of VMs; where the optimum for your game (balancing overhead vs latencies) – you'll need to find out yourself.

- The third GC-related problem is related to asynchronous I/O (in our context – socket I/O). Intensive server-side asynchronous I/O tends to cause problems with GC at least under CLR, as to pass the buffer to an asynchronous Win32 API, it needs to be "pinned" (i.e. cannot be relocated, what reflects pretty badly on CLR's copying GC), and having too many pinned buffers may cause CLR's GC to stall, up to the point of being deadlocked. While there is a workaround for it, via SocketAsyncEventArgs (or you can always go into an unmanaged mode, accessing Win32 APIs directly and losing being cross-platform pretty much as we've discussed it for C++ in [[TODO!]] section above), this is a complication one needs to be aware about in a highly-loaded network-oriented environments. Also I have no idea whether the workaround would work as intended under Mono.

JIT

Just-In-Time (JIT) compilation, also known as dynamic translation, is compilation done during execution of a program – at run time – rather than prior to execution

— Wikipedia —

- Unless your target platform has a JIT compiler for bytecode of your language, you're most likely looking at 10x+ performance penalty
  - Fortunately, all the languages mentioned above do have JIT, with only one unfortunate exception (leaving discussion about Lua/LuaJIT aside until "Scripting Languages" section). Erlang, while working on BEAMJIT, still seems to have it only as a proof-of-concept 🙁 .[3]
- Even when compared with JIT-enabled cross-platform language, C++ performance can be made at least *somewhat* better 99% of the time. On the other hand, 95% of the time you won't bother with such optimizations. Possible exceptions include heavy AI and/or heavy physics simulations (especially if they go well with SSE).

SSE

Streaming SIMD Extensions (SSE) is an SIMD (Single Instruction Multiple Data) instruction set extension to the x86 architecture

— Wikipedia —

---

[1] Rust being the only exception

[2] I know that Microsoft prefers to call it "Execution Engine", but it still looks like a VM, swims like a VM, and even quacks like a VM

[3] As for Python, while CPython as such doesn't have JIT, other Python implementations, such as native PyPy and JVM-based Jython, do have JITs.

Personal Preferences and FSMs

Out of the aforementioned cross-platform programming languages, I am especially fond of Erlang's actors (and it also reportedly has a good record for development of large-scale distributed systems, though an overhead due to apparent lack of JIT is significant). Java and Python are not bad either (within their own applicability limits). I have never been a big fan of C#, in particular because it traditionally has the blurriest line between cross-platform APIs and platform-specific stuff (which is not really surprising as such policy makes perfect business sense for Microsoft), but if you're planning your servers as Windows-only – it will certainly do, and if you're going to go Linux – Mono MIGHT work for your too (though in the latter case it is not that obvious).

On the other hand, I need to note that with some self-discipline, FSMs described in Chapter V (and which are strictly equivalent to Erlang's actors/processes), can be easily implemented in *any* of these languages (and in C++ too).

Scripting Languages

We went through C++ and cross-platform languages, but we're not done yet.

"On the server side (unlike client-side) protection from bot writers is not an issue (as server-side code is never exposed to players)

As it was mentioned in Chapter V, for game development, there is a common practice to use scripting languages for game logic. People writing in scripting languages include, but are not limited to, are game designers. Moreover, on the server side (unlike client-side) protection from bot writers is not an issue (as server-side code is never exposed to players), so it means that scripting languages become more feasible for the server-side. Therefore, it seems to make perfect sense to allow using some kind of scripting language on the server too.

Two most common scripting languages, used in games, are Lua and JavaScript. I won't go into comparison of these two languages, but will just note that both will do their job when it comes to game scripting. Just one thing worth mentioning in this regard is that there is an internal conflict within Lua (between main Lua team and Mike Pall/LuaJIT, who actively dislikes changes in Lua 5.3, so LuaJIT doesn't seem likely to support Lua 5.3+, ever(!)); this kind of internal conflicts can be really devastating for the language in the medium- and long-run, which makes it an argument against Lua ☹ .

The most common concern about allowing scripting on the server side is related to performance. However, with LuaJIT (with limitations mentioned above, and I don't really like them) and V8 JavaScript (which also has it's own JIT), this is much less of a concern than for non-JIT-ted script engines.

On Languages as Such

"From the 50'000-feet point of view, 90% of the differences between modern mainstream programming languages (as they're normally used - or better to say, SHOULD be used - at application-level) are minor or superficial.

I know that I will be hit hard (once again) for not going into a lengthy discussion about pros and cons of different programming languages (those which I mentioned above and those which I failed to mention). However, my strong position is that from the 50'000-feet point of view, 90% of the differences between modern mainstream programming languages (as they're normally used – or better to say, SHOULD be used – at application-level) are minor or superficial.[4] This is also confirmed by Line-to-Line conversion exercise discussed in "Line-to-Line conversions: '1.5 code bases'" section below.

Another observation which helps in this regard, is that there is a tendency for modern programming languages to converge as the time goes. For example, C++11 code is much closer to Python code than C++03, and Java 5+ (with generics) is much closer to C++ than Java 4- (the one without generics). Programming languages borrow certain constructs and practices (usually best ones, but it is not guaranteed) from each other, bringing them closer as the time goes.

Still, there are two things which tend to be quite different between the programming languages. The first and more obvious one, is, of course, the difference between manual and automated memory management. Still, with more-or-less modern C++ (with widespread use of containers and std::unique_ptr<>), the difference is not that drastic.

The second thing which MIGHT be quite different between the languages, is related to support for lambdas (which, as we've discussed in "Take 3. Lambda Continuations to the Rescue! Callback Pyramid" and "Take 4. Futures" subsections above, is important). For example, lambdas in Python [StackOverflow.PythonLambdaLoop] and C# [StackOverflow.C#LambdaLoop] have rather strange peculiarities with regards to lambdas within loop (or maybe it's C++ peculiarity that it behaves as intuitively expected?). However, in most cases, some strict equivalent between the languages still exists.

One further word of caution is related to co-routines. While co-routines/fibers do simplify development (this stands to some extent even when we compare them to futures), they have significant practical drawbacks related to lack of "stack snapshot" (which is necessary to implement quite a few FSM goodies, including realistic production post-mortem, see Chapter V for details [[TODO! add section on coroutines and "stack snapshot" to Chapter V]]). Also, their support is still much less universal than that of lambdas.[5]

[4] it doesn't really stand for Erlang, and I am not 100% sure whether it stands for Lua, as I don't have practical experience with it, but C++/C#/Java/Python/Javascript as-you-use-them-for-application-level-programming are all pretty much the same, saving for relatively limited amount of oddities and peculiarities

[5] For C++, you can use fibers, but IIRC Java as such doesn't support them, and Python 2 which is still used quite a lot, doesn't have coroutines

Which Language is the Best? Or On Horses for Courses

horses for courses

An allusion to the fact that a racehorse performs best on a racecourse to which it is specifically suited.

— Wiktionary —

Right above, we've described quite a few options for server-side programming languages. The Big Question is, as usual, the following: which one to choose?

My two ~~cents~~ points in this regard are the following. First, there is no such thing as "the best language for everything". What we need is a language-best-for-some-specific-task. And here there are quite a few different scenarios, from "just a scripting language for game designers to work with" (where C++ and even Java are pretty much out of question), to "time-critical simulation code", with "something for integration with enterprise web apps" in between. As a very wild guess, you might want to use Lua or JavaScript for the first one, C++ for the second one, and Java/C# for the third one (been there, seen that). Doing everything in one single language, while possible, in many cases will be suboptimal.

My second point in this regard is that with FSMs, it is easy to combine FSMs written in different languages, in any way you want. Personally, I've made such things myself for three languages: C++, Java, and JavaScript. It went along the following lines:

- Originally, the whole thing (both outside-FSMs infrastructure code and intra-FSM code) was written in C++. Great performance, full control, no problems with GC, everybody was really happy, etc. etc. But finding good C++ developers isn't easy 🙁 .
- As a result, at some point, it was decided to make an analytics portal and to develop it in Java.
- As pure DB access wasn't sufficient (as they needed real-time updates, and DB triggers didn't look optimal at all) Java guys asked for a way to get the data from C++ system.

- Ok, here went a line-to-line translation project of outside-FSM infrastructure code into Java (to facilitate writing FSMs in Java), see "Line-to-Line Translations: "1.5 code bases"" section below for further details[6]
- This outside-FSM infrastructure Java code was compatible at message format level with C++ code, which means that from C++ FSM standpoint, Java-based FSM was indistinguishable from a C++-based one, and vice versa.
- So, C++ and Java FSMs could interact easily (after agreeing on interfaces, for more details see Chapter [[TODO]]), without no problems whatsoever. In particular, Java FSMs were able to "subscribe" to the data "published" by C++ FSMs, and get all the updates in real-time (most of the data necessary was already published by C++ FSMs, so Java FSM subscribing to the data they needed, was mostly possible without changing C++ code).

"Here went a line-to-line translation project of outside-FSM infrastructure code into Java (to facilitate writing FSMs in Java)

In a different project (and similar situation), a JavaScript FSM was produced to allow server-side scripting (in addition to existing C++ FSMs). In this case, C++ outside-of-FSM code was re-used, which called process_event() (written in JavaScript) from within. The same approach can be (more or less easily) extended to all the other programming languages of interest, see "Supporting Different Environments" section below for further discussion.

In any case, all the paradigms of our FSMs were transparently maintained for all the FSMs across all the supported languages. This included more or less the following things:

- process_event() as a single access point to our FSM, see Chapter V
  - current_time was passed to process_event() either as an explicit parameter, or via TLS and current_time() call (see Chapter V for details)
- timer actions (in those projects, it was timer messages, but now I suggest same-thread futures instead, see "Take 4. Futures" subsection above)
- communication interfaces (see Chapter [[TODO]]), including:
  - support for non-blocking RPCs (it was OO-style same-thread callbacks, but now I suggest same-thread futures instead, see "Take 4. Futures" subsection above)
  - support for state synchronization interfaces (see Chapter [[TODO]])) with same-thread callbacks
- all the recording/replay goodies described in Chapter V

---

[6] we could try to go JNI route instead, but we preferred pure Java and didn't regret

this decision

Supporting ANY language/compiler/JIT: Is It Worth the Trouble?

The next obvious question on this way is the following:

Are such cross-language things worth the trouble of implementing them?

Well, of course, YMMV, but from my experience the answer is

Absolutely!

In such an FSM-based multi-language development paradigm you're no longer tied to one programming language. You may say "hey, this is what CLI/Mono (as well as non-Java compilers into JVM bytecode) are about!" Right, but with CLI/CLR you're still tied to one type of VM (ok, two if we're Windows-only).

And with FSM-based cross-language approach, we're not restricted to one single VM, or to the availability of specific compilers which compile into that single VM. With cross-language FSMs we can use the very best language/compiler pair for each specific job – whether it is Lua/LuaJIT, or JavaScript/V8, or Python/PyPy, or Java/HotSpotVM, or C++/LLVM (note that none of these popular and very-well performing combinations is possible under CLI/CLR).

"With cross-language FSMs we can use the very best language/compiler pair for each specific job – whether it is Lua/LuaJIT, or JavaScript/V8, or Python/PyPy, or Java/HotSpotVM, or C++/LLVM

I rest my case.

Supporting Different Environments

The next question (assuming that I've managed to sell you the idea of using cross-language FSMs) is "how to implement them?"

From my experience, there are two possible approaches. The first one is to have a C++ outside-of-FSM code, and then to integrate C++ into each of the engines you need. Usually, it is not that difficult. For example, you can have C++ threaded communication code running under JNI and calling your Java MyFSM.process_event() fram there. Or under CLI, it is possible to have unmanaged code doing pretty much the same thing. Or with LuaJIT/V8, it is easy to have C++ app calling appropriate script engine.

The second approach is related to line-to-line translations.

Line-to-Line Translations: "1.5 code bases"

Originally, I've written a nice 1500-word piece about line-to-line translations, but then realized that it doesn't really warrant that many words in the context of this book. Still, as it was promised in Chapter V, here comes a brief overview of this not-that-well-known technique.

Let's assume that you already have a working piece of code in C++, and want to port in into Java (it will work pretty much the same for other languages too, like C++-to-ActionScript for the client side, but let's use C++-to-Java for the purposes of our example).

The aim of line-to-line conversion is not only to port the code, but also to keep roughly 1-to-1 correspondence between the original code and the translated code; as we will discuss below, this correspondence is very important for further maintenance of the port (and code maintenance is the thing which haunts all the multiple code bases in the real world).

The idea behind is the following. When you have sufficiently straightforward and platform-independent code in any modern OO programming language, the essence of the code can be translated to a different OO programming language in a very straightforward manner. For example, if your C++ code is implementing Dijkstra's pathfinding algorithm as follows, [7]:

```
pair<map<const Vertex*,int>, map<const Vertex*,const Vertex*>>
   dijkstra(const Graph& g, const Vertex* source) {
  set<const Vertex*> Q;
  map<const Vertex*,int> dist;
  map<const Vertex*,const Vertex*> prev;

  for(const Vertex* v: g.vertexes()) {
    dist[v] = INT_MAX;
    prev[v] = NULL;
    Q.insert(v);
    }

  dist[ source ] = 0;

  while(Q.size()>0) {
    //find u from Q with minimum dist[u]
    const Vertex* u = NULL;
    int distU = INT_MAX;
    for(const Vertex* it : Q) {
      int distIt = dist[it];
        if(distIt < distU) {
        u = it;
        distU = distIt;
      }
    }
    //u found
    assert(u!=NULL);

    Q.erase(u);

    for(const Vertex* v : g.neighborsOf(u)) {
      int alt = dist[u] + g.length(u,v);
      if(alt < dist[v]) {
        dist[v] = alt;
        prev[v] = u;
      }
    }
  }
  return pair<map<const Vertex*,int>,map<const Vertex*,const Vertex*>>(dist,prev);
}
```

...then, when you need to rewrite this code into, for example, Java, you can simply take your (supposedly working) C++ code, and to write its Java equivalent along the following lines:

```
1  public class Dijkstra {
2    public static Pair<TreeMap<Vertex,Integer>,TreeMap<Vertex,Vertex>>
3       dijkstra(Graph g, Vertex source) {
4     TreeSet<Vertex> Q = new TreeSet<Vertex>();
5     TreeMap<Vertex,Integer> dist = new TreeMap<Vertex,Integer>();
6     TreeMap<Vertex,Vertex> prev = new TreeMap<Vertex,Vertex>();
7
8     for(Vertex v: g.vertexes())  {
9        dist.put(v, new Integer(Integer.MAX_VALUE));
10       prev.put(v, null);
11       Q.add(v);
12    }
13
14    dist.put(source, new Integer(0));
15
16    while(Q.size()>0) {
17      //find u from Q with minimum dist[i]
18      Vertex u = null;
19      int distU = Integer.MAX_VALUE;
20      for(Vertex it: Q) {
21        int distIt = dist.get(it).intValue();
22        if(distIt<distU) {
23           u = it;
24           distU = distIt;
25        }
26      }
27      //u found
28      assert u!=null;//be careful to keep your Java asserts
29                  //  consistent with your C++ asserts;
30                  //see Chapter [[TODO]] for further discussion
31                  //  on asserts in C++
32
33      Q.remove(u);
34
35      for(Vertex v:g.neighborsOf(u)) {
36        int alt = dist.get(u).intValue() + g.length(u,v);
37        if(alt < dist.get(v).intValue()) {
38          dist.put(v,new Integer(alt));
39          prev.put(v,u);
40        }
41      }
42    }
43    return new Pair<TreeMap<Vertex,Integer>,TreeMap<Vertex,Vertex>>(dist,prev);
44  }
45 }
```

[[TODO: place C++ and Java side by side in a book]]

As you can see, ported Java code visually looks very similar to C++ original; moreover, we can easily see the one-to-one correspondence between the lines of C++ code and Java code. When we have such C++ and Java code, we don't really have two separate code bases, as they're too closely related to name them separate. I

prefer to name such "C++ and some-other-language" pairs as "1.5 code bases" (at least it is clearly more than 1 code base, and certainly less than 2).

In practice, it means that for really platform-independent C++ code, in most cases we can produce an equivalent code in a different programming language, but with the same semantics and (hopefully ;-)) producing exactly the same result. Moreover,

it is usually possible to produce an equivalent code which has one-to-one line-to-line correspondence with the original.

This last observation is extremely important in practice, for the purposes of code maintenance. As it is pretty well-known in the industry,[8] having two code bases very frequently leads to major problems because of code maintenance issues. In other words, after having changed one code base, it is often a problem to make a strictly equivalent change in another code base. However, with line-to-line conversion and "1.5 code bases", this maintenance process (while still not being a picnic!) becomes significantly simplified: after we've had our code bases equivalent, and we've made a single change in our first code base, then making an equivalent change becomes a breeze: just look at source control differences for the first code base, and apply an equivalent thing to the second code base. It is important to note that

in this apply-changes-to-second-code-base process, there is usually no need to understand the essence of the change which was made; most of the time, the change can be applied based only on general understanding of inter-language equivalence rules[9]

Here in original 1500-word piece there were examples of modifying the C++ code – getting differences from the source control – applying those differences to Java, but within the scope of this book, it probably would be an overkill, so I've eventually decided to skip it. [[TODO!: write a separate article about it with more examples]]

---

[7] based on [Wiki.Dijkstra]

[8] and I've seen several times myself as competitors successfully started second client with a separate code base, and then second client started to lag behind in development, up to the point of being unplayable, with subsequent abandoning of the second client

[9] in fact, I'm pretty sure that, given restricted dialect of C++, which I am normally using for platform-independent code, it is perfectly possible to build a C++-to-Java (or C++-to-ActionScript) compiler *at source level*; however, parsing C++ (which is not a LALR(1) grammar) is difficult, so I've never had enough time to undertake such an

endeavor

Line-to-Line Translations: Are They Practical?

The code above is an interesting exercise, but of course, it is merely an example, so you may still have questions whether this exercise scales well to larger-scale pieces of code.

As noted above, I was personally involved in an exercise of porting C++ into Java in a Line-to-Line manner. Porting 20'000 lines of code took 2 weeks for the first 80%, and two months for the remaining 20%, and worked happily ever after (the code was changed since the port, but quite rarely). I don't know how it would scale to a million of lines of code, or to a code which is changed twice a day, or to a code which is not as straightforward. Still, if you're out of other options, line-to-line source translations may happen to work for you.

> **"Porting 20'000 lines of code took 2 weeks for the first 80%, and two months for the remaining 20%, and worked happily ever after**

Also note that performance-wise the converted code might be not top-notch one (while concepts and ideas are generally very similar between the languages, subtle performance-related details don't). In other words, with good conversion if your algorithms were O(N) they generally should stay O(N), but you may easily face 20% performance hit (potentially more in extreme and fringe cases) compared to the best possible code in target language.

One further thing to keep in mind in this regard, is that porting from C++ to Java (C#/...) is generally simpler than the other way around. In particular, this is because while removing manual memory management is trivial, adding it can be quite difficult and MAY require intimate knowledge of the program internals (which goes against the idea of purely mechanistic conversion).

Inter-Language Equivalence Testing: FSM Replay Benefits

In quite a few cases, you may need to port a part of your code from one language to another one. It may happen, for example, to optimize the time-critical FSM, or to have "1.5 code bases" in a line-to-line conversion manner as described right above. And with all such conversions, one of the biggest problems is the question "how we can be sure that the code-in-new-language and the code-in-old-language are strictly equivalent?"

Fortunately, for FSMs there is an easy way to test the code equivalence. The procedure goes as follows:

- "record" a big chunk of inputs and outputs for FSM-being-ported (and running old code); "recording" can be done along the lines described in

Chapter V, and may be done even in production.

- "replay" it in lab on the new code. (as described in Chapter V)

- if the results are exactly the same for old code and new code, on a sufficiently large chunk of real-world data, it means very good chances that the code is indeed equivalent (at least within the bounds which are of practical interest). In practice, it has been noticed that for quite a big site, if there is no bug after the first four hours after new code deployment, there won't be any bug in "core logic" at all. Pretty much the same applies to record/replay testing.

    - if there is a non-equivalence, it can be found very quickly by simply running the same "replay" over both languages in debugger line by line, and comparing corresponding variables.

On Code Generators and YACC/Lex (or Bison/Flex)

"As soon as you've got lots of boilerplate code - you MIGHT be able to generate it with your own code generator

One thing which doesn't strictly belong to server-side, but which I need to mention somewhere, is YACC/Lex. As we'll see later, there are quite a few cases where having your own source-code-generator is beneficial. Two most obvious examples include Interface Definition Language (a.k.a. IDL, discussed in Chapter [[TODO]]), and prepared-statements-code-generator (discussed in Chapter [[TODO]]). Other game-specific things (usually not really comping code, but dealing with some declarative statements and converting them to code) might also be helpful (in general, as soon as you've got lots of boilerplate code – you MIGHT be able to generate it with your own code generator). In the (rather extreme) case you may even be able to write your own code generator to support co-routines-with-stack-snaphot.

While these compilers not strictly required (and you're able to write all the code you need by hand), they will speed up your development a lot. Just as one example: if you need a thousand of those prepared statements, writing them by hand in C++ (or Java/pick-your-poison), while possible, is very tedious and error-prone. The same goes for any kind of marshalling/IDL.

Whenever you have your code generator, it always works as follows:

- You have a file in your own language (usually more of declarative nature than of imperative nature).

- Then, you run your code generator over it, obtaining kinda-source code in your programming language.
  - **This generated kinda-source code MUST NEVER EVER be modified manually.** Instead, either source-code-in-your-own-language needs to be modified, or your code generator.
- Then, you compile kinda-source with your usual language compiler (or interpret it, whatever)

In most cases, the best way to implement such compilers is via YACC/Lex (or Bison/Flex, which is pretty much the same thing). As for these generators performance is not important, alternatively to classical YACC/Lex/Bison/Flex you may want to look at [PLY] (Python Yacc/Lex). The idea of all of them is pretty much the same:

"In most cases, the best way to implement such compilers is via YACC/Lex (or Bison/Flex, which is pretty much the same thing).

- you're writing "language grammar" (in .y file, which is more or less reminiscent of BNF forms)
- you're specifying what exactly you want to do with it as you're parsing (within the same .y file)
- you're compiling this .y file and obtaining your C (or Python, in case of PLY) code
- you're running this compiled code over your source file, and (if you've done everything right) are obtaining an abstract syntax tree (AST)
  - as you've got your AST, you can generate any code you need, out of it

For further information on C-language YACC, please refer to the classical tutorial [Niemann]. One further trick I'm using a lot for such code generators, is the following:

- define YYSTYPE as a C++ class (which will be essentially your "AST node"); usually YYSTYPE is int, but nothing prevents you from re-defining it
- define member functions for your YYSTYPE so that you add other AST nodes into current one. Don't be afraid to make deep copies here – you won't notice performance differences anyway.
- use code such as { $$.add($1,$2); } within your .y file (see tutorial mentioned about on the meaning of these magical $$ and $1/$2).
- when the whole hierarchy is processed, you'll get your whole AST at the (logically) topmost rule of your .y file.

Due to lots of copies, this approach is damn inefficient compared to traditional compilers,[10] but for vast majority of our gaming purposes parser performance

won't matter, saving you quite a bit of development time (and as it is run only on your development/build machines, it won't affect performance of your runtime code at all).

---

[10] and was also reported to fail under some YACC implementations when trying to compile hundreds of thousand of lines, but this problem is solvable

[[To Be Continued…

This concludes beta Chapter VI(e) 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 VII, "Modular Architecture: Protocols."]]

[–] References

[NoBugs2012] 'No Bugs' Hare, "Memory Leaks and Memory Leaks"
[StackOverflow.C#LambdaLoop] "Captured variable in a loop in C#", StackOverflow
[StackOverflow.PythonLambdaLoop] "What do (lambda) function closures capture in Python?", StackOverflow
[Wiki.Dijkstra] "Dijkstra's algorithm", Wikipedia
[PLY] David Beazley, http://www.dabeaz.com/ply/
[Niemann] Tom Niemann, "Lex & Yacc Tutorial"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Asynchronous Processing for Finite State Machines/Actors:…*

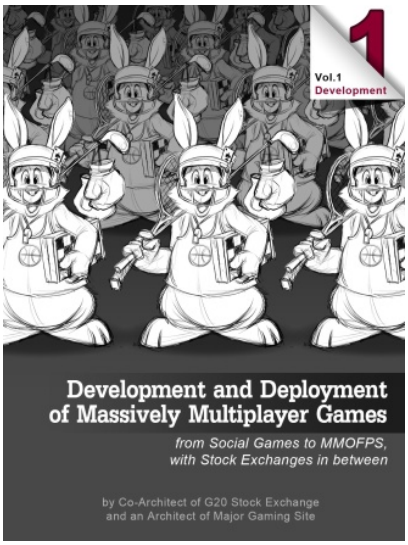
MMOG. RTT, Input Lag, and How to Mitigate Them »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: game, multi-player, programming language, server

Copyright © 2014-2016 ITHare.com

MMOG. RTT, Input Lag, and How to Mitigate Them

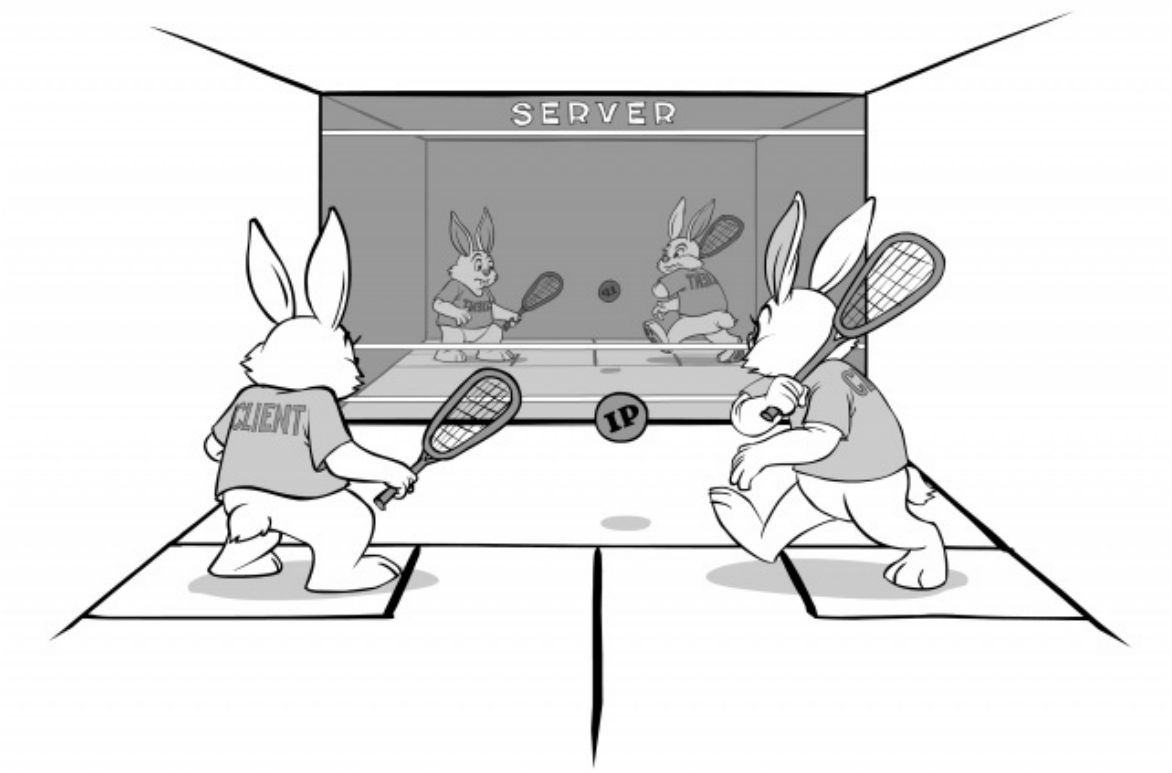
posted January 25, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko



[[*This is Chapter VII(a) 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.]]

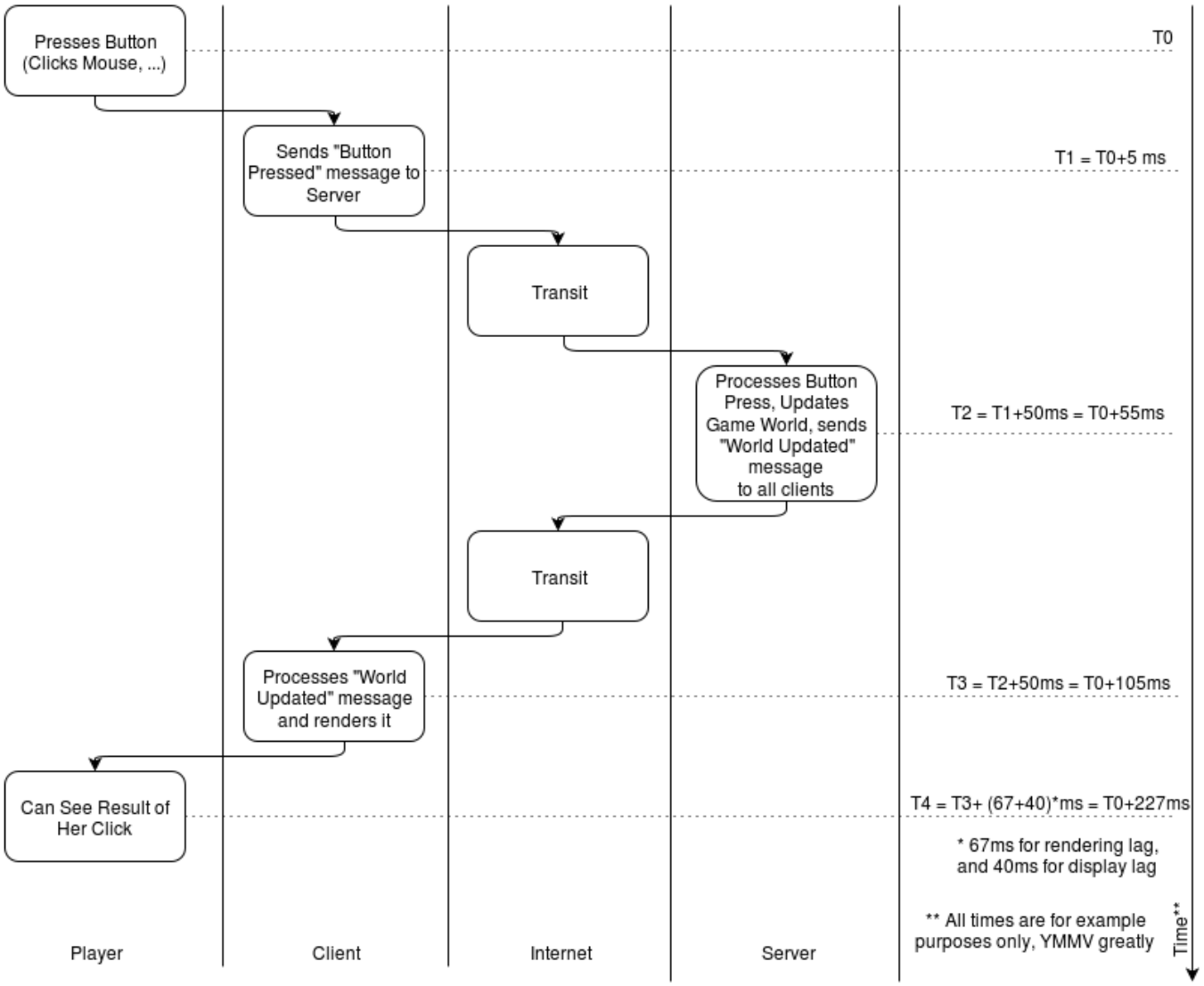
Now we're ready to discuss what the MMOs are all about – protocols. However, don't expect me to discuss much of the lava-hot "UDP vs TCP" question here – we're not there yet (most of this question, alongside with the ways to mitigate their respective issues, will be discussed in detail in Chapter [[TODO]]). For now we need to understand the principles behind the MMO operation; mapping them to specific technologies is a related but different story.



Data Flow Diagram, Take 1

Note that if your game is fast-paced (think MMOFPS or MMORPG), the approach described with regards to Take 1 Diagram, won't allow you to produce a game which doesn't feel "sluggish" (it will work, but won't feel responsive). However, please keep reading, as we will discuss the problems with this simple diagram, and ways to deal with them, later.

To see what the MMOG protocols are about, let's first draw a very simple data flow diagram for a typical not-so-fast MMO. As it was discussed in Chapter III, our server needs to be authoritative. As a result, the usual flow in a simplistic case will look more or less as follows:



NOT REALLY PRACTICAL FOR FAST-PACED GAMES: SEE FIG VII.2 FOR NECESSARY IMPROVEMENTS

Fig VII.1

Despite visual simplicity of this diagram, there are still a few things to be mentioned:

- All the specific delay numbers on the right side are for example purposes only. Your Mileage May Vary, and it may vary greatly. Still, the numbers do represent a rather typical case.
- It may seem that the client here is pretty "dumb". And yes, it is; most of the logic in this picture (except for rendering) resides on the server side. However, in most of the games there are some user actions which cause client-only changes (and don't cause any changes to the server-side game world), they can and should be kept to the client. These are mostly UI things (like "show/hide HUD", and usually things such as "look up"), but for certain games this logic can become rather elaborated. Oh, and don't forget stuff such as purchases etc.: if you keep them in-game (see Chapter [[TODO]] for details), it will require quite a lot of dialogs with an associated client-side logic, and these (select an item, info, etc. etc.) are also purely client-side until the player decides to go ahead with the purchase.

- Last but certainly not least: for fast-paced games, there is one big problem with the flow shown on this diagram, and the name of the problem is "latency". It is obvious that for this simplistic data flow, the delay between player pressing a button, and her seeing results of herself pressing the button (which is known as "input lag"), will be at least so-called round-trip-time (RTT) between client and server (which is shown as 100ms for Fig VII.1, see "RTT" section below for more discussion regarding RTT). In practice, though, there is quite a bit added to the RTT, and for our example on Fig VII.1, 100ms RTT resulted in 227 overall delay. And if this delay (known as "input lag", which is admittedly a misnomer) exceeds typical human expectations, the game starts to feel "laggy", all the way down to "outright unplayable" :-(. Let's take a closer look at these all-important input lags.

"For fast-paced games, there is one big problem with the flow shown on this diagram, and the name of the problem is "latency" (a.k.a. 'input lag')

Input Lag: the Worst Nightmare of an MMO developer

Note: if your game is slow- or medium-paced (including casino-like games such as poker), you can safely skip this section.

As noted above, for MMOs a lot of concerns is about relation between two times: input lag, and user expectations about it. Let's consider both of them in detail.

Input Lag: User Expectations

First, let's take a look at user expectations, and of course, user expectations are highly subjective by definition. However, there are some common observations which can be obtained in this regard. As a starting point, let's quote Wikipedia [Wikipedia.InputLag]:

> **"Testing has found that overall "input lag" (from controller input to display response) times of approximately 200 ms are distracting to the user."**

Let's take this magic number of 200ms as a starting point for our analysis (give or take, the number is also confirmed in several other sources[[TODO: add links]], and is also consistent with human reaction time [LippsEtAl], so it is not just Wikipedia taking it out of blue).

On the other hand, let's note that strictly speaking, it is not exactly 200ms, and it certainly varies between different genres. Still, even for the most time-critical games the number below 100-150ms is usually considered "good enough", and for any real-time interaction the lag of 300ms will be felt easily by lots of your players (though whether it will feel "bad" is a different story). To be more specific, for the remaining part of this Chapter let's consider two sample games: one being a simple OurRPG with input lag tolerance of 300 ms (let's assume it doesn't have fights and is more about social interactions, which makes it less critical to delays), and another game being OurFPS with input lag tolerance of 150ms.

Let's also note that these 150-300ms of input lag tolerance is just a fact of life (closely related to human psychology/physiology/etc.) so that we cannot really do much about it.

Input Lag: How Much We Have Left for MMO

The very first problem we have is that there are several things eating out of this 200ms allocation (even without our MMO code kicking in). This is lag introduced by game controller, lag introduced by rendering engine (that depends on many things, including such things as the size of render-ahead queue), and display lag (mostly introduced by LCD monitors).

> "These 150-300ms of input lag tolerance is just a fact of life (closely related to human psychology/physio so that we cannot really do much about it.

Typical mouse lag is 3-6ms [TomsHardware.GraphicsCardsMyths], less for gaming mice. For our purposes, let's account for any game controller lag as 5ms.

Typical rendering engine lags vary between 50ms and 150ms. 50ms (=3 frames at 60fps), is rather tricky to obtain, and is not that common, but still possible. More common number (for 60fps games) is 67ms (4 frames at 60fps), and 100-133ms are not uncommon either [Leadbetter2009].

Typical display lag (not to be confused with pixel response time, which is much lower and is heavily advertised, but it is not the one that usually kills the game) starts from 10ms, has a median of a kind around 40ms, and goes all the way to 100ms [DisplayLag.Display-Database].

It means that out of the original 150-300ms we originally had, we need to subtract a number from 60ms to 255ms. Which means that in quite a few cases the game is already lagging even before an MMO and network lag has kicked in ☹ .

To be a bit more specific, let's note that we cannot really control such things as mouse lag and display lag; we also cannot realistically say "hey guys, get the Absolutely Best Monitor", so at least we should aim for a median player with a median monitor. Which means that we should assume that out of our 150-300 ms, we need to subtract around 45ms (5ms or so for game controller/mouse, and 40 for a median monitor).

Now let's take a look at the lag, introduced by a rendering engine. Here, we CAN make a difference. Moreover, I am arguing that

for MMOs, rendering latencies are even more important than for single-player games

The point here is that for a single-player game, if we'd manage to get overall input lag say, below 100ms, it won't get that much of an improvement for the player (as long as it is fair to all the players), as this number is below typical human ability to notice things. However, for an MMO, where we're much closer to the magic 150-300ms because of RTTs, effects of the reduced latency will be significantly more pronounced. In other words, the difference between 100ms and 50ms for a single-player game won't feel the same as the difference between 200ms and 150ms for an MMO.

For the purposes of our example calculation, let's assume we've managed to get a rendering engine with a pretty good 67ms latency. This (combined with 45ms mentioned above) means that we've already eaten 112ms out of our 150-300ms initial allocation. And even if everything else will work lightning fast, we need to have RTT<40ms for OurFPS, and RTT<180ms for OurRPG.[1]

---

[1] in practice, it is even worse, see further discussion in "Accounting for Packet Losses and Jitter" section below 🙁

Input Lag: Taking a Bit Back

One trick which MAY be used to get a bit of "input lag" back is by introducing client-side animations. If, immediately after the button press, client starts some animation (or makes some sound, etc.), while at the same time sending the request to the server side – from end-user perspective the length of this animation is "subtracted" from the "input lag". For example, if in a shooter game you'll add a 50ms trigger pulling animation (while sending the shot right after the button press) – from player's perspective, the "Input Lag" will start 45ms later, so we'll get these 45ms back. Adding tracers to the shoots is known to create a feeling that bullets travel with limited speed, buying another 3 or so frames (50 ms) back (however, tracers are more controversial at least at close distances).

While capabilities of such tricks are limited, when dealing with the Input Lag, every bit counts, so you should consider if they are possible for your game.

[[TODO? – add another diagram to illustrate it and/or add it to further diagrams?]]

RTT

Now let's take a look at that RTT monster,[2] which is the worst nightmare for quite a few of MMO developers out there. RTT (="Round-Trip Time") depends greatly on the player's ISP (and especially on the "last mile"), but even in a very ideal case, there are hard limits on "how low you can go with regards to RTT". Very roughly, for RTT and, depending on the player's location, you can expect ballpark numbers shown in Table VII.1 (this is assuming the very best ISPs etc.; getting worse is easy, getting significantly better is usually not exactly realistic):

Player Connection	RTT (not accounting for "last mile")
On the same-city "ring" or "Internet Exchange" as server (see [Wikipedia.InternetExchanges], but keep in mind that going out of the same city will increase RTT)	~10-20 ms
Inter-city, cities separated by distance D	At the very least, 2*D/ c_{fib} (c_{fib} being speed of light within optical fiber, roughly c_{vacuum}/1.5, or ~2e8 m/s). Practically, add around 20-

	50ms depending on the country.
Trans-US (NY to SF)	At the very least (limited by c_{fib}) ~42 ms; in practice – at least 80 ms.
Trans-atlantic (NY to London)	At the very least (limited by c_{fib}) ~56 ms [Grigorik2013]; in practice – at least 80 ms.
Trans-pacific (LA to Tokyo)	At the very least (limited by c_{fib}) ~90 ms, in practice – at least 120ms.
A Really Long One (NY to Sydney)	At the very least (limited by c_{fib}) ~160 ms [Grigorik2013]; in practice – at least 200 ms.

In addition, you need to account for player's "last mile" as described in Table VII.2:

	Additional "last-mile" RTT
Added by player's "last mile": cable	[Grigorik2013] reports ~25ms, my own experience for games is about 15-20ms[3]
Added by player's "last mile": DSL	[Grigorik2013] reports ~45ms, my own experience for games is more like 20-30ms[3]
Added by player's Wi-Fi	~2-5 ms
Added by player's concurrent download	Anywhere from 0 to 300-500ms

Several things to keep in mind in this regard:

- If your server is sitting with a good ISP (which it should), it will be pretty close to the backbone latency-wise. This means that in most of "good" cases, real player's latency will be one number from Table VII.1, plus one or more numbers from Table VII.2 (and server's "last mile" latency can be written off as negligible); it is still necessary to double-check it (for example, by pinging from another server).
- The numbers above are for hardware servers sitting within datacenters. Virtualized servers within the cloud tend to have higher RTTs (see Chapter [[TODO]] for further discussion), with occasional delays (when your cloud neighbour suddenly started to eat more CPU/bandwidth/...) easily going into multiple-hundreds-of-ms range ☹. BTW, you *can* get cloud without virtualization (which will eliminate these additional delays), more on it in Chapter [[TODO]].

- LAN-based games (with wired LAN having RTTs below 1 ms) cannot be really compared to MMOs latency-wise. If your MMO needs comparable-to-LAN RTT latency to be playable – sorry, it won't happen (but see below about the client-side prediction which may be able to alleviate the problem in many cases, though at the cost of significant complications)
- No, CDN won't work for games (at least not those traditional CDNs which are used to improve latencies for web sites)

And while we're at it, three things that you'll certainly need to tell to your players with regards to RTT/latency:

- no, better bandwidth doesn't necessarily mean better latency (this will be necessary to tell answering questions such as "how comes that as soon as I've got better 30Mbit/s connection, your servers started to lag on me?")
- it is easy to show whatever-number-we-want in the client as a "current latency" number, but comparisons of the numbers reported by different games are perfectly pointless (this actually is a Big Fat Argument to avoid showing the numbers at all, though publishing the number is still a Business Decision).
- When saying "it was much better yesterday", are you sure that nobody in your household is running a huge download?

"No, better bandwidth doesn't necessarily mean better latency

---

[2] this monster is quietly hiding behind the curtain of LAN until you start to test with real-world RTTs
[3] the difference can be attributed to downloads which tend to cause longer RTTs; also gamers tend to invest in better connectivity

Back to Input Lag

Ok, from Table VII.1 and Table VII.2 we can see that in the very best case (when both your server and client are connected to the very same intra-city ring/exchange, everything is top-notch, last mile is a non-overloaded cable, no concurrent downloads running in the vicinity of the client while playing, etc. etc.) – we're looking at 35-45ms RTT. When we compare it to our remaining 40-180ms, we can say "Great! We can have our cake and eat it too, even for OurFPS!" And it might indeed work (though it won't be *that* bright, see complications in "Accounting for Losses and Jitter" subsection below). On the negative side, it means having a server on each and every city exchange, and losing (or at least penalizing) lots of players who don't have this kind of connectivity ☹ .

"Within the same (large)

Within the same (large) country, the best-possible RTT goes up to around 80-100ms. Which means that with a simple diagram on Fig VII.1 we MIGHT be able to handle OurRPG, but not OurFPS (though again, see complications below). Country-specific servers are very common, and are not that difficult to implement and maintain, but they still restrict flexibility of your players (and also can have adverse effects on "player critical mass" [[TODO! add discussion on "critical mass" to Chapter I]]. Single-continent servers (with RTTs in the range of 100-120ms) are close

country, the best-possible RTT goes up to around 80-100ms.

cousins of country-specific ones, and are also frequently used for fast-paced games.

Purely geographically, for US the best server location for a time-critical game would be somewhere in Arkansas 🙂 . More realistically (and taking into account real-world cables), if trying to cover whole US with one single server, I'd seriously consider placing it at a Dallas or Chicago datacenter, it would limit the maximum RTT while making the games a bit more fair.

If you want a world-wide game, then maximum-possible RTT goes up to 220ms. Worse than that, there is also significant difference for different players. While simple data flow shown on Fig VII.1 might still fly for a relatively slow-paced world-wide RPG (think Sims), but MMOFPS and other combat-related things are usually out of question.

Data Flow Diagram, Take 2: Fast-Paced Games Specifics

Note: if your game is slow- or medium-paced (including casino-like games such as poker), you can safely skip this section.

Note 2: if your game is fast-paced (think MMOFPS or MMORPG), the approach described with regards to Take 2 Diagram, still isn't likely to produce a game which doesn't feel "laggy". However, please keep reading, as we will discuss the remaining problems, and the ways to deal with them, in Take 3.

Ok, our calculations above seem to show that we can get away with a simplistic diagram from Fig. VII.1 even for some of fast-paced fps-based games.

Well, actually, we cannot, at least not yet: there is one more important network-related complication which we need to take into account. This is related to mechanics of the Internet (and to some extent – to the mechanics of simulation-based games).

Internet is Packet-Based, and Packets can be Lost

First of all, let's talk about mechanics of the Internet (only those which we need to deal with at the moment). I'm not going to go into any details or discussions here, let's just take it as an axiom that

when the data is transmitted across the Internet, it always travels within packets,
and each of these packets can be delayed or lost

This stands regardless of exact protocol being used (i.e. whether we're working on top of TCP, UDP, or something really exotic such as GRE). In addition, let's take as another axiom that

each of these packets has some overhead

For TCP the overhead is 40+ bytes per packet, for UDP – it is usually 28 bytes per packet (that's not accounting for Ethernet headers, which add their own overhead). For our current purposes, exact numbers don't matter too much, let's just note that for small updates they're substantial.

"For TCP the overhead is 40+ bytes per packet, for UDP – it is usually 28 bytes per packet (that's not accounting for Ethernet headers).

Now let's see how these observations affect our game data flow.

Cutting Overhead

The first factor we need to deal with, is that for a fast-paced game sending out a world update in response to each and every input is not feasible. This (at least in part) is related to the per-packet overhead we've mentioned above. If we need to send out an update that some PC has started moving (which can be as small as 8 bytes), adding overhead of 28-40 bytes on top of it (which would make 350-500% overhead) doesn't look any good.

That's at least one of the reasons why usually simulation is made within a pretty much classical "game loop", but with rendering replaced with sending out updates:

```
1  while(true) {
2    TIMESTAMP begin = current_time();
3    process_input();
4    update();
5      //update() normally includes all the world simulation,
6      //  including NPC movements, etc.
7    post_updates_to_clients();
8      //here, we're effectively combining all the world updates
9      //  which occurred during current 'network tick'
10     //  into as few packets as possible,
11     //  effectively cutting overhead
12   TIMESTAMP elapsed = current_time()-begin;
13   if(elapsed<NETWORK_TICK)
14     sleep(NETWORK_TICK-elapsed);
15 }
```

With this approach, we're processing all the updates to the "game world" one "network tick" at a time. Size of the "network tick" varies from one game to another one, but 50ms per tick (i.e. 20 network ticks/second) is not an uncommon number (though YMMV may vary significantly(!)).

Note that on the server-side (unlike for client-side game loop from Chapter V) the choice of different handling for time steps is limited, and it is usually the variation above (the one waiting for remainder of the time until the next "tick") which is used on the server-side. Moreover, usually it is written in ad-hoc-FSM (a.k.a. event-driven) style, more or less along the following lines:

```
void MyFSM::process_event(TIMESTAMP current_time,
  const Event& event) {
  //'event' contains ALL the messages that came in
  //   but are not processed yet
  process_input(event);
  update();
  post_updates_to_clients();
  post_timer_event_to_myself(SLEEP_UNTIL,current_time+NETWORK_TICK);
}
```

Accounting for Losses and Jitter

So far so good, but we still didn't deal with those packet losses and sporadic delays (also known as "jitter").

If we won't do anything (and will stay within the simple schema shown on Fig. VII.1) – then each lost (or substantially delayed) packet will mean visible (and unpleasant) effects on the player's screen: everything will stop for a moment, and then "jump" to the correct position when the next packet arrives.

"If we will stay within the simple schema shown on Fig. VII.1 – then each lost packet will mean visible (and unpleasant) effects: on player's screen everything will stop for a moment, and then "jump" to the correct position when the next packet arrives.

To deal with it we need to introduce a buffer on the client side, so that if the packet traveling from server to the client is lost or delayed, we have time to wait for it to arrive (or for its retransmitted copy, sent in case of packet loss, to arrive). This is very much the same thing which is routinely done for other jitter-critical stuff such as VoIP.

The delay we need to introduce with this buffer, depends on many factors, but even with the most aggressive UDP-based algorithms (such as the ones which re-send the whole state on each network tick, or the one described at the very end of [GafferOnGames.DeterministicLockstep], see Chapter [[TODO]] for further discussion), the minimum we can do is having a buffer of one network tick to account for one-lost-packet-in-a-row.[4] In practice, the buffer of around three "network ticks" is usually desirable (that's for really aggressive retransmit policies mentioned above).

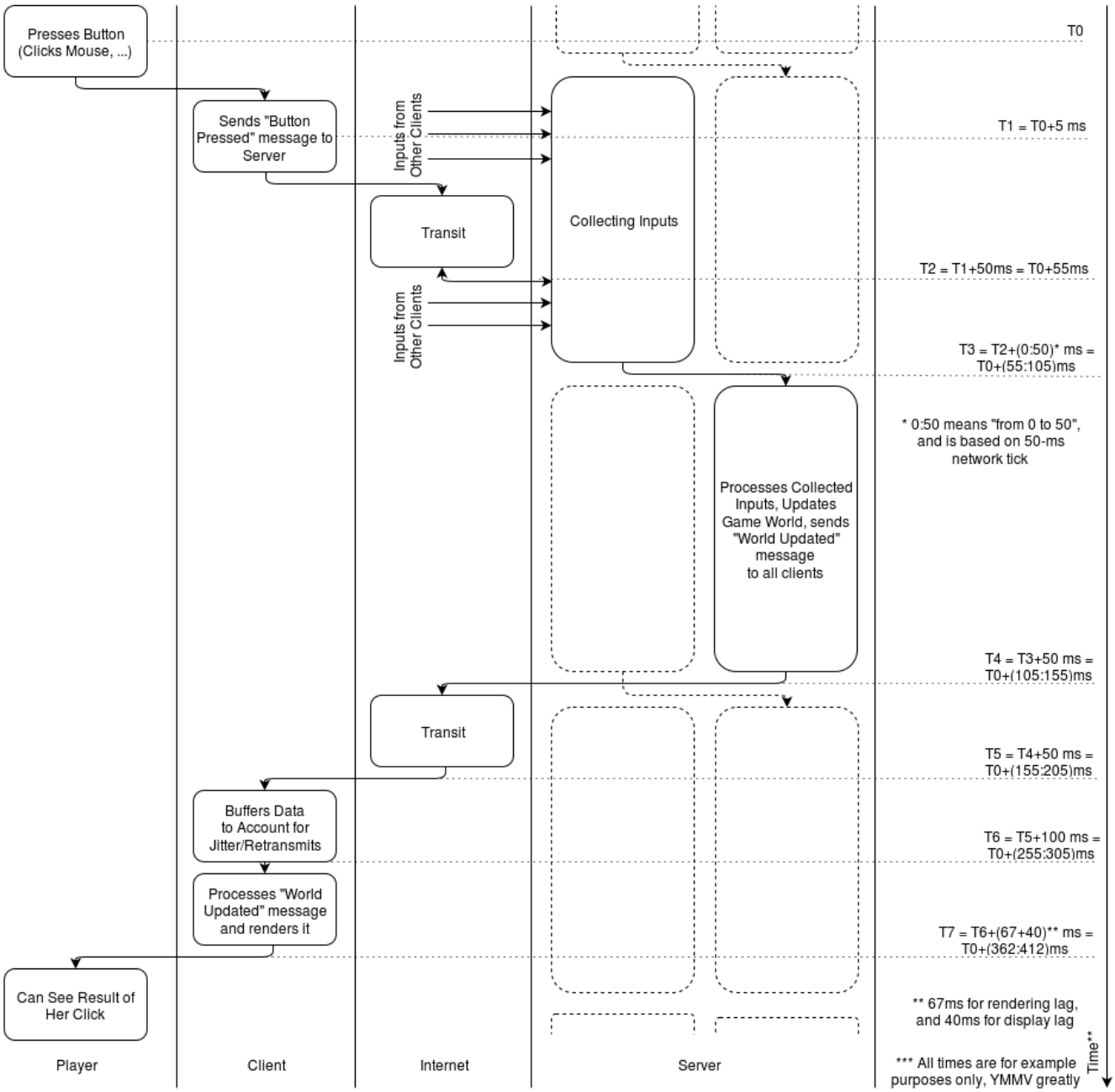
If going TCP route, we're speaking about retransmit delays of the order of RTT, which will usually force us to have buffer delays of the order of N*RTT (like 3*RTT) 🙁 , which is substantially higher. These delays-in-case-of-packet-loss are one of the Big Reasons why TCP is not popular (to put it mildly) among the developers of fast-paced games. More on it in Chapter [[TODO]].

One thing which should be noted in this regard, is that to some extent this buffer MAY be somewhat reduced due to render-ahead buffering used by the rendering engine. It would be incorrect, however, to say that you can simply subtract the time of render-ahead buffer by the rendering engine (by default 3 frames=50ms for DirectX) from the time which we need to add for RTT purposes. Overall, this is one of those things which you'll need to find out for yourself.

Take 2 Diagram

These two considerations discussed above (one related to overhead and game loop, and

another related to client-side buffer) lead us to the diagram on Fig VII.2:



STILL NOT REALLY PRACTICAL FOR FAST-PACED GAMES:
SEE FIG VII.3 FOR FURTHER IMPROVEMENTS

Fig VII.2

If we calculate all the delays on the way (using some not-too-unreasonable assumptions mentioned on Fig VII.2), we can see that for RTT=100ms overall delay reaches really annoying 350-400ms (as always, YMMV). As “normal” latency tolerances (as discussed above) are within 150-300ms, this in turn means that for the majority of fast-paced simulation games out there, implementing your system along the lines of this diagram will lead to really “laggy” player experience 🙁 .

One additional problem with this approach is that we effectively have our visual frame rate equal to “network tick”; as “network ticks” are usually kept significantly lower than 60 per second (in our examples it was 20 per second) – it means that we’ll be rendering at 20fps instead of 60fps, which is certainly not the best thing visually.

It leads us to the next iteration of our flow diagram, which introduces substantial (and non-

trivial) client-side processing.

Data Flow Diagram, Take 3: Client-Side Prediction and Interpolation

Note: if your game is slow- or medium-paced (including casino-like games such as poker), you can safely skip this section too.

So, we have these two annoying problems: one is lag, and another one is client-side frame rate. To deal with them, we need to introduce some client-side processing. I won't go into *too much* details here; for further discussion please refer to the excellent series on the subject by Gabriel Gambetta: [Gambetta]; while he approaches it from a bit different side, all the techniques discussed are exactly the same.

Client-Side Interpolation

The first thing we can do, is related to the client-side update buffer (the one we have just introduced for Take 2). To make sure that we don't render at "network tick" rate (but rendering at 60fps instead), we can (and should) interpolate the data between the "current frame" (the last one within the update buffer), and "previous frame" in the update buffer. This will make the movement smoother and we'll get back our 60fps rendering rate. Such client-side interpolation is quite a trivial thing and doesn't lead to any substantial complications. On the negative side, while it does make movement smoother, it doesn't help to improve input lag 🙁 .

Client-Side Extrapolation a.k.a. Dead Reckoning

"The next thing we can do, is to go beyond interpolation, and to do some extrapolation.

The next thing we can do, is to go beyond interpolation, and to do some extrapolation. In other words, if we have not only object positions, but also velocities (which can be either transferred as a part of the "World Update" message or calculated on the client-side), then – in case if we don't have the next update yet because the packet got delayed – we can extrapolate the object movement to see where it *would move if nothing unexpected happens*.

The simplest form of such extrapolation can be done by simple calculation of x1=x0+v0, but can also be more complicated, taking into account, for example, acceleration. This is known as "dead reckoning", though the latter term is used in several similar, but slightly different cases, so I'll keep using the term "extrapolation" for the specific thing described above.

The benefit of such extrapolation is that we can be more optimistic in our buffering, and not to account for the worst-case, when 3 packets are lost (extrapolating instead in such rare cases). In practice it often means (as usual, YMMV) that we can reduce the size of our buffer down to one outstanding frame (so at each moment we have both "current frame" and "previous frame", but nothing else).

Running into the Wall, and Server Reconciliation

"What if when we're extrapolating NPC's movement, he runs into the wall?

On the flip side, unlike interpolation, extrapolation causes significant complications. The first set of complications is about internal inconsistencies. What if when we're extrapolating NPC's movement, he runs into the wall? If this can realistically happen within our extrapolation, causing visible negative effects, we need to take it into account when extrapolating, and detect when our extrapolated NPC collides, and maybe even to start an appropriate animation. How far we want to go on this way – depends (see also "Client-Side Prediction" section below), but it MAY be necessary.

The second set of extrapolation-related issues is related to so-called "server reconciliation". It happens when the update comes from the server, but our extrapolated position is different from server's one. This difference can happen even if we've faithfully replicated all the server-side logic on the client side, just because we didn't have enough information at the point of our extrapolation. For example, if one of the other players has pressed "jump" and this action has reached the server, on our client-side we won't know about it at least for another 100ms or so, and therefore our perfectly faithful extrapolation will lead to different results than the server's one. This is the point when we need to "reconcile" our vision of the "game world" with the server-side vision. And as our server is authoritative and is "always right", it is not that much a reconciliation in a traditional sense, but "we need to make client world look as we're told".

On the other hand, if we implement "server reconciliation" as a simple fix of coordinates whenever we get the authoritative server message, then we'll have a very unpleasant visual "jump" of the object between "current" and "new" positions. To avoid this, one common approach (instead of *jumping* your object to the received position) is to start new prediction (based on new coordinates) while continuing to run "current" prediction (based on currently displayed coordinates), and to display "blended" position for the "blending period" (with "blended" position moving from "current" prediction to "new" prediction over the tick). For example: displayed_position(dt) = current_predicted_position(dt) * (1-alpha(dt)) + new_predicted_position(dt) * alpha(dt), where alpha(t) = dt/BLENDING_PERIOD, and 0 <= dt < BLENDING_PERIOD.

[[TODO: other methods?]]

Client-Side Prediction

With client-side interpolation and client-side extrapolation, we can reduce latencies a bit; in our example on Fig. VII.2 we should be able to get back around 50ms (and also get real frame rate up to 60fps). However, even after these improvements it is likely that the game will feel "sluggish" (in our example, we'll have the overall input lag at 300+ms, which is still pretty bad, especially for a fast-paced game).

To improve things further, it is common to use "Client-Side Prediction". The idea here is to start moving player's own PC as soon as the player has pressed the button, eliminating this "sluggish" feeling completely. Indeed, within the client we do know what PC is doing – and can show it; and if we're careful enough, our prediction will be *almost-the-same* as the server authoritative calculation, at least until the PC is hit by something

that has suddenly changed trajectory (or came out of nowhere) within these 300ms or so.

> "The idea here is to start moving player's own PC as soon as the player has pressed the button, eliminating this "sluggish" feeling completely.

On the negative side, Client-Side Prediction causes quite serious discrepancies between "game world as seen by server" and "game world as seen and shown by client". In a sense, it is similar to the "reconciliation problem" which we've discussed for "Client-Side Extrapolation", but is usually more severe than that (due to significantly larger time gap between the extrapolation and reconciliation). One of the additional things to be kept in mind here, is that keeping a list of "outstanding" (not confirmed by server yet) input actions, and re-applying them after receiving every authoritative update is usually necessary, otherwise quite unpleasant visual effects can arise (see [Gambetta.Part2] for further discussion of this phenomenon).

In addition, the problem of PC-running-into-the-wall (once again, in a manner similar to client-side extrapolation, but with more severe effects due to larger time gap) usually needs to be addressed.

To make it even more complicated, inter-player interactions can be not as well-predictable as we might want, so making irreversible decisions (like "the opponent is dead because I hit him and his health dropped below zero") purely on the client side is usually not the best idea (what if he managed to drink a healing potion which you don't know about yet as the server didn't tell you about it?). In such cases it is better to keep the opponent alive on the client side for a few extra milliseconds, and to start the rugdoll animation only when the server does say he's really dead; otherwise visual effects like when he was starting to fall down, but then sprang back to life, can be very annoying.

Take 3 Diagram

Adding these three client-side things gets us to the following Fig VII.3:

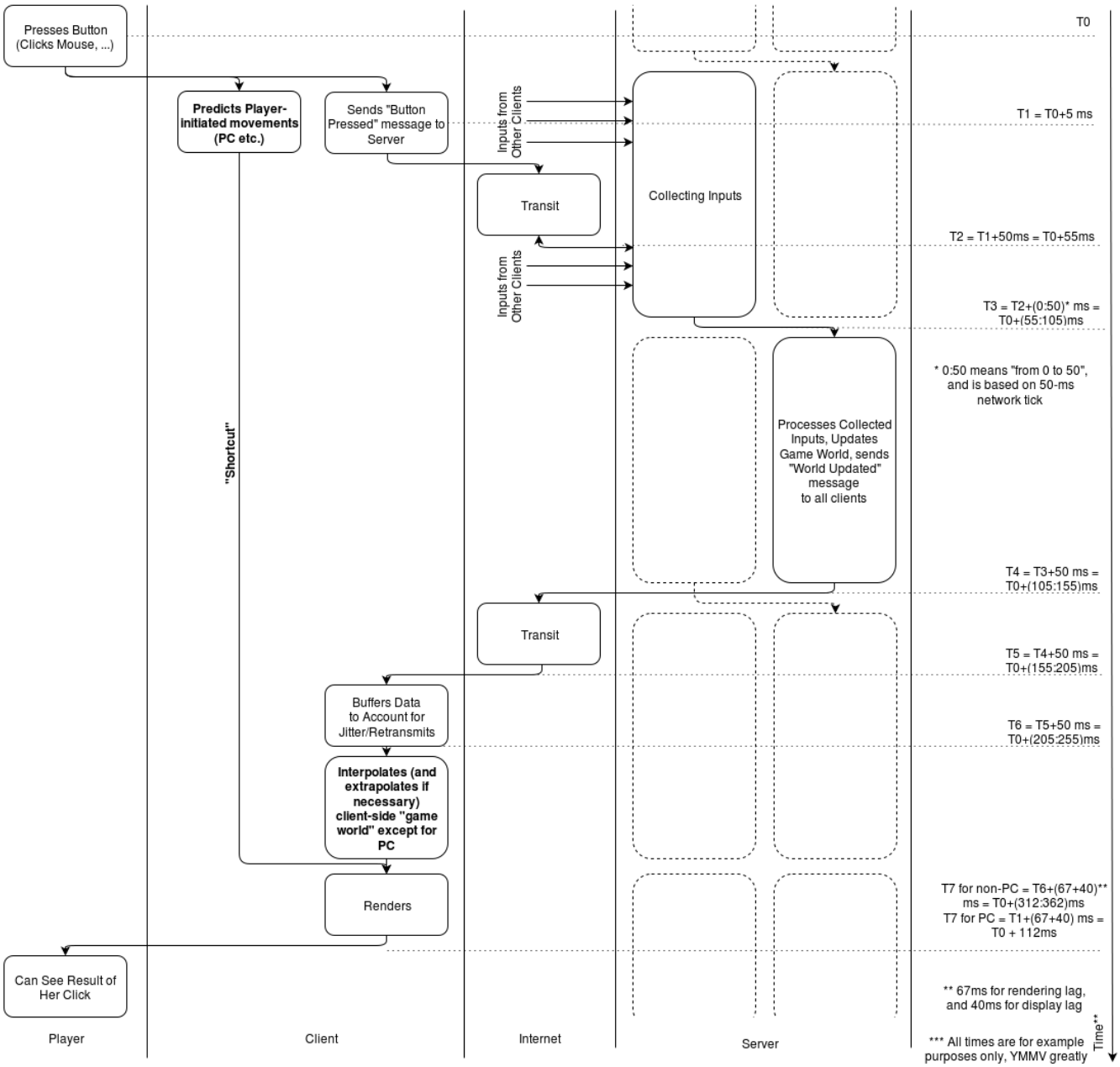


Fig VII.3

As we can see, processing of the authoritative data coming from server is still quite slow (despite an about 50ms improvement due to reducing size of client-side buffer, which became possible due to relying on client-side extrapolation when the server packet is delayed). But the main improvement in perceived responsiveness for those-actions-initiated-by-player (and it is these actions which cause the “laggish” feeling, as timing of the actions by the others is not that obvious) comes from the Client-Side Prediction “shortcut”. Client-Side Prediction is processed purely on the client-side, from receiving controller input, through client-side prediction, and directly into rendering, without going to server at all, which (as you might have expected) helps latency a damn lot. Of course, it is just a “prediction”, but if it is 99% correct 99% of the time (and in the remaining cases the difference is not too bad) – it is visually ok.

So, with Fig VII.3 (and especially with client-side prediction) we've managed got quite an improvement at least for those actions initiated by PC; at 112ms the game won't feel too sluggish. But can we say that with these numbers, everything is really good now? Well, sort of, but not exactly. The remaining problem is that there is still a significant (and unavoidable) lag between any update-made-by-server and the moment when our player will see it. This (as [Gambetta.Part4] aptly puts it) is similar to living in the world where speed of light is slow, so we see what's going on, with a perceivable delay.

"The remaining problem is that there is still a significant (and unavoidable) lag between any update-made-by-server and the moment when our player will see it.

In turn, for some Really Fast-Paced games (think shooters) it leads to unpleasant scenarios when I'm as a player making a perfect shoot from a laser weapon, but I'm missing because I'm actually aiming at the position-of-the-other-player which I can see, and it is an inherently *old* position of the player (behind by 200ms or so even compared to server's authoritative world, and even more compared to his own vision). And this is the point where we're getting into realm of controversy, known as Lag Compensation.

Lag Compensation

The things we've discussed above, are very common, and are known to work well. The next thing, known as Lag Compensation (see also [Gambetta.Part4]), is much more controversial.

Lag Compensation is aimed to fix the problem outlined above, the one when I'm making a perfect shot, and missing because I'm actually aiming at the *old* position of the other player.

The idea behind Lag Compensation is that the server (keeping an authoritative copy of everything) can reconstruct the world at any moment, so when it receives your packet saying you're shooting *with your timestamp* (and all the other data such as angle at which you're aiming etc. etc.) , it can reconstruct the world at the moment of your shot (even according to your representation), and make a judgement whether you hit or missed, based on that information. This can be used to compensate for the delay, and therefore to make your "clean shots" much better.

On the other hand (in addition to some other oddities described in [Gambetta.Part4]), Lag Compensation is wide open to cheating ☹ . If I can send *my timestamp,* and server will implicitly trust it – I am able to cheat the server, making the shot a bit later while pretending it was made a bit earlier.

In other words, Lag Compensation can be used to compensate not only for Network Lag, but also for Player Lag (poor player reaction), as they're pretty much indistinguishable from the server point of view

While effects of cheating can be mitigated to a certain extent by putting a limit on "how much client timestamp is allowed to differ from server timestamp", it is still cheating (and as soon as the potential for cheating is gone, so is any benefit from compensation).

That's exactly why Lag Compensation is controversial, and I suggest to avoid it as long as possible. If you cannot avoid it – good luck, but be prepared to cheaters starting to abuse your game as soon as you're popular enough.

On the other hand, three client-side techniques above (Client-Side Interpolation, Client-Side Extrapolation, and Client-Side Prediction) do not make server trust the client, and are therefore inherently impossible to this kind of abuse.

There Are So Many Options! Which ones do I need?

With all these options on the table, an obvious question is "hey, what exactly do I need for my game?" Well, this is a Big Question, with no good answer until you try it for your specific game (over real link and/or over latency simulator). Still, there are some observations which may help to get a reasonable starting point:

- if your game is slow-paced or medium-paced (i.e. actions are in terms of "seconds") – all chances are that you'll be fine with the simplest dataflow (the one shown on Fig VII.1)
- if your game is MMORPG or MMOFPS – you'll likely need either the dataflow on Fig VII.2, or the one on Fig VII.3
  - in this case, it is often better to start with the simpler one from Fig VII.2 and add things (such as Client-Side Interpolation, Client-Side Extrapolation, Client-Side Prediction) gradually, to see if you've already got the feel you want without going into too much complications
  - if after adding all the "Client-Side *" stuff you still have issues – you may need to consider Lag Compensation, but beware of cheaters!
  - for really serious development, you may need to go beyond these techniques (and/or combine them in unusual ways), but these will probably be too game-specific to discuss them here. In any case, what can be said for sure is that you certainly need to know about the ones discussed in this Chapter, before trying to invent something else 🙂 .

[[To Be Continued…

This concludes beta Chapter VII(a) 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 VII(b), "Publishable State"]]

[–] References

[Wikipedia.InputLag] "Input Lag", Wikipedia
[LippsEtAl] David B. Lipps, Andrzej T. Galecki, James A. Ashton-Miller, "On the Implications of a Sex Difference in the Reaction Times of Sprinters at the Beijing Olympics", PLOS ONE
[TomsHardware.GraphicsCardsMyths] Filippo L. Scognamiglio Pasini, "The Myths Of Graphics Card Performance: Debunked", tom's Hardware
[Leadbetter2009] Richard Leadbetter, "Console Gaming: The Lag Factor"
[DisplayLag.Display-Database] "Display Input Lag Database"
[Wikipedia.InternetExchanges] "List of Internet exchange points by size"
[Grigorik2013] Ilya Grigorik, "High Performance Browser Networking", Chapter 1
[GafferOnGames.DeterministicLockstep] Glenn Fiedler, "Deterministic Lockstep"

[Gambetta] Gabriel Gambetta, “Fast-Paced Multiplayer”
[Gambetta.Part2] Gabriel Gambetta, “Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation”
[Gambetta.Part4] GabrielGambetta, “Fast-Paced Multiplayer (Part IV): Headshot! (AKA Lag Compensation)”

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG Server-Side. Programming Languages*

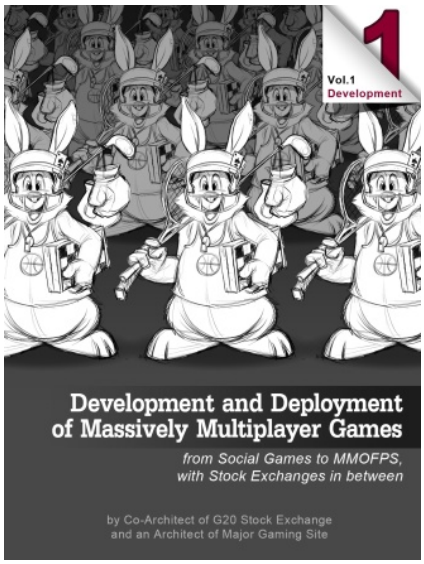
MMOG: World States and Reducing Traffic »

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

Copyright © 2014-2016 ITHare.com

MMOG: World States and Reducing Traffic

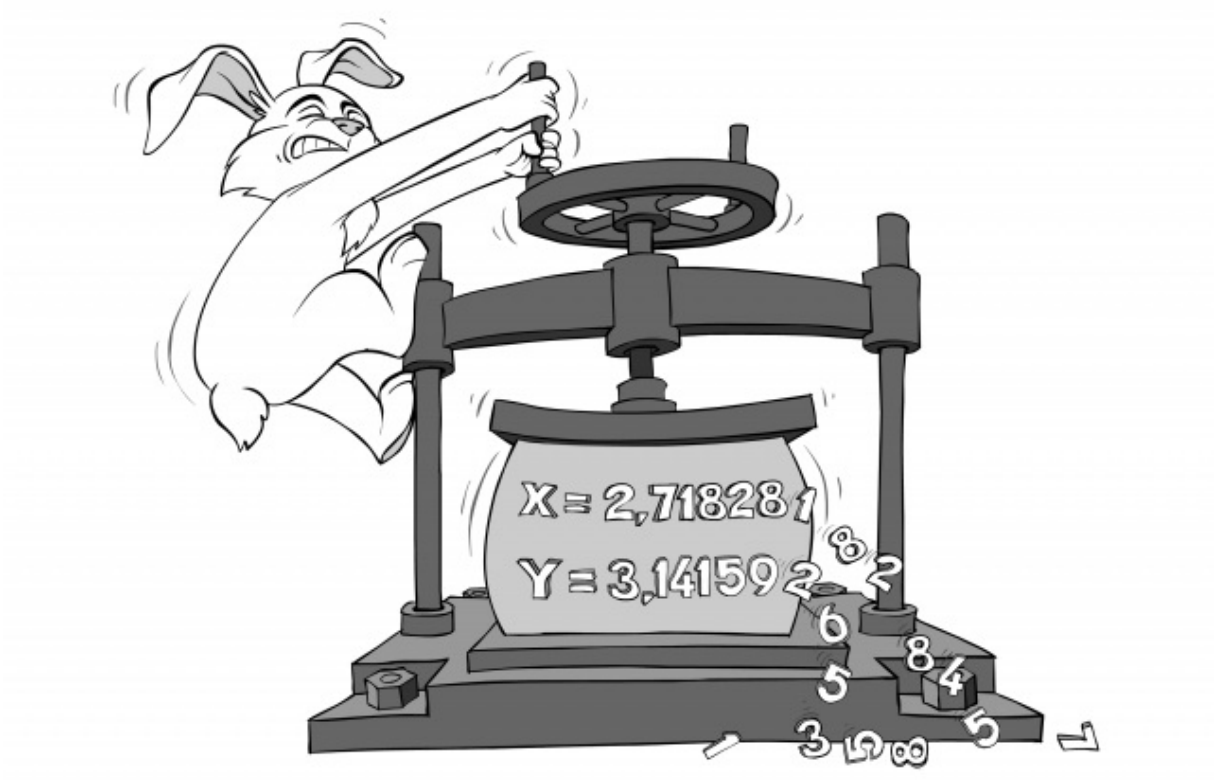
posted February 1, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko



[[*This is Chapter VII(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.]]

Ok, so we've finished describing data flows which may apply to your game, and can now go one level deeper, looking into specifics of those messages going between client and server. First, let's take a closer look at the message that tends to cause most of trouble at least for fast-paced simulation-based games. This is "World Update" message from Fig. VII.1-Fig. VII.3, which in turn is closely related to Publishable World State.



Server-Side, Publishable, and Client-Side Game World States

Among aspiring simulation-based game developers, there is often a misunderstanding about Game World State – "Why we need to care about different states for our Game

World, and cannot have only one state, so that Server-Side State is the same as Client-Side One?"

The answer is that "Well, depending on your game, you MIGHT be able to have one state, but for simulation games, in many cases you won't". The problem here is purely technical, but very annoying – it is a problem of bandwidth.

"The most important reason to minimize bandwidth is that traffic is expensive.

The most important reason to minimize bandwidth is that traffic is expensive. While traffic prices go down and, as of beginning of 2016, you can get unmetered 1Gbit/s for around $500-1000/month, and unmetered 10GBit/s for around $3-5K/month, it is still far from being free. On the other hand, if you're too wasteful with your traffic (like trying to send all the updates to all the meshes over the network), it simply won't fit into your player's "last mile" (connection from him to his ISP). And if you've done a good job already, still keep in mind that, as an additional incentive, making your Publishable State smaller tends to make updates to it smaller, and the smaller updates are – the less chances you have to overload your player's "last mile" (and overloading "last mile" inevitably leads to latencies going through the roof for that specific player).[1]

Note that here we're not discussing systems based on so-called "deterministic lockstep" network models; with all their simplicity, they don't work well for MMOs (in fact, [GafferOnGames.SnapshotsAndInterpolation] doesn't recommend it for Internet games with over 4 players).

---

[1] interestingly, sometimes reducing server packet size MAY help even if client's "last mile" overload is caused by a concurrent download, as there are *some* routers out there configured to give preference to smaller packets; unfortunately, I don't have stats on how widespread this effect is in practice

Client-Side State

Let's consider an example MMORPG game, OurRPG. Let's assume that our players can move within some 3D world, talk, fight, gain experience, and so on. Physics-wise, let's assume that we want to have rigid body physics and ragdoll animations, but our fights are very simple and don't really simulate physics and have fight movements animated instead (think "Skyrim").

If we have our game as a single-player, the only thing we will need, would be a Client-Side State – complete with all the meshes (with thousands of triangles per character), textures, and so on.

Server-Side State

Now, as we're speaking about MMOs, we need a Server-Side State. And one thing we can notice about this Server-Side State is that it doesn't need to be as detailed as Client-Side

State.

As we don't need to render anything on the server side, we usually can (and SHOULD) use much more low-poly 3D models on the server side

"In practice, for most classical RPGs you can get away with simulating each of your PCs and NPCs as a box (parallelepiped), or as a prism (say, hexagonal or octagonal one).

Actually, to keep the number of our servers within reason, we need to leave only the absolute minimum of processing on the server side, and this absolute minimum is defined as "drop everything which doesn't affect gameplay". In practice, for most classical RPGs (those without karate-like fights where limb positions are essential for gameplay) you can get away with simulating each of your PCs and NPCs as a box (parallelepiped), or as a prism (say, hexagonal or octagonal one). Cylinders are also possible, though these usually apply if you don't really make a classical polygon-based 3D simulation. Models of your server-side rooms can (and SHOULD) also be simplified greatly – while you do need to know that there is a wall there with a lever to be pulled in the middle, in most cases you don't need to know the exact shape of the lever.

In extreme cases, you won't even need 3D on server side at all. While this is not guaranteed, start your analysis from checking if you can get away with 2D server-side simulation – even if you will need 3D, such analysis will help you to drop quite a few things which are unnecessary on the server side.

For OurRPG, however, we do need 3D on the server side (well, we want to simulate rigid body stuff and ragdolls, not to mention multi-level houses). On the other hand, we don't need more than hexagonal prism (with additional attributes such as "attacking/crouching/..." and things such as "animation frame number") to represent our PCs/NPCs; when it comes to rigid objects simulated on the server-side – they also can be represented using only a few dozens triangles each.

When we need to simulate ragdoll on the server-side – we won't even try to simulate movements of all the limbs. What we will do, however, is calculate movement of the center of mass of the dying character. While for some games this may happen to result in too-unrealistic movements, for some other games we might be able to get away with it (and doing it this way will save lots of CPU power on the server-side), so that's what we'll try first.

This polygon reduction will lead to a drastic simplification from the classical Client-Side State (the one we'll need to render the game).

Publishable State

Now, as we've got Server-Side State and Client-Side State, we need to pass the data from the Server-Side to the Client-Side. To do it, we'll need another state – let's name it Publishable State.

And one thing to note about Publishable State is that it usually SHOULD be simpler than Server-Side state. When discussing simplifications of the Publishable State compared to Server-Side state, let's observe the following:

- As a rule of thumb, Publishable State MAY be simplified compared to Server-Side State. For example, for OurRPG the following simplifications are possible:
  - to represent PCs/NPCs, we usually can (and therefore SHOULD) throw away all the meshes, and use only a tuple of (x,y,z,x-y-angle,animation-state,animation-frame)[23]
    - in addition to the tuple required for rendering, there likely to be dozens of fields such as "inventory", "relationships with the others", and so on; whether they need to be published, depends on your client-side logic.
      - By default (and until proven that you need specific field for the client side), avoid publishing these things. The smaller your publishable state is – the better.
      - In some cases, however, you may need them. For example, if your game allows to steal something from PC/NPC, then your client's UI will likely want to show other character's inventory to find out what can be stolen. This information about the other character's inventory MAY be obtained by request, or MAY be published. In the latter case, it becomes a part of publishable state. Note that making inventory publishable won't have too bad effect on the update size, as it will be optimized via delta compression (see "Delta Compression" subsection below); on the other hand, it will increase traffic during initializations/transitions, so depending on your game it still MAY make sense to exclude inventory from publishable state and make this information available on request from a client.
      - Even if you need such rarely-changing fields as a part of your publishable state, you usually SHOULD separate them from the frequently-changed ones (for example, into separate structs or something). This is related to them having different timing requirements, which potentially lead to different retransmit policies, and it is simpler to express these policies when you have separate structs. For example, inventory is updated rarely, and is usually quite tolerant to delays of the order of 3*RTT or so; as a result, it is usually unwise to be too aggressive with re-sending it (in other words, it is usually ok to send it once and to wait for 2*RTT for confirmation before re-sending it). On the other hand, coordinates and other rendering-related stuff do need to be updated in real-time, so you should be quite aggressive with re-sending them. More discussion on re-transmission policies will be provided in Chapter [[TODO]].

"to represent PCs/NPCs, we usually can (and therefore SHOULD) throw away all the meshes, and use only a tuple of (x,y,z,x-y-angle,animation-state,animation-frame)

  - to represent rigid objects, we again SHOULD throw away all the meshes and use

only (x,y,z,x-y-angle,x-z-angle,y-z-angle) tuple.

- Whenever we CAN make Publishable State smaller, we SHOULD do it (see reasoning about reducing bandwidth above).

---

[2] actually, we can use this representation for Server-Side too, but it may or may not be convenient for the Server-Side. On the other hand, removing meshes is an almost-must for Publishable State

[3] whether we need velocities to be published is not that obvious, see "Dead reckoning" section below

Why Not Keep them The Same?

"If we'd try to transfer all the thousands of triangles every "network tick" while our character is moving, we'd need to send around 100Kbytes per "network tick" per character, and if our PC can see 20 characters at the same time, and we're using 20 "network ticks" per second, we'll end up with 40MBytes/second/

Now let's go back to the question – why not use the very same Client-Side State as Server-Side State and as Publishable State? The answer is simple – because of bandwidth. Just compare – if we'd try to transfer all the thousands of triangles every "network tick" while our character is moving, we'd need to send around 100Kbytes per "network tick" per character, and if our PC can see 20 characters at the same time,[4] and we're using 20 "network ticks" per second, we'll end up with 100KBytes/tick/character * 20characters/player * 20ticks/second = 40MBytes/second/player; this would in turn mean that we can fit only 30 players in that $5K/month 10Gbit/s channel (not to mention that only a few people will be able to play the game), Big Ouch! With our Publishable State (and even before any compression techniques are used) it is more like 50 bytes/tick/character, or (with the same assumptions) is much more manageable 50bytes/tick/character * 20characters/player * 20ticks/second = 20KBytes/second/player.[5]

Throw in the reduced Server-Side CPU load (which you will be paying for) for simplified Server-Side State, and the need to have simplified Server-Side State and Publishable State becomes obvious.

---

[4] here we're implying that we've implemented "interest management" to avoid sending unnecessary stuff, see "Interest Management" section below for further discussion

[5] it can be reduced further (see "Compression" section below), but for the moment this 3+ orders of magnitude improvement will suffice.

Non-Sim Games and Summary

For non-simulation games (such as social games or blackjack), the difference between different States is much less pronounced, and in many cases Server-Side State MAY be the same as Publishable State (though Client-Side State often will still be different). For example, whenever a card is

dealt for a blackjack game, usually it is represented as an immediate update of the Server-Side State to reflect that the card is already dealt, and update to Server-Side State is immediately pushed to the Client. All the animation of the card being dealt, is processed purely on the Client-Side (so that Client-Side State is updated without any input from the Server while the card is flying across the table).

On the other hand, even if we try to generalize our findings over the whole spectrum of the MMO games (from social ones to MMOFPS), two observations can be made. First of all, whatever our game is, the following inequation should stand:

Publishable State <= Server-Side State <= Client-Side State

The second observation is the following:

we SHOULD work hard on reducing the size of Publishable State

Publishable State: Delivery, Updates, Interest Management, and Compression

Ok, so we've decided on our Publishable State (and have done it in a reasonably optimal way), and know how to update it on the server side. The next question we face is "How to deliver this Publishable State (including updates) from Server to Client?"

Of course, the most obvious way of doing it would be just to transfer the whole state once (when the client is connected), and then to transfer updates whenever the update of the Game World occurs (which may be "each network tick" for quite a few simulation-based games out there).

However, very often we can do better than that traffic-wise. And as reducing traffic is a Good Thing(tm) both for the reducing server costs and player's latencies, let's take a closer look at these optimizations.

Interest Management: Traffic Optimization AND Preventing Cheating

Interest Management deals with sending each client only those updates within the Game World, which it needs to render the scene. It is very important for quite a few games out there.

> "Mathematically speaking, without Interest Management, the amount of data on our servers will need to send (to all players combined), is O(N^2). Interest Management reduces this number to O(N).

Let's consider OurRPG mentioned above, and the Publishable State which needs to transfer 50 bytes/network-tick/character. Now let's assume that OurRPG is a big world with 10000 players. Transferring all the data about all the players to all the players would mean transferring 10000characters*50bytes/tick/character*20ticks/second = 10MBytes/second to each player, and 100GBytes/second total (and that's with our Publishable State being reasonably optimal, i.e. without transferring meshes). However, if we notice that out of that 10000 players each given player can see only 20 other players (which is the case most of the time for most of the more-or-less realistic scenes) – then we can implement "Interest Management" and send each player only those updates-which-are-of-interest-to-her (in other words, sending only those things which are needed for rendering). Then, we need to send only 20characters*50bytes/tick/character*20ticks/second = 20KByte/second to each player (200MBytes/second total), MUCH better.

Mathematically speaking, without Interest Management, the amount of data on our servers will need to send (to all players combined), is O(N^2). Interest Management (if properly implemented) reduces this number to O(N). The same thing from a bit different perspective can be stated as

Interest Management normally allows to establish a capping on amount of traffic sent to each player, regardless of total number of players in the game.

In practice, implementations of the Interest Management can vary significantly. In the simplest form, it can be a sending only information of those characters which are currently within certain radius from the target player (or even "send updates only to players within the same "zone"). In more complicated implementations, we can take into account walls etc. between players. The latter approach will also help to address "see-through-walls" cheating.

This also leads us to a second advantage of Interest Management:

Interest Management (if properly implemented) MAY allow you to address "lifting fog-of-war" and "see-through-walls" cheats

The logic here is simple: if the client doesn't receive information on what is going on in "fog-of-war" areas or behind the wall, then no possible hacking of the client will allow to reveal this information, making this kind of attacks pretty much hopeless.

An extreme case of this class of cheats would be for an (incredibly stupid) poker site which has pocket cards data as a part of Publishable State and doesn't implement any Interest Management. It would mean that such an implementation will send pocket cards to all the clients (and then clients won't show other players' cards until the flag *show_all_cards* is sent from the server). DON'T DO THIS – the client will be hacked very soon, with pocket card revealed to cheaters from the very beginning of the hand (ruining the whole game). Interest Management (or even better – excluding pocket cards from Publishable State altogether, with, say, point-to-point delivery of pocket cards) is THE ABSOLUTE MUST for this kind of games. More or less the same stands for quite a few MMORTS out there, where lifting "fog of war" via cheating would give way too much unfair advantage.

"An extreme case of this class of cheats would be for an (incredibly stupid) poker site which has pocket cards data as a part of Publishable State and doesn't implement any Interest Management.

Note that when choosing you Interest Management algorithm, you need to think about worst-case scenarios when a large chunk of your players gather in the same place (what about that wedding ceremony which everybody will want to attend?). This can be really unpleasant, and you do need to think how to handle it well in advance. If going beyond the most obvious (and BTW working pretty well) solution of "we don't have any Big Events, so it won't be a problem" – things may become complicated (and if your game is a 3D one – the same scenarios can easily bring the number of triangles to be rendered on the client-side, beyond any reasonable limits, bringing any graphics card to its knees). One of the ways to deal with it – is to limit the number of transferred-characters to a constant limit (ensuring that O(N) thing), and when this limit is exceeded – to render the rest as a "generic crowd" simulated purely by client-side and wandering by some simple rules (and the same "generic crowd" people can be rendered as really-low-poly models to deal with polygon numbers issue).

Before Compression: Minimizing Data

One thing which needs to be mentioned even before we start to compress our Publishable State, is that most of the time we can (and SHOULD) minimize the amount of data we want to include into our Publishable State. Way too often it happens that we're publishing data field in an exactly the same form as it is available on the Server-Side, and this form is usually redundant, leading to unnecessary data being transferred over the network. A few common minimization rules of thumb:

- DON'T transfer doubles; while double operations are cheap (at least on x86/x64), transferring them is not. In 99% of cases, transferring a float instead won't lead to any noticeable changes.
- DO think about replacing floats with fixed-point numerics (in fact, an integer with an understanding where the point is, or more precisely – what is the multiplier to be used to convert from Server-State data to Publishable State and vice versa)
    - one pretty bad example of float being obviously too much, is transferring angle for an RPG. In most cases, having it transferred as 2-byte fixed-point with lower

7 bits being fraction, will cover all your rendering needs with an ample reserve

- for coordinates, calculations are more complicated, but as long as we need a fixed spatial resolution (and for rendering this is exactly what we need), fixed-point encodings are inherently more efficient than floating-point ones, as we don't need to transfer exponent for fixed-point. In addition, with standard floats it is more difficult to use non-standard number of bits. For example, if we have a 10000m by 10000m RPG world, and want to have positioning with a precision of 1cm, then we need 1e6 possible values for each coordinate. With fixed-point numerics, we can encode each coordinate with 20 bits, for 40 bits (5 bytes) total. With floats, it will take 2*32 bits = 8 bytes (that's while having comparable spatial resolution(!)), or 60% more (and if we'd transfer doubles – it would go up to 16 bytes, over 3x loss compared to fixed-point encoding).
- yet another case for transferring fixed-point numerics is all kinds of currencies (actually, it is cents which are transferred, and the rest is just interpretation)

Compression

Now we have our Publishable State with a proper Interest Management, and want to reduce our traffic further. Let's name those techniques which help us to take whatever-we-want-to-publish (after Interest Management has filtered out whatever is not necessary for the specific client), and to deliver it to the client in an optimized way, "Compression Techniques". Note that we'll interpret "Compression" much broader than usual ZIP or JPEG compression (and it will have quite a few things which are not typically used for compression), but essentially all of "Compression Techniques" are still following exactly the same pattern:

- take some data on the source side of things (server-side in our case)
- "compress" it into some kind of "compressed data"
- transfer the compressed data over the Internet
- "decompress" it back on the receiving side (with or without data loss, see on "lossless" vs "lossy" compression below)
- to get more-or-less-the-same data on the target side of things.

Also let's note that some of the techniques described below, while being well-known, are usually not named "compression"; still, I think naming them "Compression Techniques" (as a kind of "umbrella" term) makes a lot of sense and provides quite useful classification.

To make our compression practical and limited (in particular, to avoid using global states), let's define more strictly what "Compression Techniques" can and cannot do:

- "Compression Techniques" are allowed to keep a buffer (of limited size) of past values on both sides (just like ZIP/LZ77 does)
  - we MAY refer to the buffer (explicitly or implicitly) to reduce the amount of data sent
  - using this buffer creates complications when working over UDP, but there are known ways of handling it which will be discussed in Chapter [[TODO]]

- "Compression Techniques" are allowed to know about the nature of specific fields we're transferring; these specifics can be described, for example, in IDL (see Chapter [[TODO]] for more details)
    - in particular, if we have two fields, one of which is coordinate, and another one is velocity along the same coordinate, this relation MAY be used by our "Compression Technique"
- "Compression Techniques" are allowed to rely on game-specific constants, as long as they're game-wide
    - for example, if we know that for OurRPG the usual pattern when user presses "forward" button, is "linear acceleration of A m/s^2 until speed reaches V, then constant speed" – we ARE allowed to use this knowledge (alongside with A and V constants) to reduce traffic
- "Compression Techniques" are NOT allowed to use anything else. In other words, we won't consider things like client-side-extrapolation-which-takes-into-account-running-into-the-wall, as "Compression" (doing it would require "Compression" to know wall positions, and we want to keep our "Compression" within certain practical limits).
- "Compression Techniques" can be either "lossless" or "lossy". If compression is "lossy", we MUST be able to put some limits on the maximum possible "loss" (for example, if our compression transfers "x" coordinate in a lossy manner, so that client_x MAY differ from server_x, we MUST be able to limit maximum possible (server_x – client_x)). In the sections below, all the compression techniques are lossless unless stated otherwise.

Lossy compression

Lossy compression (irreversible compression) is the class of data encoding methods that uses inexact approximations (or partial data discarding) to represent the content.

— Wikipedia —

Now let's start discussing various flavours of compression.

Delta Compression

Arguably the most well-known compression is so-called "delta compression". Actually, there are two subtly different things known under this name in the context of games.

The first flavour of "delta compression" is about skipping those fields of the game state which exist in the publishable state, but which didn't change since the last update (usually, you're just transferring a bit saying "these field didn't change" instead). This kind of "delta compression" is an extremely common technique (known at the very least since Quake) which is applicable to *any* type of field, whether it is numerical or not. This, in turn, allows publishing such rarely changing things as player's inventories (though see note in "Publishable State" section above about omitting inventory from publishable state completely, or about making it available on demand; while not always possible, this is generally preferable).

VLQ

A variable-length quantity

The second flavour of "delta compression" is a close cousin of the first one, but is still a bit different. The idea here is to deal with situations when a *numerical* field changes (so skipping the field completely is not really an option), but instead of transferring new

value, to transfer a difference between "new value" and "old value" (pretty much like (A)DPCM is doing for audio signals). For example, if the field is an x coordinate, and has had an "old value" of 293.87, "new value" is likely to be "293.88", and is unlikely to be 0, so spectrum of differences becomes strongly skewed towards values with smaller absolute value, that enables further optimizations. The gain here can be obtained by either simply using less bits to encode the difference, or to play around with variable-length encodings such as VLQ, or to rely on running another layer of compression (such as Huffman compression, see "Classical Compression" subsection below) which will generally encode more-frequently-occurring-bytes (in this case – zeros) with less bits.

(VLQ) is a universal code that uses an arbitrary number of binary octets (eight-bit bytes) to represent an arbitrarily large integer.

— Wikipedia —

Let's note that the second flavour of "delta compression" can also be made "lossy": we MAY round the delta transferred, as long as we're sure that pre-defined loss limits are not exceeded. Note that ensuring of loss limits usually requires server to keep track of the current value on the client side, so that rounding errors, while accumulating, still remain below the loss limit (and are corrected when the limit is about to be exceeded).

Dead Reckoning as Compression

Dead reckoning

In navigation, dead reckoning or dead-reckoning (also ded for deduced reckoning or DR) is the process of calculating one's current position by using a previously determined position, or fix, and advancing that position based upon known or estimated speeds over

Another big chunk of simulation-related "Compression Techniques" is known as "dead reckoning". Note that despite obvious similarities, use of "dead reckoning" for the purposes of compression is subtly different from it's use for client-side extrapolation (see "Client-Side Extrapolation a.k.a. Dead Reckoning" section above)[6] When using dead reckoning for client-side extrapolation purposes we're trying to deal with latency: we don't have information on the client-side (yet), and trying to predict the movement instead, reducing perceivable latency; to do it, no server-side processing is required, and there is no precision loss. When using dead reckoning for compression purposes, we do know exact movement, and know on the server side how exactly the client will behave, so we can use this knowledge as a Compression Technique to reduce traffic (normally – as a "lossy" compression); for compression purposes, we do need server-side processing and "data loss" threshold.

The idea with a classical "dead reckoning" is to use velocities to "predict" the next value of the coordinate, while putting a limit on maximum deviation of the server-side coordinate from the client-side coordinate, so from "Compression" point of view it is a "lossy" technique with a pre-defined limit on data loss.[7]

Let's consider an example. Let's say that we have tuple (x,vx) as a part of our publishable state, and that at a certain moment client has it as (x0,vx0), and that server knows this (x0,vx0) for this specific client.[8] Now, an update comes in to the server side, which needs to make it (x1,vx1). Server calculates (x0+vx0,vx0) as a "predicted" state, and sees

elapsed time and course
— Wikipedia —

if it is "too different" from (x1,vx1).[9] If it is not too different – server can skip sending any update for this coordinate (and if it is too different – the second flavour of "delta compression" can be used to send a message fixing the difference).

For further discussion of the classical "dead reckoning" as compression (with a discussion of associated visual effects), see, for example, [Gamasutra.DeadReckoning].[10]

One last note – not only coordinates can be compressed using dead reckoning-like compression; actually, pretty much anything which can be predicted with high probability, can benefit from it. One practical example of such non-coordinates compressable by dead reckoning, is animation frame number (that is, if you need to transfer it).

"Not only coordinates can be compressed using dead reckoning-like compression; actually, pretty much anything which can be predicted with high probability, can benefit from it.

---

[6] in literature, it is usually considered to be one single "Dead Reckoning" algorithm (part of "DIS" a.k.a. IEEE1278) which reduces both perceivable latency and traffic. However, due to differences in both the effects and implementation, I prefer to consider these two uses of Dead Reckoning separately
[7] while it can be made lossless, it won't get much in terms of compression, so the lossless variation is almost-never used
[8] as noted above, when using UDP, this is tricky, but doable, see Chapter [[TODO]] for further details
[9] "too different" here is the same as "exceeding pre-defined loss limit"
[10] despite the title, most of the discussion within is not about latency, but about reducing traffic with a pre-defined threshold, which we refer to as one of "Compression Techniques"

Dead Reckoning as Compression: Variations

"Dead reckoning" as described above, is certainly not the only way to use kinematic equations to optimize traffic. Possible variations include such things as:

- using "delta compression" (the second variety described above) to encode data when the "loss limit" is exceeded
- using accelerations in addition to velocities (and predicting velocities based on accelerations)
- calculating velocities/accelerations (using previous values in the buffer) instead of transferring them
- use of smoothing algorithms to avoid sharp change of coordinates when the correction is issued. These are similar to the smoothing algorithms used for server reconciliation (see "Running into the Wall, and Server Reconciliation" section above), and the same smoothing algorithm can be used for both purposes. Whether to consider smoothing a part of compression (or a post-compression handling) – is not that important and it depends.
- using knowledge about the game mechanics to reduce traffic further.

- As one example, if in OurRPG velocity of PC always grows in a linear manner with fixed acceleration until it reaches a well-defined limit – this can be used to calculate "predicted speed" and to avoid sending updates along this typical pattern.

Classical Compression

LZ77

LZ77 is the lossless data compression algorithm published by Abraham Lempel and Jacob Ziv in 1977.

— Wikipedia —

Classical lossless compression (such as ZIP/deflate) usually uses two rather basic algorithms. The first one usually revolves around LZ77[11] (with the idea being to find similar stuff in the earlier buffer and to transfer a reference instead of verbatim stream). The second algorithm is usually related to so-called Huffman coding,[12] with the idea being to find out what symbols occur in the stream more frequently than the others, and to use less bits to encode these more-frequently-used symbols. Of course, there are lots of further variations around these techniques, but the idea stays pretty much the same. ZIP's deflate is basically a combination of LZ77 and Huffman.

Unfortunately, classical compression algorithms, such as deflate, are not well-suited for game-related compression. One of the reasons behind is that (as it was shown for deflate in [DrDobbs.OnlineCompression], these algorithms are usually not optimized to handle small updates (in other words, "flush" operation, which is required to send an update, is expensive for ZIP and other traditional stream-oriented algorithms).

On the other hand, it is possible to have a compression algorithm optimized for small updates; one example of such an algorithm is an "LZHL" algorithm in the very same [DrDobbs.OnlineCompression] by my esteemed translator. Like deflate, it is a combination of LZ77-like and Huffman-like compression, unlike deflate, it is optimized for small updates.[13]

"It is possible to have a compression algorithm optimized for small updates; one example of such an algorithm is an "LZHL" algorithm

If nothing else, you can always try to use Huffman (or Huffman-like, as described in [DrDobbs.OnlineCompression]) coding for your packets. I won't go into too much details of Huffman algorithm as such here (it is described very well in [Wiki.Huffman]), but one trick which may help here with regards to games, is the following. Usually, implementations of Huffman algorithm transfer "character frequency tables" as a part of compressed data; this leads to the complications in case of lost packets (or, if you transfer the table for each packet, they will become huge). For games, it is often possible to pre-calculate character frequency table (for example, by gathering statistics in a real game session) and to hardcode this frequency table both into the server and into the client. In this case, lost packets won't affect frequency tables at all, and this variation of Huffman will work trivially over both TCP and UDP. Note though that usually gains from Huffman are rather limited (even if your data has lots of redundancies, don't expect to gain more than 20% compression from pure Huffman), but it is usually better than nothing.

[11] and its close cousins such as LZ78 and LZW
[12] or a bit more efficient but much slower arithmetic coding
[13] note that LZHL as such won't work directly over UDP, and some significant adaptation will be necessary to make it work there; for TCP and TCP-like streams, however, it has been seen to work very well

Combining Different Compression Mechanisms and Law of Diminishing Returns

It is perfectly possible to use different compression mechanisms together. For example:

- for relatively static data (such as inventory), delta compression (1st variation), followed by classical compression, can be used
- for very dynamic coordinate-like data – dead reckoning (as a lossy compression), with dead reckoning using delta compression (2nd variation), using VLQ to encode differences, can be used

Note that the examples above are just that – examples, and optimal case for your game may vary greatly.

One further thing to note when combining different compression mechanisms, is that all of them are merely reducing redundancy in your data, so even if they're not conflicting directly,[14] traffic reduction from applying two of them simultaneously, will almost universally be less than the sum of reductions from each of them separately. In other words, if one compression gives you 20% traffic reduction and another one – another 20%, don't expect two of them combined to give you 20%+20%=40% or 1-(0.8*0.8)=36% reduction – most likely, it will be less than that 🙁.[15]

[14] examples of such direct conflicts would be trying to use dead reckoning *after* classical compression, or using LZ77 compression *after* Huffman compression
[15] while there are known synergies between different compression algorithms, notably for LZ77 *followed by* Huffman, they're very few and far between

Traffic Optimization: Recommendations

When speaking about optimizing traffic, I usually recommend the following order of doing it:

- minimize your Server-Side State. It is important not only to minimize traffic, but also to minimize server-side CPU load

- minimize your Publishable State. Be aggressive: throw away everything, and add fields to your Publishable State only when you cannot live without them
- split your Publishable State into several groups with different timing requirements
- make sure to use “Delta Compression” (the first variation above) to allow skipping updates for non-changing objects
  - treat “non-changing objects” broadly; for example, for many games out there an object which keeps moving with the same speed in the same direction, can be treated as “non-changing” (alternatively, you can handle it via “dead reckoning”)
- think about “Dead Reckoning” compression, keeping adverse visual effects in check (and reducing threshold if necessary)
  - don’t forget about variations, they may make significant difference depending on specifics of your game
- think about running Classical Compression on top of the data compressed by previous techniques, but don’t hold your breath over it
  - deflate as such won’t work for most of the games (due to the cost of “flush”, see above)
  - LZHL works for TCP, but adapting it for UDP will require an additional effort (and will hurt efficiency too)
  - Huffman with pre-populated frequency tables (see above) will work for UDP, but the gains are limited
- when combining different compression techniques, keep in mind that their order is very important
- I strongly suggest to separate all types of compression from the rest of your code (including simulation code)
  - moreover, I strongly suggest to say that compression code should be generated by your IDL compiler based on specifications in IDL, instead of writing compression ad-hoc. More on IDL in Chapter [[TODO]].

“Minimize your Publishable State. Be aggressive: throw away everything, and add fields to your Publishable State only when you cannot live without them

[[To Be Continued…

This concludes beta Chapter VII(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 VII(c), “Point-to-Point Communications”]]

[–] References

[GafferOnGames.SnapshotsAndInterpolation] Glenn Fiedler, “Snapshots and interpolation”, Gaffer on Games
[Gamasutra.DeadReckoning] Jesse Aronson, “Dead Reckoning: Latency Hiding for

Networked Games", Gamasutra
[DrDobbs.OnlineCompression] Sergey Ignatchenko, "An Algorithm for Online Data Compression"
[Wiki.Huffman] "Huffman coding", Wikipedia

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG. RTT, Input Lag, and How to Mitigate Them*

MMOG. Point-to-Point Communications and non-blocking RPCs »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, compression, game, multi-player, network, protocol, server

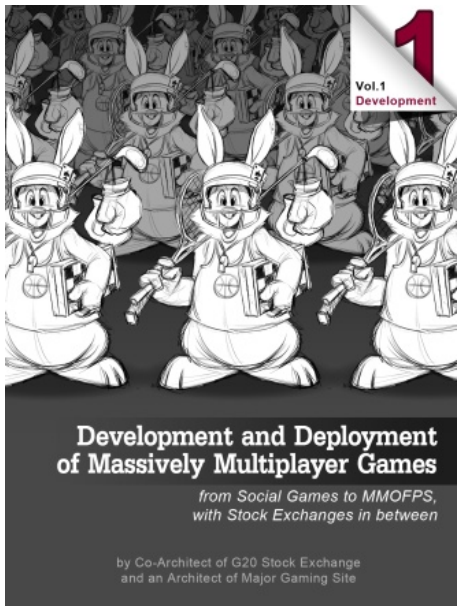
Copyright © 2014-2016 ITHare.com

MMOG. Point-to-Point Communications and non-blocking RPCs

posted February 8, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

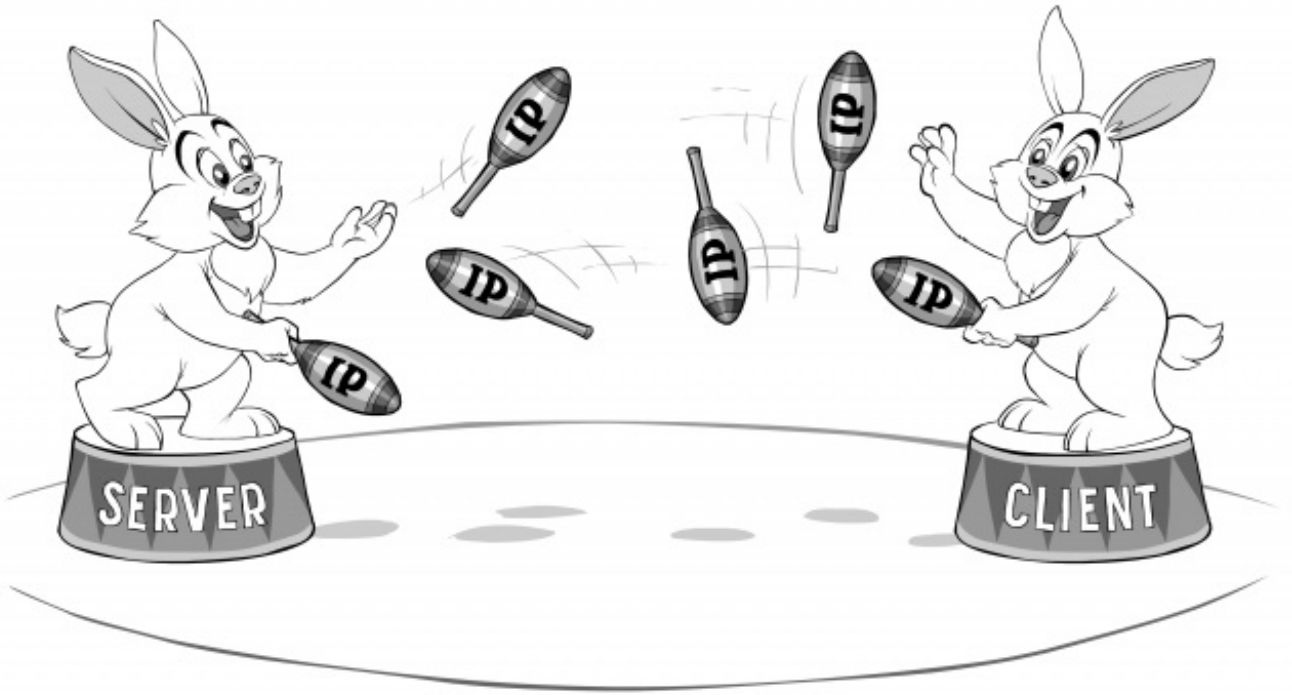
[[*This is Chapter VII(c) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further 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.]]



After we've discussed Publishable State, the next thing we'll need for our MMO is Point-to-Point communications. While Publishable State is mostly about servers communicating with clients, Point-to-Point communications can happen either between client and server, or between two servers. These two types of Point-to-Point communications have quite a bit in common, but there are also substantial differences.

Note that differences between TCP and UDP are still beyond the scope until Chapter [[TODO]]; for now we're speaking of *what we need,* and not about *how to implement it.*



RPCs

Regardless of the nature of Point-to-Point communications (whether it's being between client and server, or between two servers), they share certain common properties.

In particular, it is common for games to implement point-to-point communications as non-blocking Remote Procedure Calls (RPCs). While this is not required (and you can use simple message exchange instead – with either hand-written or IDL-based marshalling), non-blocking RPCs tend to speed up development significantly.

It should be noted, however, that while non-blocking RPC are perfectly viable for games, you Really SHOULD keep away from blocking RPC (as in DCE RPC/COM/CORBA)

The reason for it is the following. With games, you SHOULD use event-driven/FSM programming (if I didn't manage to convince you about it in Chapter V, just trust most of game developers out there, and take a note of most of them using FSMs at least to some extent; in particular, classical game loop and simulation loop are FSMs). And with event-driven FSMs, any blocking operation (especially the one which involves waiting for remote entity) is a Big No-No.

Implementing Non-Blocking RPCs

To implement non-blocking RPCs, you need a way to specify signatures of your remotely-callable functions; such specification defines the interface (and often protocol, though see more on encodings in [[TODO!]] section below) between RPC caller and RPC callee. Sometimes (like in Unity), it is done by adding certain attributes ([RPC]/[ClientRpc]/[Command] method attributes in Unity) to existing functions/methods.

However, usually I prefer to have my own explicit IDL (with an IDL compiler) instead. The reason for this preference for a separate IDL is that whenever we specify RPC signatures right in the code, it means that having them in the code-written-in-a-different-language, we'll need at least to specify them once again in the second language (what makes code maintenance extremely error-prone).[1]

We'll discuss implementation of your own IDL in the [[TODO]] section, but for the purposes of our current discussion it doesn't really matter whether we're using intra-language RPC specifications (like in Unity), or our own external IDL (as we'll discuss below).

IDL

An interface description language or interface definition language (IDL), is a specification language used to describe a software component's application programming interface (API).

— Wikipedia —

---

[1] in theory, you could use one language as an IDL for another one, but I haven't seen such things (yet?)

Specifics of Non-blocking RPCs

Non-blocking RPCs have some peculiarities, both for implementing them, and for using them. In general, there are two cases for non-blocking RPCs.

The first case is a non-blocking RPC, which returns void (and can't throw any exceptions). For such void RPCs, everything is simple – caller just marshals RPC parameters, and sends a message to the callee, and the callee unmarshals it and executes RPC call, that's about it. From all the points of view (except for pure syntax), calling such an RPC is the same as sending a message (with all the differences being of purely syntactic nature).

A typical example of such an RPC (as defined in an IDL) is something along the following lines:

```
STRUCT Input {
  bool left;
```

```
  bool top;
  bool right;
  bool bottom;
  bool shift;
  bool ctrl;
};

void move_me(Input in);
```

Non-void RPCs

The second (and much more complicated) case for RPCs is an RPC which either returns a value, or is allowed to throw an exception (or both). An example IDL for such a non-void RPC is the one from Chapter VI:

```
int dbGetAccountBalance(int user_id);
```

"Non-void RPCs are significantly more complicated to implement, and most of the popular game engines out there do NOT support them

These non-void RPCs are significantly more complicated to implement, and most of the popular game engines out there do NOT support them (see Chapter [[TODO]] for more information about Unity/Photon and Unreal Engine).

The main issue with implementing non-void RPCs is for the caller to specify what to do when the function returns (or throws an exception). There are many ways of doing it, they were discussed in Chapter VI, section "Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between)" (with FSMFutures being my personal favorite at the moment). On the other hand, while implementing them is difficult, once they are available, they do simplify development significantly, so you will want to use them if your engine supports them.

Whenever your engine doesn't support non-void-RPCs, you'll usually need to make another RPC call in the opposite direction when you're done

In this case, our last example will need to be rewritten along the following lines:

```
//Game World Server to DB Server:
void dbGetAccountBalance(FSMID where_to_reply, int user_id);

//DB Server to Game World Server:
```

```
void gameWorldGotAccountBalance(int user_id, int balance);
```

or in more general manner:

```
//Game World Server to DB Server:
void dbGetAccountBalance(FSMID where_to_reply, int request_id, int user_id);

//DB Server to Game World Server:
void gameWorldGotAccountBalance(int request_id, int balance);
```

While this will work, it is quite cumbersome and inconvenient (substantially worse than even Take 2 from Chapter VI).[[TODO! add these RPC to Chapter VI as "Take 1a"]]

Same-thread operation

> "In other words, you can write your code 'as if' all-your-code-within-the-same-FSM executed within the same thread

Another thing to understand about non-blocking RPCs is that due to non-blocking nature, other things can happen within the same FSM while the RPC is executed. This can be seen as either blessing (as it allows for essentially parallel execution while staying away from any thread synchronization), or a curse (as it complicates understanding), but needs to be kept in mind at all the times you are dealing with non-blocking RPCs. One positive thing to note in this regard is that for most sane implementations, and regardless of using any of the ways to report back described in Chapter VI, you don't need to care about thread synchronization (as all the callbacks/lambdas/futures will be called in the context of the same thread). In other words, you can write your code "as if" all-your-code-within-the-same-FSM executed within the same thread (and whether it will be actually the same thread or not, is not that important); from a bit different perspective, you can think of all the callbacks "as if" they're essentially the same as co-routines (but using a different syntax).

Client-to-Server and Server-to-Client Point-to-Point communications

Now, as we've discussed the similarities between point-to-point communications, we need to describe differences. And arguably the most important difference between Client-to-Server and Server-to-Server communications, is related to disconnects. As a rule of thumb, for Server-to-Server communications the disconnects are extremely rare, and all the disconnects are transient (that is, unless your whole site is down). It means that we can expect that they are restored really quickly, which in turn means that we can try to hide temporary loss of connectivity from application layer. On the other hand, for Client-to-Server (and

Server-to-Client) communications, this "restored really quickly" observation doesn't stand, and dealing with disconnects becomes an important part of application logic.

Let's speak about Client-to-Server and Server-to-Client communications first.

Inputs

One thing which you'll inevitably need to transfer from client to server, is player inputs. For a non-simulation game (think blackjack, stock exchange, or social game), everything is simple: you've got an input – you're sending it to the server right away.

For simulation games, however, it is not that trivial. Traditionally, simulation-based games usually operate in terms of "simulation ticks", and usually single-player games are just polling the state of keyboard/mouse/controller on each tick. As a result, when moving from a single-player simulation game to the network one, it is rather common to mimic this behaviour just by client sending state of (keyboard+mouse+controller) to the server on each tick (which becomes a "network tick"). An alternative (also pretty common) approach would send only *changes* to this (keyboard+mouse+controller) state; this can be done either as soon as the state is changed, or again on "tick".

"However, as soon as we realize that packets can be lost, handling inputs becomes a bit different.

As long as there are no disconnects (nor packet loss), there is no that much difference between these approaches. However, as soon as we realize that packets can be lost, handling inputs becomes a bit different.

If we're transferring state of player's input devices on each tick, then in case of lost packet[2] PC will effectively stop on the server-side; moreover, at the same time, if we implement Client-Side Prediction, it will be running on the client side.

On the other hand, if we're transferring only *changes* to keyboard/mouse/controller state, then in case of packets being lost, our PC will keep running for some time (until we detect disconnect) even if player has already released the button; this may potentially lead to PC running off the cliff even if the player's actions didn't cause it ☹ (just by disconnect happened at an unfortunate time).

A kind of "hybrid" approach is possible if we're using client-to-server acknowledgment packets (which will arise in a pretty much any game world state publishing schema, see Chapter [[TODO]] for further discussion) to distinguish between "player is still keeping the button pressed" and "we have no idea, as the

packet got lost" situations. In other words, if an acknowledgment arrived, but without any information about the keyboard state change – then we know *for sure* what is going on on the client side,[3] if there is no acknowledgment – then something is wrong, so our server can stop PC before he runs off the cliff.

Overall, there is no one universal answer to these questions, so you'll basically need to pick one schema, try it, and see if it works and feels fine for your purposes in case of pretty bad connections.

---

[2] that is, beyond capabilities of input buffer [[TODO!: add input buffer to Fig VII.2/VII.3]]
[3] and if keyboard state change has happened, it can and SHOULD be combined with the acknowledgment IP packet to save on bandwidth, but this is a bit different story, discussed in Chapter [[TODO]]

Input Timestamping

One issue which is often associated with inputs, is client-side input timestamp (in practice, usually it will be a tick-stamp). This is indeed necessary to facilitate things such as Lag Compensation described in "Lag Compensation" subsection above. On the other hand, as soon as server starts to trust this timestamp, this trust (just as about any kind of trust out there) can be abused. For example, if within your game you have a Good-Bad-Ugly-style shootout, and compensate for the lag, then the Bad guy, while having worse reaction, could compensate for it by sending "shoot" input packet with an input timestamp which is 50ms earlier than the real time, essentially gaining an unfair advantage for these 50ms. In general, such cheating (regardless of way of implementing it[4]) is a fundamental problem of *any* kind of lag compensation, so you should be really sure how to handle various abuse scenarios before you introduce it.

"For example, you have a Good-Bad-Ugly-style shootout, and compensate for the lag, then the Bad guy, while having worse reaction, could compensate for it by sending "shoot" input packet with an input timestamp which is 50ms earlier than the

---

[4] no, measuring pings instead of relying on input timestamps doesn't prevent the cheat, it just makes the cheat a bit more complicated

"Macroscopic" Client Actions

In addition to sending bare input to server, client usually needs to implement some actions which go beyond it.

Examples of such “macroscopic” actions include such sequences of inputs as:

real time, essentially gaining an unfair advantage for these 50ms.

- player looking at object (usually processed purely on client-side)
- client showing HUD saying that “Open” operation is available because object under the cursor is container (again, processed purely on the client-side)
- client pressing “Action” button (which means “Open” in this context)
- client showing container inventory (obtained via an RPC call, or taken from Publishable State)
- player choosing what to take out
- only then client invoking a Client-to-Server RPC such as take_from_container(item_id, container_id)

For such RPC calls as take_from_container(), disconnect during the call can be simply ignored in most cases (so that player will need to press a button again when/if the connection is restored)

Another set of “macroscopic” actions (usually having even longer chains of events before RPC call is issued) is related to dialog-based client-side interactions such as in-game purchases. In these cases, all the interactions (except, maybe, for some requests for information from the server) usually stay on the client-side until the player decides to proceed with the purchase; when this happens, Client-to-Server RPC call containing all the information necessary to proceed with the purchase, is issued.

For such RPC calls, handling of disconnect during an RPC call is not that obvious. If you want to be player-friendly (and usually you should be), you need to consider two scenarios. The first one is when the disconnect is transient, and client is able to reconnect soon; then, you need a mechanism to detect whether your RPC call has reached the server, to get the result if it did, and to re-issue the call if it didn’t; this would allow to make disconnect look really transient for the player, and to show the result of the purchase as if the disconnect has never occurred. To implement it, you’ll need to implement both re-sending of RPC call on the client side, and dealing with duplicates on the server-side, in a manner similar to the one described in “Server-to-Server” Communications section below.

The second scenario occurs when the RPC call is interrupted by disconnect before obtaining the reply, and disconnect takes that long that client gets closed (or server gets restarted). In this case, the only things we can practically do for the player, are not directly related to the communication protocols (but they still need to be done). Two most common features that help to make player not *that* unhappy in this second scenario, are (a) to send her an e-mail if the "purchase" RPC call has reached the server (it doesn't help to vent frustration if the call didn't reach the server ☹), and (b) to provide her with a way to see the list of all her purchases from the client when she's back online (which we need to do anyway if we want to be player-friendly).

"The second scenario occurs when the RPC call is interrupted by disconnect before obtaining the reply, and disconnect takes that long that client gets closed (or server gets restarted).

Server-to-Client

While server does send a lot of information to client (both as a part of Publishable State, and as replies to Client-to-Server RPC calls), it is not too common to call RPC from the server side.[[TODO!: add note to Chapter VI/"Asynchronous" that it is not too common to do it this way, and that it is usually client-side-driven rather than server-side-driven]]

On the other hand, in some cases such RPC calls (especially void RPC calls without the need to process reply on the server side) are helpful. One such example is passing pocket cards to the client in a poker game. This will allow to exclude pocket cards from Publishable State (which in absence of Interest Management allows for rampant cheating, as was described in "Interest Management: Traffic Optimization AND Preventing Cheating" section above).

Server-to-Server Communications

As noted above, from the point of application layer Server-to-Server communications can be made seamless (hiding disconnects, including those resulting from FSM relocations, from application layer). However, this comes at the cost of infrastructure level doing this work behind the scene. One fairly common protocol which does achieve seamless handling of disconnects, implements two related but distinct features.

First, as noted above, we'll be usually dealing with "non-blocking RPC calls" anyway. To support some kind of callback (whether being OO, lambda, or future), we'll need to keep a list of "outstanding RPC requests" (with their respective IDs) on the caller side anyway. And as soon as we have this list of "outstanding RPC calls", we have sufficient information to re-send RPC request in case of lost packet/disconnect.[5]

Idempotence

Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.
— Wikipedia —

On the other hand, this technique, while guaranteeing that we will get *at least one* RPC request on the callee side for each RPC call on the caller side, doesn't guarantee that it will be *the only one*. In other words, if implementing only the logic described above, duplicate RPC calls on callee side can happen for a single RPC call on the caller side. While making all the RPC calls *idempotent* would solve this problem, in practice making sure that each and every call is idempotent at the application layer, is not exactly realistic.

That's why a second part of processing (this time – on the callee side) needs to be added. For example, we can make the callee side keep the list of "recently-processed RPC request_ids" (with associated replies), and if some request with an ID from this list comes in – we should just to provide the associated reply without calling anything on the callee side. This scenario may legitimately happen if the connection was lost-and-restored after the request was received, but before the reply was acknowledged, but the handling mentioned above, guarantees that everything is handled "as if" disconnect has never happened.

As soon as we have these two parts of processing (in practice, it will be a bit more complicated, as information on "which replies can be dropped from the list" will need to be communicated too, plus, most likely, we'll need to implement handshakes to distinguish between new connection and the broken one) – we can say that our Server-to-Server communication is tolerant to all kinds of transient inter-server disconnects. This is necessary not only to deal with inter-server disconnects at TCP level (which are extremely rare in practice), but is also one of prerequisites to deal with scenarios when we're restoring/moving an FSM (see Chapter VI, section "Failure Modes and Effects" for details).

"As soon as we have these two parts of processing – we can say that our Server-to-Server communication is tolerant to all kinds of transient inter-server disconnects.

An alternative (similar, but not identical) way of dealing with such transient-disconnect issues, is to create two "guaranteed delivery" message streams (going into opposite directions), with each of the streams keeping its own list of "unacknowledged messages", and re-sending them on loss-and-restore of underlying connection; on the receiving side, a simple "last ID processed" is sufficient[6] to filter out all the duplicates.

[5] as noted in Chapter VI, section “On Inter-Server Communications”, we’ll probably use TCP for inter-server communications anyway, so such re-send will need to happen only on TCP disconnect/reconnect
[6] that is, assuming that message IDs are guaranteed to be monotonous

[[To Be Continued…

This concludes beta Chapter VII(c) 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 VII(d), “IDL: Encodings, Mappings, and Backward Compatibility”]]

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *MMOG: World States and Reducing Traffic*

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, game, multi-player, network, protocol, RPC, server

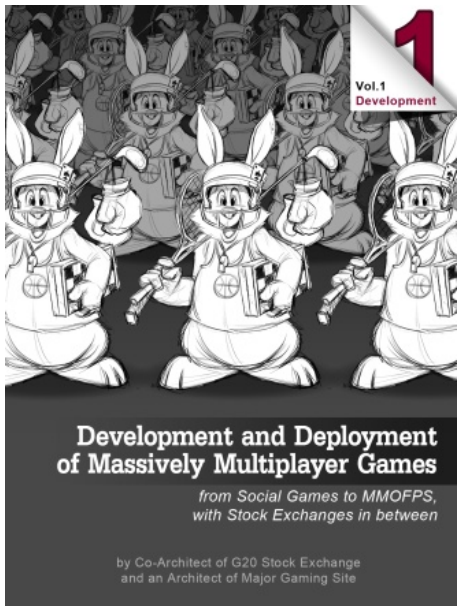
Copyright © 2014-2016 ITHare.com

IDL: Encodings, Mappings, and Backward Compatibility

posted February 15, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter VII(d) 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.]]



As we've discussed those high-level protocols we need, I mentioned Interface Definition Language (IDL) quite a few times. Now it is time to take a closer look at it.

Motivation for having IDL is simple. While manual marshalling is possible, it is a damn error-prone (you need to keep it in sync at least at two different places – to marshal and to unmarshal), not to mention too inconvenient and too limiting for further optimizations. In fact, the benefits of IDL for communication were realized at least 30 years ago, which has lead to development of ASN.1 in 1984 (and in 1993 – to DCE RPC).

These days in game engines, quite often a (kinda) IDL is a part of the language/engine itself; examples include [RPC]/[Command]/[SyncVar] tags in Unity 5, or UFUNCTION(Server)/UFUNCTION(Client) declarations in Unreal Engine 4. However, for most game and game-like communications I still prefer to have my own IDL. The reason for it is two-fold: first, standalone IDL is inherently better suited for cross-language use, and second, none of in-language IDLs I know are flexible enough to provide reasonably efficient compression for games; in particular, per-field Encodings specifications described below are not possible[1]

> **"However, for most game and game-like communications I still prefer to have my own IDL.**

[1] and even if Encodings (along the lines described below) are implemented as a part of your programming language, they would make it way too cumbersome to read and maintain

IDL Development Flow

With a standalone IDL (i.e. IDL which is not a part of your programming language), development flow (almost?) universally goes as follows:

- you write your interface specification in your IDL

    - IDL does NOT contain any implementation, just function/structure declarations

- you compile this IDL (using IDL compiler) into stub functions/structures in your programming language (or languages)

- for callee – you implement callee-side stub functions in your programming language

- for caller – you call the caller-side stub functions (again in your programming language). Note that programming language for the caller may differ from the programming language for callee

One important rule to remember when using IDLs is that

Never Ever make manual modifications to the code generated by IDL compiler.

"Modifying generated code usually qualifies as a Really Bad Idea

Modifying generated code will prevent you from modifying the IDL itself (ouch), and usually qualifies as a Really Bad Idea. If you feel such a need to modify your generated code, it means one of two things. Either your IDL declarations are not as you want them (then you should modify your IDL and re-compile it), or your IDL compiler doesn't do what you want (then you need to modify your IDL compiler).

Developing your own IDL compiler

Usually I prefer to develop my own IDL compiler. From my experience, costs of such development (which are of the order of several man-weeks provided that you're not trying to be overly generic) are more than covered with additional flexibility (and ability to change things when you need) it brings to the project.

With your own IDL compiler:

- whenever you feel the need to change marshalling to a more efficient one (without any changes to the caller/callee code) – no problem, you can do it
- whenever you need to introduce an IDL attribute to say that this specific parameter (or struct member) should be compressed in a different manner[2] (again, without any changes to the code) – no problem, you can add it
- whenever you want to add support for another programming language – no problem, you can do it

- you can easily have ways to specify the technique to extend interfaces (so that extended interfaces stay 100% backwards-compatible with existing calls/callees), and to have you IDL compiler check whether your two versions of the IDL *guarantee* that the extended interface is 100% backwards-compatible. While techniques to keep backward compatibility are known for some of the IDLs out there (in particular, for ASN.1 and for Google Protocol Buffers), the feature of comparing two versions of IDL for compatibility, is missing from all the IDL compilers I know**[[IF YOU KNOW AN IDL COMPILER WHICH HAS AN OPTION TO COMPARE TWO VERSIONS OF IDL FOR BACKWARD COMPATIBILITY – PLEASE LET ME KNOW]]**

Now to the queston "how to write your own IDL compiler". Very briefly, the most obvious and straightforward way is the following:

- write down declarations you need (for example, as a BNF). To start with your IDL, you usually need only two things:
  - declaring structures
  - declaring RPCs
  - in the future, you will probably want more than that (collections being the most obvious example); on the other hand, you'll easily see it when it comes 🙂
- then, you can re-write your BNF into YACC syntax
- then, you should be able to write the code to generate Abstract Syntax Tree (AST) within YACC/Lex (see the discussion on YACC/Lex in Chapter VI).
- As soon as you have your AST, you can easily generate whatever-stubs-you-want.

AST

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language.

— Wikipedia —

---

[2] see section "Publishable State: Delivery, Updates, Interest Management, and Compression" above for discussion of different compression types

IDL + Encoding + Mapping

Now, let's take a look at the features which we want our IDL to have. First of all, we want our IDL to specify protocol that goes over the network. Second, we want to have our IDL compiler to generate code in our programming language, so we can use those generated functions and structures in our code, with marshalling for them already generated.

When looking at existing IDLs, we'll see that there is usually one single IDL which defines both these things. However, for a complicated distributed system such as an MMO, I suggest to have it separated into three separate files to have a clean separation of concerns, which simplifies things in the long run.

The first file is the IDL itself. This is the only file which is strictly required. Other two files (Encoding and Mapping) should be optional on per-struct-or-function basis, with IDL compiler using reasonable defaults if they're not specified. The idea here is to specify only IDL to start working, but to have an ability to specify better-than-default encodings and mappings when/if they become necessary. We'll see an example of it a bit later.

ASN.1

Abstract Syntax Notation One (ASN.1) is a standard and notation that describes rules and structures for representing, encoding, transmitting, and decoding data in telecommunications and computer networking.

— Wikipedia —

The second file ("Encoding") is a set of additional declarations for the IDL, which allows to define Encodings (and IDL+Encodings effectively define over-the-wire protocol). In some sense, IDL itself is similar to ASN.1 language as such, and IDL encodings are similar to ASN.1 "Encoding Rules". IDL defines *what* we're going to communicate, and Encodings define *how* we're going to communicate this data. On the other hand, unlike ASN.1 "Encoding Rules", our Encodings are more flexible and allow to specify per-field encoding if necessary.

Among other things, having Encoding separate from IDL allows to have different encodings for the same IDL; this may be handy when, for example, the same structure is sent both to the client and between the servers (as optimal encodings may differ for Server-to-Client and Server-to-Server communications; the former is usually all about bandwidth, but for the latter CPU costs may play more significant role, as intra-datacenter bandwidth usually comes for free until you're overloading the Ethernet port, which is not that easy these days).

The third file ("Mapping") is another set of additional declarations, which define what kind of code we want to generate to use for our programming language. The thing here is that the same IDL data can be "mapped" into different data types; moreover, there is no one single "best mapping", so it all depends on your needs at the point where you're going to use it (we'll see examples of it below). Changing "Mapping" does NOT change the protocol, so it can be safely changed without affecting anybody else.

In the extreme case, "Mapping" file can be a file in your target programming language.

Example: IDL

While all that theoretical discussion about IDL, Encodings, and Mappings is interesting, let's bring it a bit down to earth.

Let's consider a rather simple IDL example. Note that this is just an example structure in the very example IDL; syntax of your IDL may vary very significantly (and in fact, as argued in "Developing your own IDL compiler" section above, you generally SHOULD develop your own IDL compiler – that is, at least until somebody makes an effort and does a good job in this regard for you):

```
1  PUBLISHABLE_STRUCT Character {
2    UINT16 character_id;
3    NUMERIC[-10000,10000] x;//for our example IDL compiler, notation [a,b] means
4                            //  "from a to b inclusive"
5                            //our Game World has size of 20000x20000m
6    NUMERIC[-10000,10000] y;
7    NUMERIC[-100.,100.] z;//Z coordinate is just +- 100m
8    NUMERIC[-10.,10.] vx;
9    NUMERIC[-10.,10.] vy;
10   NUMERIC[-10.,10.] vz;
11   NUMERIC[0,360) angle;//where our Character is facing
12                        //notation [a,b) means "from a inclusive to b exclusive"
13   enum Animation {Standing=0,Walking=1, Running=2} anim;
14   INT[0,120) animation_frame;//120 is 2 seconds of animation at 60fps
15
16   SEQUENCE<Item> inventory;//Item is another PUBLISHABLE_STRUCT
17                            // defined elsewhere
18 };
```

This IDL declares *what* we're going to communicate – a structure with current state of our Character.[3]

---

[3] yes, I remember that I've advised to separate inventory from frequently-updated data in "Publishable State" section, but for the purposes of this example, let's keep them together

Example: Mapping

Now let's see how we want to map our IDL to our programming language. Let's note that mappings of the same IDL MAY differ for different communication parties (such as Client and Server). For example, mapping for our data above MAY look as follows for the Client:

```
MAPPING(“CPP”,“Client”) PUBLISHABLE_STRUCT Character {
  UINT16 character_id;//can be omitted, as default mapping
                      //  for UINT16 is UINT16
  double x;//all 'double' declarations can be omitted too
  double y;
  double z;
  double vx;
  double vy;
  double vz;
  float angle;//this is the only Encoding specification in this fragment
              //  which makes any difference compared to defaults
              // if we want angle to be double, we can omit it too
  enum Animation {Standing=0,Walking=1, Running=2} anim;
              //can be omitted too
  UINT8 animation_frame;//can be omitted, as
                        //  UINT8 is a default mapping for INT[0,120)

  vector<Item> inventory;//can be also omitted,
                         //  as default mapping for SEQUENCE<Item>
                         //  is vector<Item>
};
```

In this case, IDL-generated C++ struct may look as follows:

```
struct Character {
  UINT16 character_id;
  double x;
  double y;
  double z;
  double vx;
  double vy;
  double vz;
  float angle;
  enum Animation {Standing=0,Walking=1, Running=2} anim;
  UINT8 animation_frame;
  vector<Item> inventory;

  void idl_serialize(int serialization_type,OurOutStream& os);
    //implementation is generated separately
  void idl_deserialize(int serialization_type,OurInStream& is);
    //implementation is generated separately
};
```

On the other hand, for our Server, we might want to have *inventory* implemented as a special class Inventory, optimized for fast handling of specific server-side use cases. In this case, we MAY want to define our Server Mapping as follows:

```
MAPPING(“CPP”,“Server”) PUBLISHABLE_STRUCT Character {
  // here we're omitting all the default mappings
  float angle;
  class MyInventory inventory;
    //class MyInventory will be used as a type for generated
    //  Character.inventory
    //To enable serialization/deserialization,
    //  MyInventory MUST implement the following member functions:
    // size_t idl_serialize_collection_get_size(),
    // const Item& idl_serialize_collection_get_item(size_t idx),
    // void idl_deserialize_collection_reserve_size(size_t),
    // void idl_deserialize_collection_add_item(const Item&)
};
```

As we see, even when we’re using the same programming language for both Client-Side and Server-Side, we MAY need different Mappings for different sides; in case of different programming languages such situations will become more frequent. One classical (though rarely occurring in practice) example is that SEQUENCE<Item> can be mapped either to vector<Item> or to list<Item>, depending on the specifics of your code; as specifics can be different on the different sides of communication – you may need to specify Mapping.

Also, as we can see, there is another case for non-default Mappings, which is related to making IDL-generated code to use custom classes (in our example – MyInventory) for generated structs (which generally helps to make our generated struct Character more easily usable).

Mapping to Existing Classes

One thing which is commonly missing from existing IDL compilers is an ability to “map” an IDL into existing classes. This can be handled in the following way:

- you do have your IDL and your IDL compiler
- you make your IDL compiler parse your class definition in your target language (this is going to be the most difficult part)
- you do specify a correspondence between IDL fields and class fields
- your IDL generates serialization/deserialization functions for your class
  - generally, such functions won’t be class members, but rather will be free-standing serialization functions (within their own class if necessary), taking class as a parameter
  - in languages such as C++, you’ll need to specify these serialization/deserialization functions as friends of the class (or to provide equivalent macro)

"I want YOU to read page 2!

Continued on Page 2... Further topics include IDL Encodings (including Delta Compression, rounding, etc.) and IDL Backward Compatibility

Example: Encoding

We've already discussed IDL and Mapping (and can now use our generated stubs and specify how we want them to look). Now let's see what Encoding is about. First, let's see what will happen if we use "naive" encoding for our C++ struct Character, and will transfer it as a C struct (except for *inventory* field, which we'll delta-compress to avoid transferring too much of it). In this case, we'll get about 60bytes/Character/network-tick (with 6 doubles responsible for 48 bytes out of it).

Now let's consider the following Encoding:

```
1  ENCODING(MYENCODING1) PUBLISHABLE_STRUCT Character {
2    VLQ character_id;
3    DELTA {
4      FIXED_POINT(0.01) x;//for rendering purposes, we need our coordinates
5                          //only with precision of 1cm
6                          //validity range is already defined in IDL
7                          //NB: given the range and precision,
8                          // 'x' has 20'000'000 possible values,
9                          // so it can be encoded with 21 bits
10     FIXED_POINT(0.01) y;
11     FIXED_POINT(0.01) z;
12     FIXED_POINT(0.01) vx;
13     FIXED_POINT(0.01) vy;
14     FIXED_POINT(0.01) vz;
15   }
16   DELTA FIXED_POINT(0.01) angle;//given the range specified in IDL,
17                              //  FIXED_POINT(0.01) can be encoded
18                              //  with 16 bits
19   DELTA BIT(2) Animation;//can be omitted, as 2-bit is default
20                         //  for 3-value enum in MYENCODING1
21   DELTA VLQ animation_frame;
22   DELTA SEQUENCE<Item> inventory;
23 };
```

VLQ
A variable-

Here we're heavily relying on the properties of MYENCODING1, which is used to marshal our struct. For the

length quantity (VLQ) is a universal code that uses an arbitrary number of binary octets (eight-bit bytes) to represent an arbitrarily large integer.
— Wikipedia —

example above, let's assume that MYENCODING1 is a quite simple bit-oriented encoding which supports delta-compression (using 1 bit from bit stream to specify whether the field has been changed), and also supports VLQ-style encoding; also let's assume that it is allowed to use rounding for FIXED_POINT fields.

As soon as we take these assumptions, specification of our example Encoding above should become rather obvious; one thing which needs to be clarified in this regard, is that DELTA {} implies that we're saying that the whole block of data within brackets is likely to change together, so that our encoding will be using only a single bit to indicate that the whole block didn't change.

Now let's compare this encoding (which BTW is not the best possible one) to our original naive encoding. Statistically, even if Character is moving, we're looking at about 20 bytes/Character/network-tick, which is 3x better than naive encoding.

Even more importantly, this change in encoding can be done completely separately from all the application code(!) – merely by changing Encoding declaration

It means that we can develop our code without caring about specific encodings, and then, even at closed beta stages, find out an optimal encoding and get that 3x improvement by changing only Encoding declaration.

Such separation between the code and Encodings is in fact very useful; in particular, it allows to use lots of optimizations which are too cumbersome to think of when you're developing application-level code.

To continue our example and as a further optimization, we can add dead reckoning, and it can be as simple as rewriting Encoding above into

```
ENCODING(MYENCODING2) PUBLISHABLE_STRUCT Character {
  VLQ character_id;
  DELTA {
    DEAD_RECKONING(0.02) {//0.02 is maximum acceptable coordinate
                    // deviation due to dead reckoning
      FIXED_POINT(0.01) x;
      FIXED_POINT(0.01) vx;
    }
    DEAD_RECKONING(0.02) {
      FIXED_POINT(0.01) y;
      FIXED_POINT(0.01) vy;
    }
    DEAD_RECKONING { //by default, maximum acceptable deviation
                  //  due to dead reckoning
                  // is the same as for coordinate
                  //  (0.01 in this case)
      FIXED_POINT(0.01) z;
      FIXED_POINT(0.01) vz;
    }
  }//DELTA
  DELTA FIXED_POINT(0.01) angle;
  DELTA BIT(2) Animation;
  DELTA VLQ animation_frame;
  DELTA SEQUENCE<Item> inventory;
};
```

When manipulating encodings is *this* simple, then experimenting with encodings to find out a reasonably optimal one becomes a breeze. How much can be gained by each of such specialized encodings – still depends on the game, but if you can try-and-test a dozen of different encodings within a few hours – it will usually allow you to learn quite a few things about your traffic (and to optimize things both visually and traffic-wise too).

"How much can be gained by each of such specialized encodings – still depends on the game, but if you can try-and-test a dozen of different encodings within a few hours – it will usually allow you to learn

Backward Compatibility

One very important (and almost-universally-ignored) feature of IDLs is backward compatibility. When our game becomes successful, features are added all the time. And adding a feature often implies a protocol change. With Continuous Deployment it happens many times a day.

And one of the requirements in this process is that the new protocol always remains backward-compatible with the old one. While for text-based[4] protocols backward compatibility can usually be achieved relatively easily, for binary protocols (and games almost-universally use binary encodings due to the traffic constraints, see "Publishable State: Delivery, Updates,

Interest Management, and Compression" section above for discussion) it is a much more difficult endeavour, and requires certain features from the IDL.

What we need from an IDL compiler is a mode when it tells whether one IDL qualifies as a "backward-compatible version" of another one

quite a few things about your traffic (and to optimize things both visually and traffic-wise too).

Ok, this feature would certainly be nice for code maintenance (and as a part of build process), but are we sure that it is possible to implement such a feature? The answer is "yes, it is possible", and there are at least two ways how it can be implemented. In any case, let's observe that two most common changes of the protocols are (a) adding a new field, and (b) extending an existing field.[5] While other protocol changes (such as removing a field) do happen in practice, it is usually rare enough occurrence, so that we will ignore it for the purposes of our discussion here.

The first way to allow adding fields is to have field names (or other kind of IDs) transferred alongside with the fields themselves. This is the approach taken by Google Protocol Buffers, where everything is always transferred as a key-value pair (with keys depending on field IDs, which can be explicitly written to the Protocol Buffer's IDL). Therefore, to add a field, you just adding a field with a new field-ID, that's it. To be able to extend fields (and also to skip those optional-fields-you-dont-know-about), you need to have size for each of the fields, and Google Protocol Buffers have it too (usually implicitly, via field type). This approach works good, but it has its cost: those 8-additional-bits-per-field[6] (to contain the field ID+type) are not free.

The second way to allow adding fields into encoded data is a bit more complicated, but allows to deal with not-explicitly-separated (and therefore not incurring those 8-bits-per-field cost) data streams, including bitstreams. To add/extend fields to such non-discriminated streams, we may implement the following approach:

- introduce a concept of "fence" into our Encodings. There can be "fences" within structs, and/or within RPC calls
    - one possible implementation for "fences" is assuming an implicit "fence" after each field; while this approach rules out certain encodings, it does guarantee correctness

"introduce a concept of

- between “fences”, IDL compiler is allowed to reorder/combine fields as it wishes (though any such combining/reordering MUST be strictly deterministic).

“fence” into our Encodings

- across “fences”, no such reordering/combining is allowed

- then, adding a field immediately after the “fence” is guaranteed to be backward-compatible as soon as we define it with a default value

  - within a single protocol update, several fields can be added/extended simultaneously only after one “fence”

  - to add another field in a separate protocol update, another “fence” will be necessary

- extending a field can be implemented as adding a (sub-)field, with a special interpretation of this (sub-)field, as described in the example below

---

[4] such as XML-based
[5] that is, until we’re throwing everything away and rewriting the whole thing from scratch
[6] Google Protocol Buffers use overhead of 8 bits per field; in theory, you may use something different while using key-value encodings, but the end result won’t be that much different

Let’s see how it may work if we want to extend the following Encoding:

```
1  ENCODING(MYENCODINGA) PUBLISHABLE_STRUCT Character {
2    UINT16 character_id;
3    DELTA {
4      FIXED_POINT(0.01) x;
5      FIXED_POINT(0.01) y;
6      FIXED_POINT(0.01) z;
7      FIXED_POINT(0.01) vx;
8      FIXED_POINT(0.01) vy;
9      FIXED_POINT(0.01) vz;
10   }
11 };
12 //MYENCODINGA is a stream-based encoding
13 //  and simply serialized all the fields
14 //  in the specified order
```

Let’s assume that we want to extend our UINT16 character_id field into UINT32, and to add another field UINT32 some_data. Then, after making appropriate

changes to the IDL, our extended-but-backward-compatible Encoding may look as follows:

```
ENCODING(MYENCODINGA) PUBLISHABLE_STRUCT Character {
  UINT16 character_id;
  DELTA {
    FIXED_POINT(0.01) x;
    FIXED_POINT(0.01) y;
    FIXED_POINT(0.01) z;
    FIXED_POINT(0.01) vx;
    FIXED_POINT(0.01) vy;
    FIXED_POINT(0.01) vz;
  }
  //Up to this point, the stream is exactly the same
  //  as for "old" encoding

  FENCE

  EXTEND character_id TO UINT32;
    //at this point in the stream, there will be additional 2 bytes placed
    // with high-bytes of character_id
    // if after-FENCE portion is not present – character_id
    // will use only lower-bytes from pre-FENCE portion

  UINT32 some_data DEFAULT=23;
    // if the marshalled data doesn't have after-FENCE portion,
    // application code will get 23
};
```

As we can see, for the two most common changes of the protocols, making a compatible IDL is simple; moreover, making an IDL compiler to compare these two IDLs to figure out that they're backward-compatible – is trivial. Formally, IDL B qualifies as a backward-compatible version of IDL A, if all of the following stands:

- IDL B starts with full IDL A
- after IDL A, in IDL B there is a FENCE declaration
- after the FENCE declaration, all the declarations are either EXTEND declarations, or new declarations with specified DEFAULT.

On Google Protocol Buffers

Google Protocol Buffers is one IDL which has recently got a lot of popularity. It is binary, it is extensible, and it is reasonably efficient (or at least not unreasonably inefficient). Overall, it is one of the best choices for a usual business app.

However, when it comes to games, I still strongly prefer my own IDL with my own IDL compiler. The main reason for it is that in Google Protocol Buffers there is only one encoding, and the one which is not exactly optimized for games. Delta compression is not supported, there are no acceptable ranges for values, no rounding, no dead reckoning, and no bit-oriented encodings. Which means that if you use Google Protocol Buffers to marshal your in-app data structures directly, then compared to your own optimized IDL, it will cost you in terms of traffic, and cost a lot.

Alternatively, you may implement yourself most of the compression goodies mentioned above, and then to use Google Protocol Buffers to transfer this compressed data, but it will clutter your application-level code with this compression stuff, and still won't be able to match traffic-wise some of the encodings possible with your own IDL (in particular, bit-oriented streams and Huffman coding will be still out of question[7]).

Therefore, while I agree that Google Protocol Buffers are good enough for most of the business apps out there, I insist that for games you usually need something better. MUCH better.

"Therefore, while I agree that Google Protocol Buffers are good enough for most of the business apps out there, I insist that for games you usually need something better. MUCH better.

---

[7] that is, unless you're using Google Protocol Buffers just to transfer pre-formatted bytes, which usually doesn't make much sense

[[To Be Continued…

This concludes beta Chapter VII(d) 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 VIII, "Engine-Centric Architecture: Unity 5, Unreal Engine 4, and Photon Server from MMO point of view"]]

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

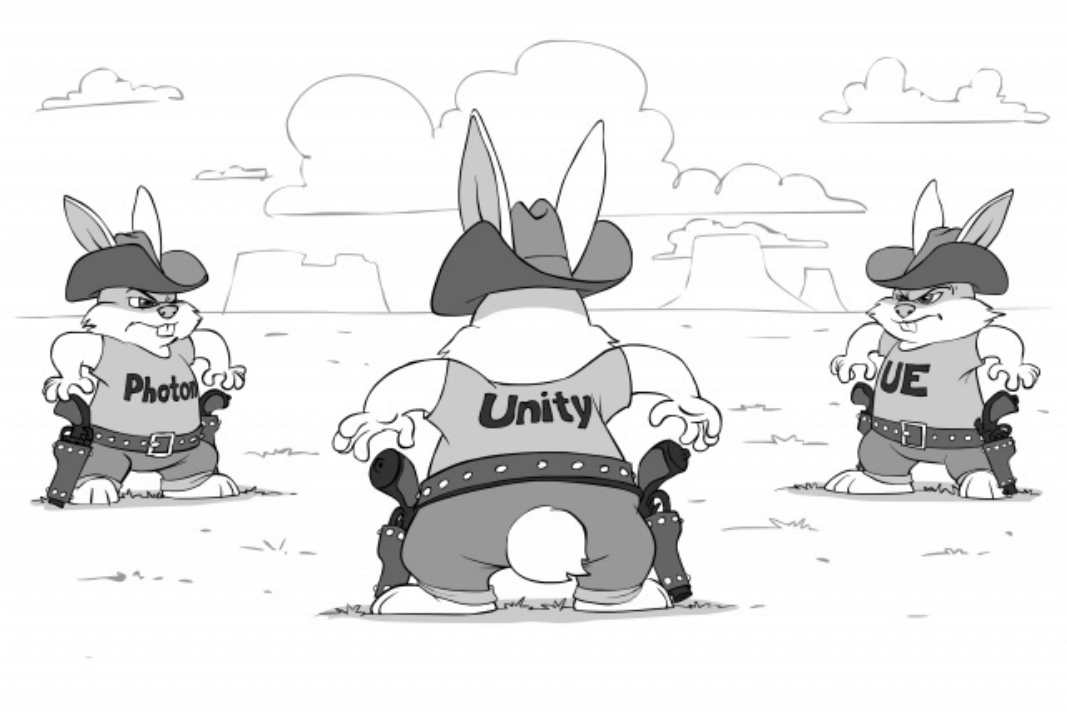
« *MMOG. Point-to-Point Communications and non-blocking RP…*

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: client, game, IDL, marshalling, multi-player, network, protocol, server

Copyright © 2014-2016 ITHare.com

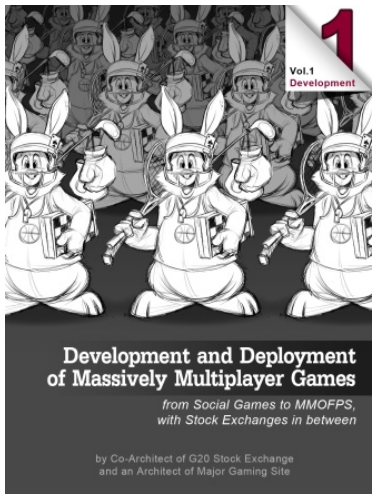
Unity 5 vs UE4 vs Photon vs DIY for MMO

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

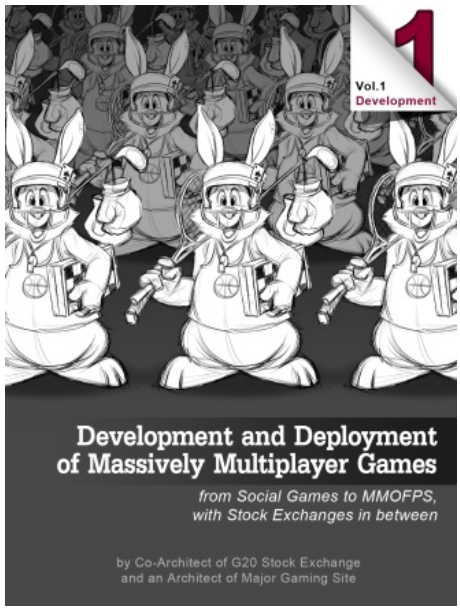
Copyright © 2014-2016 ITHare.com

Pre-Coding Checklist: Things Everybody Hates, but Everybody Needs Them Too. From Source Control to Coding Guidelines

posted February 29, 2016 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter IX 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.]]



We've discussed a lot of architectural issues specific and not-so-specific to MMOs, and now you've hopefully already drawn a nice architecture diagram for your multiplayer game.

However, before actually starting coding, you still need to do quite a few things. And I am perfectly aware that there are lots of developers. Let's take a look at them one by one.

Source Control System

"I don't want to go into a discussion *why* you need source control system, just saying that there is a consensus out there on it being necessary

To develop pretty much anything, you do need a source control system. I don't want to go into a discussion *why* you need source control system, just saying that there is a consensus out there on it being necessary for all the meaningful development environments. Even if you're single developer, you still need source control: the source control system will act as a natural backup of your code, plus being able to rollback to that-version-which-worked-just-yesterday, will save you lots of time in the long run. And if you're working in a team, benefits of source control are so numerous that nobody out there dares to develop without it.

The very first question about source control is "what to put under your source control system?" And as a rule of thumb, the answer is like

You should put under source control pretty much everything you need to build your game, but usually NOT the results of the builds

And yes, "pretty much everything" generally includes assets, such as meshes and textures.

On the other hand, as with most of the rules of thumb out there, there are certain (but usually very narrow) exceptions to both parts of this statement. For "pretty much everything" part, you MIGHT want to keep some egregiously large-and-barely-connected-to-your-game things such as in-game videos outside of your source control system (for example, replacing them with stubs), but such cases should be very few and far between. Most importantly,

the game should be buildable from source control system, and the build should be playable

, that's the strict requirement, other than that – you MIGHT bend the rules a bit.

For not including results-of-your-build – I've seen examples when having YACC-compiled .c files within source control has simplified development flow (i.e. not all

developers needed to setup YACC on their local machines), but once again – this is merely a very narrow exception from the common rule of thumb stated above.

Git

"Even I myself, being a well-recognized retrograde, has been convinced that git has enough advantages to qualify as "the way to go"

The next obvious question with regards to source control is "Which source control system to use?". Fortunately or not, there is a pretty much consensus about the-best-source-control-system-out-there being git.[1] Even I myself, being a well-recognized retrograde, has been convinced that git has enough advantages to qualify as "the way to go" for objective reasons (opposed to just being a personal preference).

Whether to host git server yourself or whether to use some third-party service such as github – is less obvious, especially for games. I would say that if by any (admittedly slim) chance your game is open source – you should go ahead with a github.

If your game is closed-source – then the choice becomes less obvious and depends on lots of things: from the size of your team to having on the team somebody who's willing to administrate (and backup!) your own git server (while it is certainly not a rocket science, it will involve some command-line stuff). Even more importantly, if your game has lots (such as multiple gigabytes) of assets – you probably should settle for an in-house git server, at least because downloading all that stuff over the Internet will take too much time.

---

[1] Actually, you can do more or less the same things with Mercurial, but unless you already have it in place, in most cases I suggest to stick with git, even in spite of Git's lack of support for locks, see "Git and unmergeable files" section below

Git and unmergeable files

For game development (and unlike most of other software development projects), you're likely to have binary files which need to be edited. More precisely, it is not only about binary files, but also includes any file which cannot be effectively merged by git. One example of these is Unity scene files, but there are many others out there (even simple text-based Wavefront .obj is not really mergeable).

A question "what to do with such files" is not really addressed by Git philosophy. The best way would be to learn how to merge these unmergeable files,[2] but this is so much work that doing it for all the files your artists and game designers need, is

hardly realistic ☹ .

The second best option would be to have a 'lock' so that only one person really works with the asset file at any given time. However, git's position with regards of locks is that there won't be mandatory locks, and that advisory locks are just a way of communication so that should be done out-of-git (!). The latter statement leads to having chat channels or mailing lists just for the purposes of locking (ouch!). I strongly disagree with such approaches:

all stuff which is related to source-controlled code, SHOULD be within source control system, locks (advisory or not) included

To use advisory (non-enforced) locks in git, I suggest to avoid stuff such as chat channels, and to use lock files (located within git, right near real files) instead. Such a lock file MUST contain the name (id) of the person who locked it, as a part of file contents (i.e. having lock file with just "locked" in it is not sufficient for several reasons). Such an approach does allow to have a strict way of dealing with the unmergeable files (that is, if people who're working with it, are careful enough to update – and push(!) – lock file before starting to work with the unmergeable file), and also doesn't require any 3rd-party tools (such as an IM or Skype) to work. For artists/game designers, it SHOULD be wrapped into a "I want to work with this file – get it for me" script (with the script doing all the legwork and saying "Done" or "Sorry, it's already locked by such-and-such user").[3] If you like your artists better than that, you can make a shell extension which calls this script for them.

The approach of lock-files described above is known to work (though creating commits just for locking purposes), but still remains quite a substantial headache. Actually, for some projects it can be so significant that they MIGHT be better with Mercurial and it's Lock Extension (which supports mandatory locking).

Let's note that there is also an issue which is often mentioned in this context, the one about storing *large* files in git, but this is a much more minor problem, which can be easily resolved (for example, by using git LFS plugin).

---

[2] Actually, for merging Unity scene files there was an interesting project [GitMergeForUnity], but it seems abandoned now ☹

[3] and of course, another script "I'm done with file", which will be doing unlock-and-push

Git and 3rd-party Libraries

"How to handle those 3rd-party libraries you're going to use?

One subtle issue with regards to source control system is how to handle those 3rd-party libraries you're going to use. Ideally, 3rd-party libraries should be present in your source control system as links-pointing-to-specific-version of the library, with your source control system automagically extracting them before you're building your game. It is important to point to a specific version of the library (and not just to head), as otherwise a 3rd-party update can cause *your* code to start crashing, with you having no idea what happened. On the other hand, such an approach means that it is *your* responsibility to update this link-to-specific-version to newer versions, at the points when you're comfortable with doing it.

git submodule does just that, and *git submodule update* will allow you to update your links to the most recent version of the 3rd-party library. However, it works only when your 3rd-party libraries are git repositories themselves

If your 3rd-party library is available as an svn repository instead of git – you may setup a git mirror of svn repository and then to use *git submodule* [StackOverflow.SvnAsGitSubmodule]. A similar trick can be used with Mercurial too (see [HgGitMirror] on creating git mirror from Mercurial).

And if you're using a non-open-source library – you'll probably need to put a copy of it under your source control system 🙁 .

BTW, about open-source and non-open-source 3rd-party libraries: there is another (even more important) issue with them, make sure to read "3rd-party Libraries: Licensing" section below.

Git Branching

"One of the things you need to decide when using git, is how do you work

As noted above, as a rule of thumb, you should be using git. And one of the things you need to decide when using git, is how do you work with branches. In pre-git source control systems, branching was a second- (if not third-) class citizen, and developers were avoiding branches at all costs. In git, however, everything is pretty much about branches, and in fact this ease-of-branching-and-merging is what gives git an advantage over svn etc.

IMO, most of the time you should be following the branching model by Vincent Driessen described in [GitFlow] and is known as "Git Flow". When you look at it for the very first time, it may

look complicated, but for the time being you'll just need a few pieces of it:

with branches

- *master* branch. As a rule of thumb, you should merge here only when milestone/release comes. All the commits to the *master* branch should come from merges from *develop* branch. Direct commits (i.e. commits which are not merges from *develop)* into *master* branch SHOULD NOT happen.
- *develop* branch. The branch which is expected to work. More precisely – it is usually understood as a branch that compiles and passes all the automated tests, though it is understood that no amount of automated testing can really guarantee that it is working. You should merge to *develop* branch as soon as you've got your feature working "for you" (and all the automated regression tests do pass). Direct commits (not from *feature* branches) into *develop* branch are usually allowed. On the other hand, leaving *develop* branch in a non-compilable or failing-automated-test state is a major fallacy, and you'll be beaten by fellow developers pretty hard for doing it (and for a good reason). Note that having *develop* branch temporary failing to compile or run tests (in a sequence like developer committed to *develop* – automated build failed – developer fixed the problem or reverted the merge[4]) is not considered a problem; it is leaving *develop* branch in unusable state for several hours which causes a backlash (and rightly so). More on it in "Continuous Integration" section below.
- *feature* branch. You should create your own *feature* branches as you develop new features. These *feature* branches should be merged into *develop* branch as soon as your feature (fix, whatever) is ready. Consider *feature* branch as your private playground where you're developing the feature until it is ready to be merged into *develop* branch. *Feature* branches are generally not required even to compile.
    - A note of caution: there is a breed of developers out there who prefer to live within their own *feature* branch for many weeks and months, implementing many different features under the same branch and postponing integration as long as they can. *This is a Bad Practice™,* and a rule-of-thumb of one-feature-for-one-feature-branch should be observed as soon as you've past your very first milestone.
    - It is also interesting and useful to note that modern source control systems (git included) tend to punish those who do their merges later. When both you and a fellow developer are working on your respective branches, and she got committed her merge 5 minutes before you, then it becomes *your* problem to

"There is a breed of developers out there who prefer to live within their own feature branch for many weeks and months, implementing many different features under the same

resolve any conflicts which may arise from the changes *both of you have made*. In most cases for a reasonably mature codebase, there won't be any conflicts, but sometimes they do happen, and

branch and postponing integration as long as they can

it is the second developer who becomes ~~a rotten egg~~ responsible for resolving conflicts

Print this profound truth in a 144pt font and post it on the wall to make your fellow developers merge their feature branches more frequently.

---

[4] not that revert of the merge in git is very peculiar and counter-intuitive, see [kernel.RevertFaultyMerge] for discussion

Continuous Integration

One thing which is closely related to source control, and which you should start using as soon as possible for any sizeable project, is Continuous Integration a.k.a. CI (not to be confused with Continuous Deployment, a.k.a. CD which is a very different beast and will be discussed in Chapter [[TODO]]).

"The basic idea behind Continuous Integration is simple: as soon as you commit something, a build is automatically run with all the tests you were able to invent by that time

The basic idea behind Continuous Integration is simple: as soon as you commit something (usually it applies to *develop* branch merges/commits as described above), a build is automatically run with all the tests you were able to invent by that time. If the build or tests fail – whoever made the "bad" commit, gets notified immediately.

In a sense, continuous integration is an extension of a long-standing practice of "night builds", but instead of builds being made overnight, they're made in real-time, further reducing the impact from "bad commits".

In general, continuous integration is almost a must-have for any serious development, how to implement it – is a different story.

In this regard, I tend to agree with [Bugayenko2014] that build-before-commit[5] is a better way to implement Continuous Integration than classical build-after-commit.

The problem with build-after-commit is the following. If (as in

nightly builds and with classical Continuous Integration) developers commit first, and only then automated build+test runs, then there is a risk that the build/test fails. And if it happens – there is a strong pressure to fix the commit-which-caused-failure (instead of reverting it) – in part, because of git peculiar behaviour when it comes to merge reverts (mentioned above). Which means that the whole team will stop development and will be working on the fix, ouch. This practice is very disruptive, and can easily introduce "commit fear" mentioned in [Bugayenko2014].

To address this problem, instead of running the build *after* commit has happened, you should make sure that your build is clean *before* committing.[6] It means that "faulty" builds never happen, yahoo!

> **"Instead of running the build *after* commit has happened, you should make sure that your build is clean *before* committing**

To follow "build-before-commit" approach within your CI system, you may want to do one of the following:

- create a VM image with your "build server" and give every developer a copy. Not that I really like this option, but it does exist.
  - it MUST be a responsibility of *every developer* to run a build+test before every commit to *develop* branch
- write your own script which takes your-*feature*-branch as a parameter, merges it with *develop* (without committing it yet!), builds, runs all the tests, and commits-the-merge-provided-that-everything-went-smoothly
  - it MUST be a responsibility of *every developer* to use ONLY this script for committing into *develop* branch
- use Travis CI as your Continuous Integration tool. Make all commits to *develop* branch ONLY via "pull requests" (they still should reside within your *feature* branches(!)).
  - in this case, Travis CI will report whether current pull request *would* build ok after merge [TravisPullRequests]. It MUST be a responsibility of every developer to check this "Travis OK" status *before performing merge* (at least in GitHub it is shown as a nice green checkbox near the pull request, if you have Travis integration enabled)
- use Rultor set of scripts (by the very same Bugayenko). *NB: I didn't try it, so I cannot vouch for it.*

---

[5] I don't agree with [Bugayenko2014] that "Continuous Integration is Dead", but I do agree that what he names "read-only master branch", and I name "building-before-committing" is a better way of doing things (though I consider it being yet another way to implement Continuous Integration, and not something radically different)

[6] an alternative would be to fix merge revert in git, and to rely on merge reverts

instead. However, as current behaviour is considered a feature rather than a bug, this is not going to happen any time soon

3rd-party Libraries: Licensing

One really important thing to remember when developing your game is that no 3rd-party library can be used without taking into account its license. Even open-source libraries can come with all kinds of nasty licenses which may prevent you from using them for your project.

In particular, beware of libraries which are licensed under GPL family of licenses (and of so-called "copyleft" licenses in general). These licenses, while they *do* allow you to use code for free, come with a caveat which forces you to publish (under the very same license) all the code which is distributed together with the 3rd-party library.[7] There are a few mitigating factors though. First, LGPL license (in contrast to GPL license) is not that aggressive, and usually might be used without the need to publish all of your own code (while changes to library code itself will still need to be published, this is rarely a problem). Second, if you're not distributing your server-side code[8] – then only the client-side code will usually need to be published (which tends to help a lot for web-based games). In any case, if in doubt – make sure to consult your legal team.

"Be careful with open-source projects which don't have any license at all

Another two things to be aware of in open-source projects, is (a) "something under license which is not a recognised open-source license (see [OpenSource] for the list of recognised ones), and (b) "something without any license at all" (you'll see quite a few such projects on github). (a) is usually a huge can of worms, and in case of (b) you cannot really use the project in any meaningful way (by default, everything out there is subject to copyright, so to use it – you generally need some kind of license).

On the other hand, anything which goes under BSD license, MIT license, or Apache license – can usually be used without licensing issues.

And of course, if you're using commercial libraries – make sure that you're complying with their terms (paying for the library does not necessarily mean that you are allowed to use it as you wish).

---

[7] in practice, it is more complicated than that, but if you want legally correct answers – you better ask your legal team

[8] distribution of server-side code may happen, for example, if you're selling your

server-side as an engine

Development Process

The next thing which you will need is almost-universally necessary (that is, unless you're a single-developer shop) and pretty much universally hated among developers. It is related to the mechanics of the development process. All of us would like to work at our leisure, doing just those things which we feel like doing at the moment. Unfortunately, in reality development is very far from this idyllic picture.[9]

For your game, you do need a process, and you do need to follow it. What kind of process to use – old-school project Gannt-chart-based planning with milestones, or agile stuff such as XP, Scrum, or Kanban – is up to you, but you need to understand how your development process is going to work.

"I am not going to discuss advantages and disadvantages of different processes here, as the associated debates are going to be even more heated then Linux-vs-Windows and C++-vs-Java holy wars combined.

I am not going to discuss advantages and disadvantages of different processes here, as the associated debates are going to be even more heated then Linux-vs-Windows and C++-vs-Java holy wars combined. Usually, however, you will end up with some kind of a process, which is (whether you realize it or not) will be some combination of agile methods; in at least two of my teams, we were using a combination of Scrum and XP long before we learned these terms 🙂 .

BTW, if you happen to consider Agile as a disease (like, for example, [AgileDisease]) – that's IMNSHO not because agile is bad per se, but most likely because you've had a bad experience dealing with an overly-confident (and way too overzealous) Certified Scrum Master who was all about following the process without even remote understanding of specifics of your project (and quite often – without any clue about programming). While I do admit that such guys are indeed annoying (and often outright detrimental for the project), I don't agree that it makes the concepts behind agile development, less useful even by a tiny bit.

One thing which should be noted about agile criticisms (such as [AgileDisease]), is that there is no real disagreement about *what* needs to be done; the sentiment in such criticisms is usually more along the lines of "we're doing it anyway, so do we need fancy names and external consultants?" To summarize my own feelings about it:

Do you need to have a well-defined development process?	**Certainly. All successful projects have one, even if it is not formalized.**
Do you need to have it written down?	Up to you. At some point you'll probably need some rules written down, but it is not a strict requirement.
Does your project need to be iterative?	**Certainly**
Do you need to have your iterations reasonably short (3 months being "way too much")?	**Certainly**
Do you need to name your iterations "sprints"?	Doesn't matter at all
Do you need to have your iteration carved in stone after it started?	It depends, pick the one which works for you at a certain stage of your project
Do you need to analyze how your iteration went?	**A good idea, whether naming it "iteration" or "sprint"**
Do you need to describe your goals in terms of 'use cases'/'user stories'?[10]	**Certainly**
Do you need to *name* them 'use cases'/'user stories'?	Doesn't matter at all
Do you need to name your project "Agile", or "Scrum", or <insert-some-name-here>?	Doesn't matter at all
Do you need a daily stand-up meeting?	Up to you, but often it is not so bad idea
	You SHOULD. It is damn important to

Do you need Product Owner (as a role)	**have opinion of stakeholders to be represented**
Do you need Product Owner as a *full-time* role?	Not necessarily, it depends
Do you need to name this role "Product Owner"?	Doesn't matter at all
Do you need Scrum Master (as a role)?	You will have somebody-taking-care-of-your-development-process (usually more than one person), whether you name it "Scrum Master" or not
Do you need a Kanban board?	Up to you
Do you need to use XP's techniques such as pair programming, merciless refactoring, test-driven development?	Up to you on case by case basis
Do you need a Certified Scrum Master on your team?	**Probably not**
Do you need an external consultant to run your Agile project?	**If you do – your team is already in lots of trouble**

Ultimately, whether you're using fancy names or not, your process *will* be a combination of agile processes, using quite a few agile techniques along the road. And it doesn't matter too much whether you're doing it because you read a book on agile, or because you've invented them yourself.

---

[9] it applies to *any* kind of development, game or not
[10] while they're not exactly similar, they're close enough for our purposes now

Issue Tracking System

Whether we like it or not, there will be bugs and other issues within our game. And even if there would be a chance that we wouldn't have any bugs – we'll have features which need to be added. To handle all this stuff, we need an issue tracking system.

> "Whether we like it or not, there will be bugs and other issues within our game.

If you're hosted on github, *and* your team is *really small* (like <5 developers) – you MIGHT get away with github built-in issue tracker. If you're hosting your own git server (*or* if your team is larger), you're likely to use some 3rd-party issue tracking. The most popular choices in this regard range from free Bugzilla, Trac, and Redmine, to proprietary (and non-free as in "no free beer") JIRA.

Which one is better – honestly, IMHO it doesn't matter much, and any of them will do the job, at least until you're running a 1000-people company[11] (in particular, all 4 systems above do allow to integrate with git).

One extra thing to think about in this regard is support for the artifacts used within your development process. Whether you want to use a Kanban board, Scrum "burndown chart", or a good old Gantt chart (or all of them together) – having these artefacts well-integrated into the same system which provides you with issue tracking can save you quite a bit of time. More importantly – it may help you to follow your own development process. So think about artifacts of your development process, and take it into account when choosing your issue tracking system. Also keep in mind that some of the plugins which implement this functionality (even for free systems(!)) can become pricey, so it is better to check pricing for them in advance.

On the other hand, this support-for-development-process-artifacts is only a nice-to-have feature of your tracking system; you can certainly live without it, and it only comes into play when all-other-parameters of your issue tracking system are about-the-same for your purposes. On the third hand 😉, these days issue tracking systems *are* pretty much about-the-same from purely issue-tracking point of view.[12]

---

[11] and if you do, you should look for a better source than this book for choosing your issue tracking system, as issue tracking is very far from being a focus here

[12] I realise how hard I will be beaten for this statement by hardcore-zealots-of-<insert-your-favorite-issue-tracking-system> but as an honest person I still need to say it

Issue Tracking: No Bypassing Allowed

There is one very important concept which you MUST adhere to while developing pretty much any software product:

ALL the development MUST go through the issue tracking system

It means that there MUST be an issue for ANY kind of development (and for each commit too). Granted, there will be mistakes in this regard, but you MUST have "each commit MUST mention its own issue" policy. The only exception to this rule should be if it is *not* a feature, but an outright bug, *and* the whole issue can be described by developer in the commit message.

It is perfectly normal for a BA to come into developer's cubicle and saying "hey, we need such and such feature, let's do it". What is not normal – is *not* to open an issue for this feature (before or after speaking to the developer). As for using e-mails for discussing features – I am against it entirely, and suggest to have an issue open on the feature, and to have all the discussion within the issue. Otherwise, 3 months down the road you will have lots of problems trying to find all those e-mails and to reconstruct the reasons why the feature was implemented *this way* (and whether it is ok to change it to a *different way*).

Even for a team of 5, for every change in the code, it is crucial to know *why* it has been made, and there should be one single source of this information – your issue tracking system.

Coding Guidelines

One last (but certainly not least) thing you should establish before you start coding, is coding guidelines for your specific project. In this regard my suggestion is *not* to copy a Big Document from a reputable source,[13] but rather start writing your own (initially very small) list of DO's and DON'Ts for your specific project. This list SHOULD include such things as naming conventions, and all the not-so-universal things which you're using within your project. More on naming conventions and project peculiarities below.

"One last thing you should establish before you start coding, is coding guidelines for your specific

Of course, your guidelines.txt file belongs to your source control system. And while you're at it – do yourself a favor and find for it the most prominent place you can think of (root directory/folder of your project is usually a pretty good candidate).

[13] this book included; in Chapter [[TODO]] there will be an example of my personal guidelines for C++, but as with any other source – don't copy it blindly

project

Naming Conventions

With naming conventions the situation is simple: it doesn't really matter which naming convention you use (myFunction() vs my_function() won't make any realistic difference, and debating it for hours is not worth the time spent). What *is* important though, is to do it uniformly across the whole project, so you should just quickly agree on *some* naming conventions and then adhere to them.

That being said, there is one thing in this regard which I actively dislike and which I am arguing against (on the basis that it reduces readability) – it is so-called "Hungarian notation". If you really really feel like naming your *name* variable as *lpszName* – the sky won't fall, but I suggest to drop these prefixes completely.

As for having some kind of naming convention for class data members – two popular conventions are *mDataMember* and *data_member_*, this is up to you whether to have such convention, it won't make that much difference anyway (that is, as long as you're using it consistently across the whole project).

Project Peculiarities

For pretty much every project you will have some peculiarities. For example, as we'll be programming within our ad-hoc FSMs, then threads will be pretty much out of question (at least outside of well-defined areas) – ok, so let's write it down into our guidelines.txt file (to the part which tells about FSMs). For a C++ project there is a common question whether you'll be using printf() or ostream for formatted output and logging – regardless of your decision,[14] it needs to be consistent for the whole project, so it also belongs to Code Guidelines. And so on, and so forth.

"For a C++ project there is a common question whether you'll be using printf() or ostream for formatted output – regardless of your decision, it needs to be

For C++, my personal set of Coding Guidelines will be discussed in Chapter [[TODO]], but as with any other 3rd-party source, you shouldn't copy it blindly and should develop your own one, based on your own task, your own style, and your own design decisions.[15]

[14] FWIW, my answer is 'neither – use cppformat instead', see Chapter [[TODO]] for further discussion

[15] roughly translated as: "whatever nonsense I write there, it is your responsibility to filter it out, so don't blame me if it doesn't work for you" 😉

consistent for the whole project

Per-Subproject Guidelines

One important thing to be mentioned here is that most of the projects will actually need more than one set of coding guidelines. Not only the subprojects can be written in different programming languages, but also subprojects can perform very different jobs, what in turn requires different guidelines.

For example, even if all your code is written in C++, the guidelines for infrastructure layer (the one outside of FSMs) and application layer (implementing FSMs) is going to be quite different. The former is going to use threads, will probably provide logging facilities so it will need to have direct file access (and probably access OS-specific services too), etc., and the latter is basically going just to call whatever-is-provided-by-infrastructure layer (concentrating on game logic rather than on "how to interact with OS").

As a result, I strongly suggest to use different guidelines for different layers of your game even if all of them are written in the same programming language; at the very least, they should be quite different between 3D engine, network engine, and game logic.

Enforcement and Static Analysis Tools

All the rules and guidelines are useless if nobody cares to follow them. Even if it is only *somebody* who ignores the guidelines, if such ignoring-guidelines-code is not rectified soon enough, it is often used as an example for some other piece of code, and so on, and so forth, which means a slippery road towards most of the code ignoring the guidelines ☹ .

To deal with *all* such guideline violations, there is no real substitute for code reviews. However, to catch *some* of them, it is usually a good thing to use an automated tool which will complain about most obvious violations. Such tools are specific to the programming language; list of such "static analysis" tools which (as I've heard, no warranties of any kind) work in real-world projects, include:

- checkstyle (Java). Checks for naming convention compliance etc.
- astyle (C/C++/Objective-C/C#/Java). Re-formats your source according to your preferences. Personally, I like to have a policy of "before committing to *develop* branch, all the code should be run through astyle".
- StyleCop (C#).

- cpplint (C++). Style checks against Google C++ style guide. *Not to be confused with lint.*

Actually, static analysis tools go much broader than mere style checking, and quite a few of them can find bugs. Most popular static analysis tools in this regard include:

- cppcheck (C++)
- PMD (Java)
- PC-lint (C/C++). Commercial.

There are also lots of other static analysis tools out there (see [Wikipedia.StaticCodeAnalysis.Tools]), but quite a few of them are known to cause more trouble then provide benefits (one of common problems of many tools is having too many false positives), so don't hold your breath until you tested the tool and see that it works for you.

[[This Concludes Vol.1 "Architecture". To Be Continued in Vol.2 "Development"…

This concludes beta Chapter IX from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Moreover, this concludes "beta" of the whole vol.1 "Architecture", yahoo! Further chapters from vol.2 "Development" will be published soon…]]

[–] References

[GitMergeForUnity] "GitMerge for Unity"
[StackOverflow.SvnAsGitSubmodule] "Is it possible to have a Subversion repository as a Git submodule?", StackOverflow
[HgGitMirror] "Create a Git Mirror", hg tip
[GitFlow] Vincent Driessen, "A successful Git branching model"
[kernel.RevertFaultyMerge] kernel.org, https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.txt
[Bugayenko2014] Yegor Bugayenko, "Continuous Integration is Dead"
[TravisPullRequests] "Travis CI. Building Pull Requests"
[OpenSource] "Open Source Initiative. Licenses by Name"
[AgileDisease] Luke Halliwell, "The Agile Disease"
[Wikipedia.StaticCodeAnalysis.Tools] "List of tools for static code analysis", Wikipedia

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« ***Unity 5 vs UE4 vs Photon vs DIY for MMO***

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture, Uncategorized
Tagged With: Agile, game, git, issue tracking, multi-player

Copyright © 2014-2016 ITHare.com