Chapter 8. Scalability

In Vol. II, we finished our first round of discussions about Client-Side (with more to follow in Vol. V), so now we can start considering the almighty Server-Side. To be honest – I am mostly a Server-Side guy – and this is IMO a Good Thing™ (there are *tons* of books out there about game Clients, and almost none about Servers).

Before we start, let's note that quite a few techniques which we already discussed in Vol. II, are applicable to Server-Side too. In particular – (Re)Actors, while being very useful for Clients, are at least as important for Servers; and while it *is* possible to build your game Clients and your Game Servers without (Re)Actors – as we'll see below, I'll argue for using (Re)Actors on Server-Side too.

Server-Side and Scalability

On the Server-Side, one of the very first things we need to take into account, is Scalability. In general, poor Scalability isn't usually observed until post-deployment; on the other hand –

we DO need to take Scalability into account while architecting and developing our system.

Otherwise, it can easily happen that at our finest hour – say, when we have a million players willing to play – we won't be able to handle even half of them (or even worse, our Servers will be slowed down to the point of being completely unplayable for everybody). In this case, instead of being the finest hour – it can easily become a disaster, with our game beginning a downward spiral towards oblivion.[1]

Let's note that Scalability is a big topic, and that this Chapter is certainly *not* the whole discussion on Scalability within this series. In particular, we were already discussing the mechanics of scaling for seamless Game World Servers in Vol. I's chapter on Communications, and we'll also discuss DB Scalability in detail in Vol. VI's chapter on Databases.

In this Chapter, we'll merely try to define a few terms – and to describe some very high-level approaches to Scalability (which, as we'll see a bit later, will affect our architecture greatly).

[1] no relation to Elder Scrolls

Performance != Scalability, but… Performance Still Matters

What to Scale?

In the context of multiplayer games, there are traditionally two main parts of the system which need to be scaled. The first one – is scaling Game Worlds; the second such part is scaling our Database.[2]

In addition, at some point you may need to scale your Matchmaking (and if you ever need to scale your Cashier and Payments – you're certainly in luck <wink />) – but from our current abstract perspective, they will be scaled in a manner similar to scaling your Game Worlds.

As a result, at least for the time being, we will concentrate on scaling of (a) Game Worlds, and (b) Database.

In the context of multiplayer games, there are traditionally two main points which need to be scaled. The first one – is scaling Game Worlds; the second such point is scaling our Database.

Improving Performance to Avoid the Need to Scale

Even *before* we need to scale our app – there is one more thing which we might be able to do to deal with the load; it is improving performance of our app, so that we can do more on the same hardware (actually – most of the time we'll need to do more on the same *CPU core*).[3]

Of course, improving performance doesn't provide infinite scalability. However, in the real-world it is all about real-world numbers. Let's take a closer look at them.

Let's consider a game which needs to handle 100'000 simultaneous players. And, as noted above – let's consider two major points which we need to scale: Game Worlds and Database.

Just like Scaling Up, improving performance doesn't provide infinite scalability. However, it happens that it is all about numbers. Let's take a closer look at them.

[2] more generally – persistent storage, but for the purposes of this book we'll name it Database anyway

[3] I don't want to engage in discussion whether "improving performance" is actually one of the forms of "Scaling Up" or not; for the purposes of this book – let's use "Scaling Up" in a

When scaling Game Worlds, and assuming a typical industry number of being able to support 100 players/core – we'll need around 1000 cores to support desired 100'000 players. Honestly – improving performance of our Game World by 1000x so we can run the whole thing on a single core, is not really realistic. In turn, it means that improving the performance of Game World Servers (while still being important to reduce Server costs(!)) won't save us from the task of ensuring Scalability for Game Worlds.

On the other hand, if trying to run a Database for the same game with 100'000 players – we'll usually be speaking about the numbers of 0.1-1 DB transaction/player/minute (see also the [[TODO]] section below). It translates into 10'000-100'000 DB transactions/minute ~= 150-1500 DB transactions/second. And this kind of load, as we'll see in Vol. VI, can be quite achievable on a single DB server (actually, even over one single DB connection - that is, if we do a really good job optimizing our DB performance).

In practice, it means that for scaling Databases, we can try to avoid dealing with Scalability until our game becomes Really Large™ (and usually, unless we're an AAA company with a huuuge marketing budget and lots of pre-launch buzz going on, it doesn't happen overnight).

Scaling Up – Doesn't Help Much for Game World Servers

As soon as we run out of options to optimize our code – we need to scale, there is no way around it. And to start discussing Scalability – we need to define some terms. First, let's observe that at least in theory, there are two different flavors of Scalability: Scaling Up (a.k.a. Vertical Scaling) and Scaling Out (a.k.a. Horizontal Scaling).

Scaling Up refers to merely buying a better hardware Server; most of the time, it is a purely hardware solution which doesn't require anything from software side.

For us (="developers"), Scaling Up sounds as a really nice idea: "Hey, we don't need to think about scaling of our software – just tell admins to buy a new Server, and we're done!" Unfortunately, there is one teensy problem with Scaling Up:

as of 2017, Scaling Up doesn't help much.

The reason for it is two-fold. First, we need to mention that

for Scaling Up, it is per-core performance which matters.

As we don't want/need to make our software scalable when Scaling Up – it usually means that our program can run only on N cores (not more); for such a program - buying new Server won't give it any performance boost other than *N*performance-of-new-cores/N*performance-of-old-cores*; this pretty much translates into the statement above.

The second observation leading to scaling up not working, is the following:

narrow sense of "purely hardware upgrade", so "improving performance" becomes a separate concept.

These days, the best per-core performance we can hope for – is an around-4GHz CPU core.[4]
Moreover, this didn't change much over last 15 years or so.[5]

Based on these two observations, we can see that if you're already running a 3GHz CPU (either server one or a desktop one) – possibilities of Scaling it Up are extremely limited; all we're speaking about – is 1.5x difference GHz-wise; in addition – it might be possible to get an additional 2x-or-so per-core performance gain due to better caches etc.

It means that

CPU-wise, the gain from Scaling Up we can expect is at most 3x.

As discussed above – for Game Servers we need about 1000x performance improvement to avoid the need to scale; and compared to required 1000x, 3x we can get from Scaling Up isn't much <sad-face />; it means that most likely, we *will* need to Scale Out our Game World Servers (more on Scaling Out below).

On the other hand – for Database servers (which tend to have loads which are already within the reach for single-DB-connection not-inherently-scalable architectures), that 3x gain which we might be able to obtain from Scaling Up, might provide additional breathing room (and a very significant one at that).

Scaling Out

Even for Databases, and even when trying to avoid dealing with Scalability by improving performance really hard, we're likely to hit a wall somewhere between 10'000 and 1'000'000 simultaneous players. And as discussed above – for Game Worlds neither Scaling Up nor improving performance will allow to support even 100'000 simultaneous players without splitting our load to multiple boxes/cores.

To deal with it –

we'll need to Scale Out.

Unlike Scaling Up which relies on performance of one single core/Server-box – Scaling Out is all about spreading the load across different CPU cores (Server boxes etc.). As noted above, this is the only feasible way to scale your programs these days (when you DO need to scale, that is). As a result – for the rest of this book, whenever I'm speaking about "Scalability" or "scaling" without specifying whether it is "Scaling Up" or "Scaling Out" – I will mean "Scaling Out".

[4] over the past 10 years, IBM Power was the CPU having the highest clock rate – and the fastest Power goes around 4.7-5GHz. Recently, AMD has reached 4.7GHz too, and Intel was rumored to release 5.1GHz CPU soon – though as of early 2017 it seems that these expectations didn't materialize.

[5] granted, per-MHz performance did increase over the last 15 years, but gains of the order of 3-4x or so do not help much with scalability

Shared-Nothing as The Only Way™ to Scale Linearly

[TODO/wiki: https://en.wikipedia.org/wiki/Shared_nothing_architecture] One very important consideration which is well-known, but is largely ignored (especially by fans of mutex-based synchronization and by DB Vendors trying to sell Enterprise versions of their DBs), is related to shared resources.

The sad truth in this regard is that

Concurrent writes to any resource cause contention, and contention kills *both* performance *and* scalability.

[TODO/wiki: Amdahl's Law]One way to explain it, is via a so-called Amdahl's Law; when applying it to the system where N threads perform 90% of their processing independently, and remaining 10% of the processing – under the common mutex, according to Amdahl's Law[6] our overall speedup (defined as "reduction in overall latency") – cannot possibly be better than 1/(1-0.9) ~= 10x, *that's regardless of number of cores you throw in.* Indeed – as soon as we have 10 cores working, our mutex will inevitably become the bottleneck, effectively preventing any further scaling.

In fact – throwing more cores than necessary, can easily *reduce* performance; this happens because

Costs of context switches are not negligible[7]

In [TODO: https://www.usenix.org/legacy/events/expcs07/papers/2-li.pdf], it was found that the costs of the thread context switch (including cache invalidation) can vary between 10'000 CPU cycles, and a *million* CPU cycles. From my experience, while I never seen such numbers as a 1'000'000 in practice – 50'000-100'000 CPU cycles are not uncommon. With this in mind, it is not surprising that

It is easy to build a multithreaded mutex-based program, which uses 8 cores to perform the same job *slower* than a single-core program which uses the same algorithm.[8]

These observations are NOT limited to threading – and apply to *any* resource which needs to be locked. In particular, with DBs there are *lots* of locks and contentions; in particular, *any* DB has to lock rows for writing,[9] *and* all-multi-object-ACID-DBs-I-know[10], inherently

[6] or to common sense, whichever comes first <wink />

[7] Amdahl's Law establishes only the *upper bound* for scaling, leaving these costs beyond its scope

[8] the problem will usually be with multithreading being too fine-grained and synchronizations too frequent, which in turn will lead to costs of switches dominating over the useful work.

[9] Yes, even MVCC-based DBs need resolve write-write conflicts via locking

[10] MySQL-with-MyISAM – or any other DB which doesn't support multi-object ACID transactions for that matter – doesn't qualify

have contention on the DB log file (and as DB log is *system-wide* – it can *easily* become The Bottleneck™).

As a result –

Shared-Nothing architectures is The Only Way™ to Scale Linearly

This all-important observation plays an extremely important role in practice, and effectively means that synchronization-based techniques such as mutex-based multithreading, and federated DBs, have *severe scaling issues*. Worse than that – this observation is not only a theoretical finding, but is also supported by numerous real-world experiences (mostly sad ones).

Contention-Related Issues

In general, contention can (and usually *will* – that is, if the load is high enough) cause the following problems:

- Significantly worse-than-linear scaling
- While *latency* of a synchronization-based multithreaded system *may* indeed improve compared to single-threaded one[11], in any case we can be 100% sure that *throughput* of such a multithreaded system becomes *worse*. This directly follows from synchronization costs being non-negligible, but is way too often ignored by massively-multithreaded zealots.
- As contention grows – synchronization costs go up. *NB: this doesn't directly follow from worse-than-linear scaling (in particular, Amdahl's Law stipulates worse-than-linear scaling even without referring to synchronization costs).*
    - This, in turn, means that as we're coming closer to the load limit – our system will behave *worse* that we predicted based on previous experiences, effectively spiraling out of control *much faster* than we expect.
    - Compare it to Shared-Nothing systems (such as (Re)Actors), where load increase tends to *reduce* context switches, and causes the system to behave *better-than-expected* in near-critical situations).

Making Overall Architecture Scalable as a Whole

Whenever we're faced with the task of the creating an MOG architecture – the very first thing we need to think about, is "how to make our architecture scalable as a whole". Here, "as a whole" is an extremely important qualifier, because – as we'll see below – it is extremely easy to design a system where only a *part* of it scales easily (at the cost of some other part becoming a Really Bad Bottleneck™ <sad-face />).

On In-Memory States and Multi-Player Games

[11] if the task you have lends itself to coarse-grained multithreading, *and* if you're careful enough

The show must go on
The show must go on, yeah yeah
Ooh, inside my heart is breaking
My make-up may be flaking
But my smile still stays on
— Queen

After discussing the very basics of Scalability in general – let's discuss Scaling Out our Game Servers.

To approach this task in an at least somewhat generic manner, let's observe that most multi-player games out there can be seen as a sequence of "Game Events". Let's define (very loosely) a multiplayer Game Event as "some dynamic interaction which involves more than one player, is limited in time and has an obviously observable outcome". Examples of Game Events include such seemingly different things as:

- Arena match
- Poker hand
- RPG fight (or talk)

Actually, *all* the multi-player games I know look to the player as a sequence of Game Events – with, maybe, some interspersed interactions which involve only one single player (for example, interactions with game environment and NPCs but not with other PCs).

***all* the multi-player games I know look to the player as a sequence of Game Events**

Now, as we realize that our multi-player game can be seen as a sequence of multiplayer Game Events, we can make a few further observations.

Observation 8.1. *If Game Event is interrupted for more than a few dozen seconds,*[12] *it is next-to-impossible to get all the players who participated in the Game Event, back to it.*

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 a two-player Game Event getting these two players back might work, but for 10+ – having *all* the players to return back to play a*t the same time* is very unlikely.[13]

Of course, if your game is a final of a big tournament with a big cash prize, you'll probably be able to reschedule it for the next day or something, but gathering the same people back

[12] unless we're speaking about big tournaments or large prizes, I'd put more or less typical time at about 1-2 minutes (with all the necessary disclaimers about it depending on your game etc. etc.).

[13] As always, there are some exceptions here and there, but they're few and far between

after 15 minutes or so of your game being irresponsive, won't be possible for the vast majority of Game Events out there.

Observation 8.2. *If Game Event is interrupted for more than a few dozen seconds, then even if we are able to reconstruct the same Game World State technically, it won't be the same from the player's point of view. Moreover, any substantial interrupt of the Game Event can easily provide an unfair advantage to some of the players.*

Being interrupted in the middle of a sword fight and being asked to resume from the middle of it (which was "who-knows-how-many-milliseconds-before-you-need-to-press-the-button") – is not likely to be satisfying for the players. In addition, if the interrupt is rather long – then from the players' perspective they will stay in a nervous state of "what exactly is my position within this fight" (which is quite unusual and therefore rather uncomfortable compared to usual "I am preparing for this fight"); this is usually worse than knowing that the whole thing will be rolled back and you can start anew (of course, if one of them was winning – it will be unfair, but well – there is no ideally fair solution here).

Being interrupted in the middle of a sword fight and being asked to resume just from the middle of it – is not likely to be satisfying for the players

Going even further into analysis of unfair advantages due to interruptions – we'll see that for quite a few games, it might be possible for a player to obtain some important game-changing information during the interrupt within the Game Event. This information can be pretty much anything – from noticing the start of opponent's move and preparing to counter it during the interrupt, to being able to run a sophisticated analysis tool in the middle of interrupted chess blitz match.

These effects are known in the game industry – though way too often they're taken into account only during deployment as an afterthought, and this can easily lead to ugly solutions and even uglier resulting problems. In "In-Memory Game World States: a Natural Fit for 'No Bugs' Rule of Thumb" section below, we'll see an example of rather crazy crash recovery logic of a large multi-million-dollar game: after the crash they first restored a perfectly correct current Game World State as-of-the-moment-of-the-crash (with this restore itself causing lots of trouble) – merely to follow it with rolling back this perfectly-correct-current-Game-World-State back to the start of Game Event – exactly because they weren't able to resume the game due to lack of players.

'No Bugs' Rule of Thumb for Multiplayer Games

The weight of evidence for an extraordinary claim
must be proportioned to its strangeness
— Pierre-Simon Laplace

Now, armed with these two observations, we can try to figure out what needs to be done if our Server app crashes in the middle of the Game Event (which inevitably causes a large interrupt in game play – that is, unless we're going for full-scale fault tolerance for all our

Servers, *and* the crash was a hardware one[14]). Personally, I prefer to state it as the following **“’No Bugs’ Rule of Thumb for Multiplayer Games”**:

Whenever Game Event is interrupted for significant time, as a rule of thumb it is better to roll back the interrupted Game Event rather than trying to restore the exact Game World State in the middle of the Game Event.

This statement is very bold – and as such, requires quite a bit of explanation. Let’s consider two options: the first one is to restore the exact Game World State at the moment of crash, and the second one is to roll back our interrupted Game Event (i.e. we restore the exact Game World State as of the beginning of current Game Event); moreover, let’s assume that the both options are feasible to implement (which is not often the case for the Option 1, but for our current analysis we can afford a bit of daydreaming).

In both cases, the player’s experience will be hurt. Of course, roll back in Option 2 obviously changes the game landscape. However, as it follows from Observation 8.2, restoring exact Game World State in Option 1 is also far from the ideal. I’d say that from the point of view of “providing the least possible disruption to the players”, for quite a few games out there both approaches are roughly equal (=”having bad impact on the game of the same order of magnitude”). In other words – *whatever we’re doing after the crash, gameplay will be hurt, and players will be inevitably unhappy (and some players will be inevitably more unhappy than the others).*

With this in mind, we should take into account considerations which have led us to Observation 8.1 – those about getting players back. If going the route of Option 1 (taking “restoring Game Event in the middle” route), we’re basically saying that “everybody who participated in that Game Event, needs to stay online for some unspecified time, just polling to see when we’re able to relaunch our Game Servers; anybody who doesn’t do it – will be punished by losing the Game Event by default”. Essentially, we’ll be punishing players for our own problems – which is certainly not good (and players tend to hate it too). On the other hand, if dealing with the crash via Option 2 – we can say “sorry, Bad Thing™ did happen – but at least we rolled back all the current Game Events, so you can come whenever-you-want and continue playing at your convenience”.

Overall, from what I’ve seen (both as a developer and as a player) – I like the second option (the “roll back to start of Game Event” one) much *much* better than the first one, at the very least – for most of the games out there. Which is exactly what is summarized in the “’No Bugs’ Rule of Thumb” boldly stated in bold[15] above.

[14] a reproducible software crash will be faithfully repeated on a reserve node (just as it happened with Ariane 5 rocket, see [Wikipedia.Cluster.LaunchFailure] for details), so no kind of Fault Tolerance will help against it.

[15] pun intended

BTW, it is exactly the same strategy which is traditionally applied for at least one brick-and-mortar game for centuries. As one poker pro has explained it: if a fire breaks out in a brick-and-mortar poker room while the hand is being played, then the whole hand gets cancelled, and all the chips are returned to their owners "as of the beginning of the hand" – regardless of the cards they were dealt and regardless of the state of the hand in general. Then, when the fire is extinguished, players can start a new hand – or some of them may leave; it is their choice, and certainly not the choice of the casino owner.

if a fire breaks out in a brick-and-mortar poker room while the hand is being played, then the whole hand gets cancelled, and all the chips are returned to their owners "as of the beginning of the hand"

Good for Us (as developers)

Now, let's note that all this analysis above stands even *before* we take into account the complexities of implementing perfectly-durable-intra-Game-Event-Game-World-States. In practice – these complexities are so big (essentially leading to pushing each-and-every-player-action into some-kind-of-durable-database – which in turn leads to increasing DB load anywhere from 10x to 1000x, and scaling DBs with that much transactions is going to be next-to-impossible) – that we'd likely to choose Option 2 just to avoid these complexities. However, my point here is different; what I am trying to say is that

In case of crash, rolling back Game Event is usually a Good Thing™ from the player's point of view

And the fact it also simplifies development – well, it means that we as developers got lucky: if the simplest-for-developers solution (the one with in-memory Game World States – see "In-Memory Game World States: a Natural Fit for 'No Bugs' Rule of ThumbTODO-ref" section below) happens to be the best one for players too, it is certainly a Good Thing™ for everybody involved.

Exception: Stock Exchanges

A word of caution for stock exchanges. If your game is a stock exchange, you generally *do* need to save every-player's-action persistently (to ensure strict correctness even in case of Game Server loss), so rolling back is not usually an option. Of course, technically we can say that with stock exchanges each single bid constitutes a Game Event, but well – it won't really simplify our jobs down the road.

That being said, it should be noted that even for stock exchanges at least the architecture based on Stateful Disposable Apps described below [[TODO - section?]], has been observed to work very well despite DB transaction numbers being rather large. At least in part, it can be attributed to two further observations: first, that for stock exchanges number of user interactions are usually not that high as for MMORPG,[16] and second, that price of the

[16] that is, if we exclude post-2007-or-so NASDAQ with lots of bots trading

hardware is generally much less of a problem for stock exchanges than for other types of games.

NOT applicable to Single-Player Games

It should also be noted that the logic above (and especially Observation 8.1) does not apply to single-player games (this includes over-the-Internet single-player games such as Internet slot machines etc.).

For a single-player game (whether Internet-based or not), the whole thing tends to work exactly the other way around: there is only one player, and she expects to resume the game *exactly* at the point when the whole thing was interrupted; moreover, the interrupted gameplay is usually already supported for single-player games, so handling it differently for the crash of our Servers will feel pretty bad. Even worse, with single-player games where the player is playing against the game (such as casinos), rolling the game back for any reason will have pretty bad implications and will raise pretty bad suspicions too.

In short – single-player games and multi-player games are two *extremely* different beasts in this regard, so observations about multi-player games SHOULD NOT be blindly extended to the single-player ones, and vice versa.

single-player games and multi-player games are two *extremely* different beasts in this regard, so observations about multi-player games SHOULD NOT be blindly extended to the single-player ones, and vice versa.

In-Memory Game World States: a Natural Fit for 'No Bugs' Rule of Thumb

From the discussion above it follows that if we had a Server crash with a subsequent reboot, then (as long as crash-reboot cycle took more than 2 minutes or so) – we'll need to roll back the interrupted Game Event, even if we have perfect data as of the exact moment of crash.

Now comes an all-important
Observation 8.3. Hey, but if we keep current Game Event in-memory only (writing any changes to the DB only *after* Game Event ends), we'll get *exactly* the behavior we need without any "rollback" efforts (and during normal operation, will lower the DB load by orders of magnitude too)

It means that for most of the multi-player games out there, we can use the following paradigm:

- we divide the game into Game Events, which need to be rolled back in case of Server crash or something
- while Game Event is in progress, this progress is maintained as a part of in-memory Game World State

- Game World States[17] SHOULD be written to DB only at the end of each Game Event, and not while the Game Event is in progress.
    - As a side benefit – this allows for the result of the Game Event to be written to DB atomically, so if there was one artifact for two players before they fight – we can be 100% sure that in DB there will be exactly one artifact after the fight regardless of whatever-has-happened.

Bingo! We can have our cake and eat it too! We've just got a very high-performance system (in-memory states without syncing to DB are about as fast as they go) – and it also provides very good player experience (well, as good as possible after something went horribly wrong).

BTW, if you choose to ignore this observation – you still can create a workable system, but the things can easily get rather ugly. Once, I've seen an architecture which wrote all the user actions to in-memory DB right away – effectively keeping perfectly current Game World State in that in-memory DB. It took them quite an effort to implement this DB, but it did work. However – whenever their Server crashed – they needed to roll-forward the whole thing, which in turn, in quite a few cases has led to the need to fix a bug-which-caused-the-crash "right on the fly" before roll-forward can be completed(!); as a result – the roll-forward implementation of the in-memory DB has been observed to cause quite a few long downtimes.

To add insult to the injury – in fact, all these efforts and complexities of roll-forward were completely pointless – because, whenever their Server crashed, their recovery procedure went as follows:

- first, they roll-forwarded all the DB logs to get a consistent DB state with all the user actions accounted for (including those actions within unfinished Game Events);[18]
- and right after the roll-forward was completed – they ran an application-level rollback to remove all those unfinished Game Events from DB; the latter was necessary exactly because of the problems with getting the players back to the same Game Event (see Observation 8.1).

[17] or a part of the Game World State, which part corresponds to the specific Game Event

[18] and, as noted above, the completing roll-forward could take fixing a bug, ouch!

In short – they made a complicated custom DB-level rollforward, only to follow it up with a complicated custom application-level rollback.

they made a complicated custom DB-level rollforward, only to follow it up with a complicated custom application-level rollback.

A competing system (couldn't help myself from bragging it was mine<wink />), simply didn't write all those unfinished Game Events into the DB while Game Events were in progress (and wrote the whole Game Event only after it is completed, instead). In case of crash (BTW, crashes were extremely rare) – it simply started from DB state (which, given the logic above, corresponded to the end of last-Game-Event), without any additional rollbacks.[19] The whole architecture was much simpler, scaled much better, and was observed to be much more reliable than competing in-memory-DB-based one described above.

On Data Consistency

One thing which comes to mind when considering such in-memory state-based processing models, is a question about data consistency: "hey, how losing information and data consistency can possibly be a good thing?". Here I need to mention that I am all *for* consistency; there is still a question, however, how to define this consistency.

As follows from the discussion above, from the player's[20] point of view, it is necessary to include "interrupts" into our definition of consistency; and to do it – we will have to say something along the lines of "if the game was interrupted for significant time in the middle of Game Event, then the consistent state is defined as the state at the beginning of the interrupted Game Event".

And as soon as we say it – our in-memory Game World State (synced to DB at the end of each Game Event) becomes a perfectly valid implementation (and a damn convenient one too <wink />) of the data consistency under the definition above. While another implementation discussed above – the one based on in-memory DB with a subsequent app-level rollback – is also valid under the same definition, it happens to be much less convenient in the real-world.

In-Memory State Summary

TL;DR on in-memory states:

[19] While in case of DB crash, a DB-level roll-forward to get consistent DB state was still necessary, but – as DB was a standard log-based RDBMS (and RDBMSs are doing log rollforwards for 50+ years now), it worked like a charm

[20] GDD, business requirements, etc.

- for multi-player games, if you disrupt a Game Event (such as match, hand, or fight) for more than a few dozen seconds – you won't be able to continue it anyway because you won't be able to get all the players-within-this-Game-Event back.
  - as a result, you'll most likely need to roll your whole Game Event back.
  - and to implement this rolling-back-to-the-beginning-of-the-Game-Event, in-memory states (with syncing to DB at the end of each Game Event) are very natural and convenient.
- As a result, the following processing model tends to work very well for multi-player games, so you SHOULD consider it very seriously:
  - Your gameflow needs to be split into separate Game Events.
  - These Game Events SHOULD be more-or-less natural from player's point of view
  - You store intra-Game-Event Game World State in-memory only.
  - You synchronize your in-memory Game World State with DB around the end of each of Game Events.
  - If your Server crashes in the middle of the Game Event – you lose your in-memory Game World State.
    - On restart – your system will restore itself from the DB, which corresponds to rolling the state back to the beginning of the interrupted Game Event.
      - It is a Good Thing™, as this is *exactly* what is required in vast majority of cases.[21]

if you disrupt a Game Event for more than a few dozen seconds – you won't be able to continue it anyway because you won't be able to get all the players-within-this-Game-Event back

The Myth of Stateless-Only Scalability

For quite a long time (and especially among webdevs), there exists a perception that to achieve scalability, all our request handlers need to be stateless (at least, they shouldn't have any state which persists between requests). In particular, in RESTful web services world, there is a lot of opposition to in-memory states; while in-memory states are not 100% prohibited, they are very much frowned upon, and all in the name of the perceived scalability. In the world of the all-popular Docker containers, it is leads to the notion that all the app containers need not only to be *immutable*, but also should be *ephemeral* (a.k.a. disposable).[22]

From a practical perspective, it translates into the following observation:

[21] YMMV, void where prohibited

[22] for more discussion on Docker, see Vol. VII's chapter on DevOps

It is widely (MIS)believed that stateless Server-Side Apps are The Only Way™ to scale Server-Side.

In this statement, it is “The Only” part which I am arguing against. Sure, having perfectly stateless request processing is a Good Thing™ – but only as long as you can afford it <sad-face />.

Pushing Scalability Problem to the Database

> *I am innocent of the blood of this just person: see ye to it.*
> *— Pontius Pilate, Matthew 27:24*

The problem with stateless processing (such as web-app-style stateless request handlers) is that

If our functional specification requires storing the state on Server-Side,[23] and we’re using stateless request handlers – then all the state inevitably ends up in the database.

This, in turn, means that by going for stateless request processing:

- There is no scalability problem on the request processing side anymore
  - In other words, we DO have perfect scalability for request processing apps, yay!
- This undeniable improvement for scaling request processing app, however, doesn’t come for free. More specifically - the whole scalability problem rears its ugly head at the database level.
  - And as we’ll see below – scaling database is much more difficult then scaling request handlers (even if we’re speaking about *stateful* request handlers), up to the point of being completely unmanageable <sad-face />.

The whole scalability problem rears its ugly head at the database level

In other words –

keeping our request handlers stateless, does NOT really solve the scalability problem; instead – it merely pushes the problem to the database.

Sure, if we’re working in a classical Huge-Company environment, we (as app developers) can say “it is not our problem anymore” (washing our hands of the matter Pilate-style).

[23] which almost-universally is the case: when going beyond simple web browsing, 100% stateless apps are very rare

However, if our aim is not only to cover our ***es to keep our current job while the project goes down, but rather want to make sure that the whole system succeeds[24] – we need to think a bit further. Most importantly, we need to realize that pushing the problem from us to DBAs isn't the end of the scalability problems; instead – we should ask ourselves:

with the kind of load we'll be throwing at the database, will it be feasible to scale the database[25]?

Databases and Scalability

As discussed above – we as app-developers DO need to think how much load we are allowed to throw at the database. And in this department, there are some pretty bad news for us. In spite of what your not-so-experienced DBA may (and your database salesman *will*) tell you – in general,[26] databases certainly do NOT scale trivially in a linear manner. Worse than that –

In a pretty much any serious real-world interactive system, it is database which is The Bottleneck™.

I remember a discussion with very knowledgeable architect-level guys from a pretty large company as early as in 2000; during the discussion, we had quite a few disagreements, but one thing was very obvious to everybody involved: *scaling everything-besides-database is trivial, it is database which is going to cause trouble scalability-wise*. Since that point, I've seen (and built) quite a few serious systems – and haven't see anything which might have changed my opinion about it.

[[TODO: wiki OLTP]]For the rest of this section, we'll be mostly speaking about so-called OLTP databases. Very very roughly - OLTP is the database where all the significant events in the system are getting recorded for the first time[27]; also – OLTP is also the database which ensures consistency, including such things as "when we try to transfer artifact X from player A to player B, we can be sure that there is exactly one copy of the artifact X regardless of the outcome". We'll discuss *much* more on OLTP (and how it can be optimized/scaled) in Vol. VI's chapter on Databases, and in Vol. VII's chapter on DB Optimizations.

[24] which BTW is the only way to think about it within DevOps paradigm; more on DevOps in Vol. VII's chapter on DevOps

[25] that is, without spending millions on hardware/licenses/maintenance

[26] at least if (a) multiple-object-ACID-transactions are necessary, and (b) there is no obvious sharding of the objects; unfortunately – most of the time both (a) and (b) happen to be the case for MOGs and MOG-like systems.

[27] After recording to OLTP – all the data can be replicated to other DBs for further analysis and reporting, so "for the first time" qualifier *is* important.

When speaking about real-world OLTP databases in 2017,[28] the following very practical observations usually stand (we'll discuss this topic in more detail in Vol. VI's chapter on Databases):[29]

- Pretty much any kind of load up to approximately 10 write-ACID-transactions/second is trivial
- When you need database loads of the order of 100 write-ACID-transactions/second – it is usually doable by using traditional database optimizations (indexes, caches, physical layout, BBWC RAID, etc.).
- Getting to 1000 write-ACID-transactions/second becomes severely non-trivial. If the database structure and loads don't allow for trivial sharding (in particular – if we need to allow players to play with anybody-they-want-to-play-with) – things start to get ugly. We'll discuss one way to do it, in Vol. VI's chapter on Databases.
- For a non-trivially-shardable database, 10'000 write-ACID-transactions/second (which, with usual MOG load patterns, roughly corresponds to about 100 billion transactions/year) inevitably becomes The Ultimate Nightmare™ for DBAs. In practice – such beasts are either huuuuge ultra-expensive systems, or Shared-Nothing systems (which require support from app-level, more on such systems in Vol. VI's chapter on Databases).
- Any systems with the load significantly above 10'000 write-transactions/second (or 100 billion write-transactions/year) tend to exhibit *both* (a) and (b) properties below:
    a) Such systems are very rarely really necessary. Most of those systems where such loads *are* necessary – are already household names. Just a few examples – Twitter handles about 200B tweets/year, Facebook is within single-digit hundreds of billions of comments and status updates per year too, UPS and Fedex deliver about 6B and 1B packages respectively (which translates into about 20x more package status updates – and again lands us into tens-to-hundreds-of-billions-write-transactions-per-day), and so on.[30]
    b) in practice, most of such systems (except for NASDAQ – see below) work either (b1) without strict ACID requirements, or (b2) for trivially-shardable scenarios (or both).

for a non-trivially-shardable database, 10'000 write-ACID-transactions/year (~= 100 billion write-transactions/year) inevitably becomes The Ultimate Nightmare™ for DBAs

[28] And however surprising it might sound – these numbers changed very little over last 15 years

[29] note that specific numbers below are extremely rough (give or take at least an order of magnitude); also let's note that we're speaking about real-world transactions (each writing at least a dozen rows, not trivially shardable, etc. etc.), so any overly-optimistic numbers such as TPC-C and any other artificial tests do not apply.

[30] Note that we didn't count statistical stuff such as counting web site visits etc. etc. However – this kind of data has absolutely nothing to do with ACID, and is trivially shardable too; as a result – it is beyond the scope of our current discussion.

- Out of those huge systems discussed above – none really require ACID (=”if a tweet of an average user[31] disappears once in a blue moon, the sky won’t really fall”), and all are almost-trivially-shardable (because of all the transactions actually being tied to one user or one package-being-delivered, without interactions with the DB between two user accounts or two packages).
- TBH, I know of only one real-world system which does need to go significantly higher than 100B ACID-transactions/year: it is post-2007-or-so NASDAQ (with lots of bot trading going on). Note that lots of the companies out there, while spending lots of efforts to achieve this number of transactions, don’t really need them (as one example, see the *Performance Perspective* section below for a brief discussion of *Uber*).
- On the other hand, games (including stock exchanges such as NASDAQ) tend to have pretty strong ACID requirements, and are *not* trivially-shardable too. We need the former because we certainly don’t want that artifact-which-costs-$20K-on-eBay, to disappear because of system crash at unlucky moment; and the latter happens because for most of the games, any player can interact with another player (with immediate consequences for *both* player accounts) just because she feels like it.

The observations above have several important consequences, but for the time being let’s note that

Increasing DB load by a factor of 10x, can easily kill the whole thing.

This, in turn, means that

When writing our apps, we MUST care about DB load.

As we’ll see below – app-level in-memory state can easily reduce number of write-ACID-transactions by factor of 10x to 1000x(!), which will make huge practical difference; as a result – at the very least we should include stateful apps into our consideration.

NoSQL to the rescue? Not really

When speaking about databases and scalability – these days we’re often told “hey, there are lots of NoSQL databases which can handle Big Data and scale to infinity”, implying that the whole scalability problem goes away.

[31] And while losing president’s tweet *can* indeed lead to quite unpleasant consequences (including diplomatic ones <ouch! />), separating such high-importance accounts and handling them separately and in durable manner, isn’t too difficult (after all, as we have seen above - it is all about numbers, and even Trump can’t produce anywhere close to a hundred billion tweets/year).

Unfortunately, it is not the case. While NoSQL databases indeed shine in certain scenarios (especially those when we need to perform read-only queries, *or* when our updates go to *independent* objects) – they tend to have very significant problems when dealing with OLTP-like load, with lots of writes and very high coherency requirements. We'll discuss these issues in detail in Vol. VI's chapter on Databases, but for now let's note that vast majority of NoSQL databases does *not* support multi-object database transactions with ACID guarantees (and proposed equivalents either don't scale, or aren't usable, or both). Yes – however sad it might sound, ACID support in most of NoSQL DBs out there is limited to single-object ACID transactions (which is not enough for 99% of OLTP processing tasks); moreover – extending this support to multiple objects under traditional NoSQL architectures will be at odds with their scalability.

BTW, I don't mean that NoSQL is a Bad Thing™; it is just that each technology should be used within its own applicability realm. In particular, these days OLTP is still better to be performed over traditional RDBMS; however – as we'll see in Vol. VI's chapter on Databases – it is perfectly possible to have eventually-consistent replica of this OLTP in Big-Data-oriented NoSQL, to process all kinds of historical queries there (which – depending on the type of query - can be *much* more efficient).

Scaling and In-Memory State

Let's compare two different approaches to scaling our MOG (or any other Server-Side-centric interactive distributed system for that matter). The first approach will be based on classical web-like Stateless Server-Side Apps. Within this model, everything is very simple: all the request processors are stateless – which means that all the state ends up in the database.

An alternative approach will be using *some* kind of in-memory state. As we'll see below, there are at least two different ways to organize such a state (via making our Server-Side Apps Stateful, or via using a centralized write-back cache) – but for the time being we'll be concerned only about having *some* kind of an in-memory state (the one which goes beyond simple read cache).

We'll be comparing Stateless vs In-Memory-State-Based systems from several different angles; in particular – we'll be concerned with (i) performance, (ii) durability, and (iii) scalability.

Performance Perspective

As noted above, in case of Stateless Server-Side Apps, we're bound to store everything to the DB. And for a vast majority of game-like systems, this is going to be prohibitively expensive. A few examples from different genres:

- For a virtual world simulation, writing everything to the database is going to be a non-starter; as the state of each player usually changes 20 times per second, making this much transactions per player is going to kill the whole thing even for a very modest number of players. *NB: I've got quite a few comments saying that that nobody in a sane state of mind will ever try using Stateless approach for a simulation; while I agree that doing so would be outright crazy, I still have to list it here at least for the sake of completeness.*

- For a casino-like game such as poker, we'll need to write every single player action to DB. This means making on average about 20 DB transactions per hand. *NB: for casino-like games, I have seen Stateless approach being tried A LOT – and most of the time, the results were devastating as the game grew.*
- Even for a social farming game, we could easily end up with several dozens of clicks per player-currently-using-farm, per minute.

Let's compare it to the In-Memory-State-Based system, the one which writes changes to Game World State to DB only at the end of the Game Event (as was discussed at length in *In-Memory Game World States: a Natural Fit for 'No Bugs' Rule of Thumb* section above):

- For a virtual world simulation, we can write changes to player state, only at the end of Game Events such as fights (conversations having consequences, etc.). It will often allow us to write things to DB once-per-minute or so (which is a 1200x(!) improvement compared to the stateless approach above).
- For a poker game, we'll need to write only the outcome of each hand to DB, corresponding to approximately 20x savings compared to naïve stateless approach.
- For a farming game, most of the time we can make an artificial Game Event (which ends either on a Really Important Achievement™, or after a certain timeout). In practice – we can easily save up to 10x DB-activity-wise (compared to the stateless app).

As we can see –

In-Memory State can easily reduce our database load by a factor of 10x-1000x.

Moreover, as it is DB which is usually The Bottleneck™ – it means that we're saving this enormous amount of load, *exactly* where it really matters.

BTW, this observation goes far beyond traditional games. Some of us remember that epic migration of *Uber* first from MySQL to PostgreSQL in 2013 (Klitzke, Migrating Uber from MySQL to PostgreSQL 2013), only to migrate back to MySQL (with a custom extension) 3 years later (Klitzke, Why Uber Engineering switched from Postgres to MySQL 2016).

it is DB which is usually The Bottleneck™ – it means that we're saving this enormous amount of load, exactly where it really matters.

While I didn't follow this story too closely myself, I heard an opinion that *Uber* would fare much better if they'd avoided the supposedly perfectly-scalable stateless-app architecture, and kept the most common update (the same source has reported it as storing "current position of the car") as mostly-in-memory only (writing to DB at large intervals, like "write the whole history of the car positions once per hour per car") – pretty much along the lines discussed above. [32] Sure, all the completed trips still need to be saved immediately (they have direct financial implications, and *do* need to be durable even if the Server-Side App crashes), but with mere 1 million trips per day which *Uber* has (that's just 30 transactions/second even accounting for intra-day load variations) – writing it down is trivial even for a single-writing-DB-connection OLTP system (in fact – the single-writing-DB-connection was seen handling 50M+ real-world transactions/day[33]; for details – see Vol. VI's chapter on Databases).

Sure, as I didn't make this analysis myself – I cannot really vouch for it, but I have to say that given the numbers above – it looks quite plausible. Moreover, I did see a large real-world game (which I unfortunately cannot name here), which experienced *exactly* this kind of problems (and the problems happened *exactly* due to making everything stateless, effectively increasing database load 10x+-fold, and causing lots of trouble for the database, DBAs, and ultimately – for end-users).

Durability Perspective

Of course, these performance benefits of In-Memory State don't come for free (nothing does). The currency we'll be paying with for this drastically improved performance, is Lack of Durability. In other words – if our App-which-handles-In-Memory-State crashes[34], we'll lose all the state which haven't been saved yet to the DB.

On the first glance this lack of Durability may look Really Bad™, but on the other hand, for most of MOGs – it is exactly the behavior we want

On the first glance this may look Really Bad™, but on the other hand, for most of MOGs – it is exactly the behavior we want (see the [[TODO]] section above for discussion). Moreover, even for a non-gaming interactive systems such as *Uber*, going along these lines is perfectly acceptable at least for some of the data. Using *Uber* data as an example – if in case of App crash we lose trips – it would be a significant problem, but losing half an hour of historical data about historical positions of the cars won't be noticeable (as these historical data is used only for statistical purposes – losing a very minor random portion of it won't change the stats).

Alternatively – there is a possibility to make our App-which-handles-In-Memory-State Fault-Tolerant (for a relevant discussion – see Chapter 10). Still, to be honest, as Fault Tolerance doesn't prevent from software-bug-induced crashes – for fast-changing business- and game-like apps I'd rather not risk to rely on it (in other words – in business world,[35]

[32] using our terminology – it would mean creating an artificial Game Event once per hour
[33] and BTW, transactions were significantly more complicated than writing the trip down
[34] depending on the specifics, it can be either Stateful Server-Side App, or the app which handles Write-Back Cache
[35] I am certainly *not* speaking about nuclear reactors or medical devices

crash costs and crash prevention costs are balanced in a way that implies that sooner or later, crashes will happen).

Scaling Perspective

[Bart] – Well, you're damned if you do,
[Chorus] – Deep, deep trouble.
[Bart] – Well, you're damned if you do,
And you're damned if you don't.
— The Simpsons

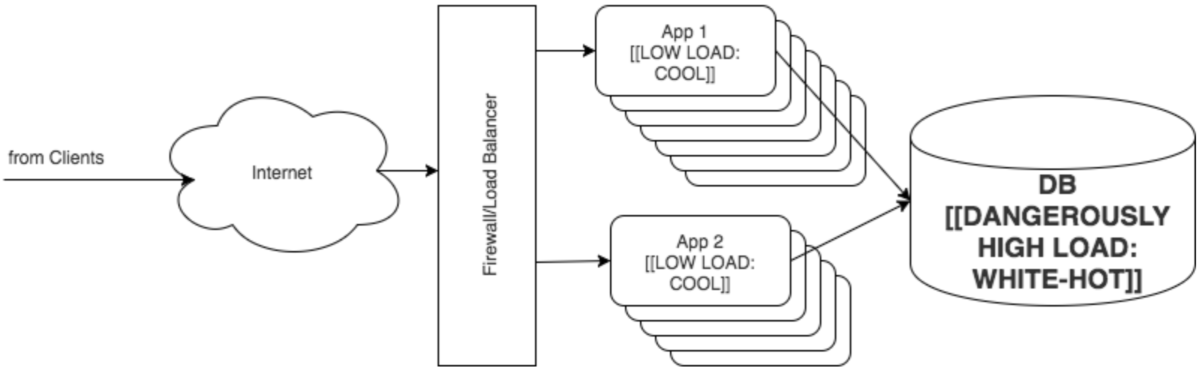
As already noted above, *performance != scaling*, so let's take a look at our Stateless vs In-Memory-State-Based systems from scaling perspective too. Actually, at this point we'll become more specific than merely deciding on "should we use In-Memory State or not", and will recognize *four* distinct scaling models.

Scaling Stateless System

As discussed above, when speaking about a system based on Stateless Apps, scalability is trivially achievable: we just need to create new (or use an existing) instance of our Stateless App – and bingo! – we got our scalability.

A very high-level diagram of this approach is shown on Fig 8.1:

Fig. 8.1



[[TODO/Fig 8.1-8.4: decide how to show "cool" and "hot"]

It all looks very simple: after Clients come to our Load Balancer, asking for a service from App 1 – they're randomly directed to one of the instances of App 1 (these instances may run on one Server Box or can be spread over several different ones); exactly the same happens for App 2 (or any other type of App).

In this model, all the Apps are perfectly stateless (i.e. they carry no meaningful state between requests), and therefore they can be created/destroyed as necessary (i.e. in Docker-speak, they're *ephemeral*). From the point of view of scaling Apps – it is a perfect scenario.

On the other hand, as discussed above, the real-world task is never formulated in terms of scaling only apps; instead – we need to scale the whole system; and in this regard Stateless-App-based systems exhibit significant problems.

as discussed above, the real-world task is always about scaling the whole system, including database; and in this regard Stateless-App-based systems exhibit significant problems.

In particular, as discussed above, for Stateless-App-based architectures, all the scalability work is pushed down to the database. Of course, for some of Server-Side developers it means merely pushing the responsibility to somebody-else with relief, but we're currently wearing our architectural hat, so assuming that "somebody will do it for us" (without an understanding how it will be done) is not really an option.

Moreover, in practice, in the model shown of Fig. 8.1, DB (which needs to handle all the state updates merely because there is no other place to store them) becomes an extremely bad white-hot bottleneck <sad-face />. Once, a DBA of such a system told me about a nightmare he had – it was about the servers which got so hot that they started melting. Fortunately, I never found myself in such position – but I can understand him perfectly; all the associated nightmares are happening because

pushing unnecessary stuff to DB-which-is-already-The-Bottleneck, is a Pretty Bad Idea™.

As discussed above – scaling DBs is a very well-known huuuuge headache (a.k.a. Deep Trouble™); achieving even 1000 real-world transactions[36] per second (which – taking into account usual MOG-like load patterns – corresponds to about 30M transactions/day, or 10 billion transactions/year) is already pretty difficult; going above this number has several very unpleasant consequences:

- Non-trivial solutions are required.
- Costs go through the roof, but as the dependency between load and costs is highly non-linear, spending more doesn't help much.[37]
- job of DBAs becomes extremely difficult.

[36] NB: we're not speaking of highly artificial tests such as TPC-C, but about real stuff

[37] BTW, make sure not to trust benchmarks-by-DB-vendors which tell you enormous numbers such as millions transactions/second; in short – all such benchmarks I've seen, were having at least one Really Big Issue™ which made them completely inapplicable to real-world cases (at least those cases which are somehow related to MOGs and OLTP DBs in general).

- overall reliability suffers, starting from a very simple observation: the more DB Server Boxes you need to run – the higher chances are that at least one of them crashes.
    - [[TODO:wiki MTBF]] This, in turn, leads to convoluted fault tolerant systems (with fault tolerance further taking its toll both in terms of bugs and in terms of reduced performance). For example, if our stateless system causes 10x more DB load – for larger loads it often means that we'll need about 30x-50x server boxes instead of one[38] - and unless we're taking special measures, it will bring MTBF of our DB down from rather comfortable "one DB failure in 5 years"[39] into disastrous "one DB failure every month".
        - As a result – we'll need to deploy some kind of fault tolerance for our DB (which will result in huuuge increase of complexity and associated costs[40]).

Of course, in some cases simple sharding will do the trick; in particular – for most of the single-player games sharding is trivial (each player sits in her own shard, with no interactions between the shards). However, as we're speaking about multiplayer games (where it is usually impossible to restrict "which players are allowed to interact with each other" – see also discussion on it in Vol. I's chapter on GDD) – sharding will rarely work (at least not without significant help from the app level).

for multiplayer games, simplistic sharding will rarely work

While (as we'll discuss it in the Vol. VI's chapter on Databases) it is usually possible to scale an MOG OLTP DB to 10B DB transactions/year and probably beyond – it is a very significant effort, which requires lots of complicated work (and while it seems having no apparent scalability problems – I didn't see it scaling beyond 10B DB transactions/year, so I cannot really vouch for it).

As a result:

- I am usually suggesting to *postpone* DB scaling as long as feasible
    - First, such a postponing significantly speeds up initial development (improving all-important Time to Market)
    - Second, by the time when we do need DB scaling, we'll know *much* more about specifics of our game, and will be able to make *much* more informed decisions.
    - That being said – it is of paramount importance for keeping the door for such scaling open (so when it is necessary – we *can* do it). To achieve this – at least

[38] In spite of what your DB vendor will tell you, you'll most likely find that for ACID transactions over the game-like tables (where everybody can interact with everybody) things *very* rarely scale in linear manner.

[39] While not exactly perfect, for most of the games it is acceptable.

[40] If using commercial RDBMS, then license for a fault-tolerant DBs will cost you arm and leg outright, but even if you're using free DBMS, costs of hardware and administration of a fault-tolerant DB are going to be very high.

we need to keep interactions between our Game Servers and our DB Server, to a very-well-defined DB Server API, with this API expressed in terms of Game Servers (and NOT in terms of SQL); more on it in Chapter 9.

- And to postpone the scaling as far as possible – reducing DB load by factor of 10x helps *a lot.*

If trying to look at the issue of DB load from a different perspective of "how we want to see our system *when* we finally need to scale" – then we'll certainly want to keep our system leaner (cheaper to maintain, less complicated, etc.), so reducing DB load also comes handy.

Overall, from whatever angle we'll look at it, we come to the same conclusion:

If we can reduce DB load by factor of 10x-100x – we should do it (even if it comes at cost of using an In-Memory State).

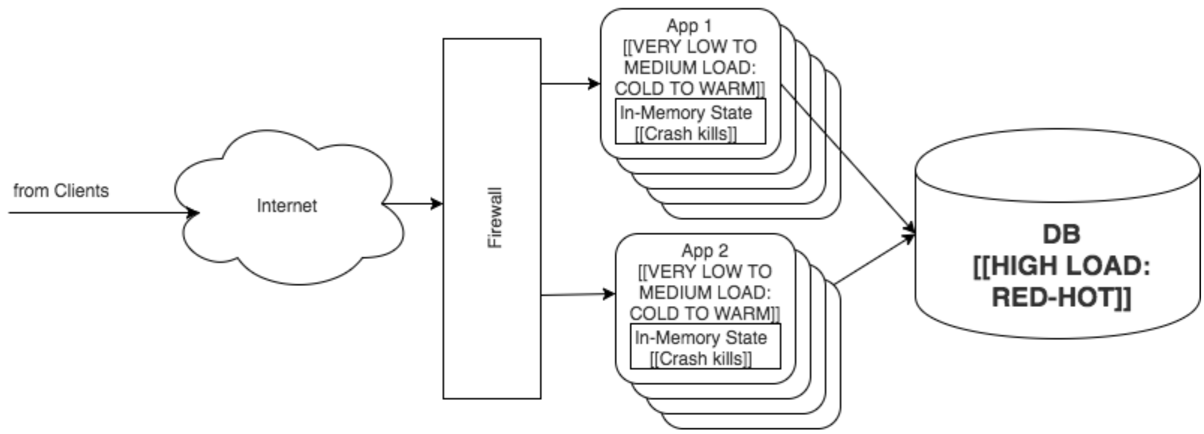
Note that I am not arguing for pursuing optimizations-which-save-mere-20%-or-so at architectural stage; these are usually too small[41] to shift the balance from one architecture to another one; however, a 10x performance improvement due to better architecture, most of the time does qualify as a game changer (pun intended).

Scaling Stateful-App-Based System

Very often, this 10x+ reduction in DB load can be achieved by using Stateful Apps (IMPORTANT: this is possible ONLY if our business logic/GDD is ok with relaxed Durability guarantees for the data which we decide to keep in-memory only).

A corresponding diagram is shown in Fig 8.2:

Fig. 8.2



Compared to Stateless-Based approach shown on Fig 8.1, we can see the following differences:

- Our Apps got In-Memory state. In turn, it means that:

[41] well, unless you find 15 of such optimizations, each providing 20% gain *and* being independent – but it rarely happens

        - This In-Memory State is not durable, and can be lost (as discussed above, this is exactly the behavior we want while Game Event is in progress)
        - Achieving balance between different Apps is not that trivial (in practice, I've never seen imbalance to be a significant problem, but I do know scenarios when it may happen[42]).
- As our Apps are now Stateful, it means that (unlike with stateless web apps), there are potentially two separate aspects for our Load Balancing:
    - First, we need to make sure that server-box-which-carries-our-state, has enough CPU power; in other words – we need to Load-Balance our Stateful Apps between different Server Boxes.
        - Most of the time, this balancing will be less perfect than Load Balancing of Stateless Apps; from what I've seen – it is possible to keep these discrepancies in check, but in certain cases it may become a rather significant headache.
        - For more discussion on such Worlds-to-Servers Load Balancing, see [[TODO]] section below.
    - Second – we may need to balance incoming players (or requests) among the Servers. While usually it is not a problem (as each of the Apps tends to serve about the same number of players) – in cases when we're broadcasting some Very Important Game™ (like some big final) to everybody-who-wants-it, this MAY start causing significant trouble. For an example of a solution – see discussion about Front-End Servers in Chapter 9.
- The main advantage of this approach is about reduced DB load. This, in turn, is achieved by writing to DB *only* at the end of Game Events.
    - In turn, it means that we need to identify those Game Events which allow/require writing to DB[43], and to understand implications of rolling back to the beginning of the Game Event in case of crash.

The main advantage of this approach is about reduced DB load

Overall, from what I've seen in the wild, Stateful-App-Based systems tend to *both* perform *and* scale much *much* better than Stateless-App-Based ones. On the other hand - of course, if Durability for *each* action taken is a firm requirement, these architectures won't fly (at least without ensuring fault tolerance for the Apps, preserving their In-Memory State).

Scaling System Based on Stateless-App plus In-Memory Write-Back Cache

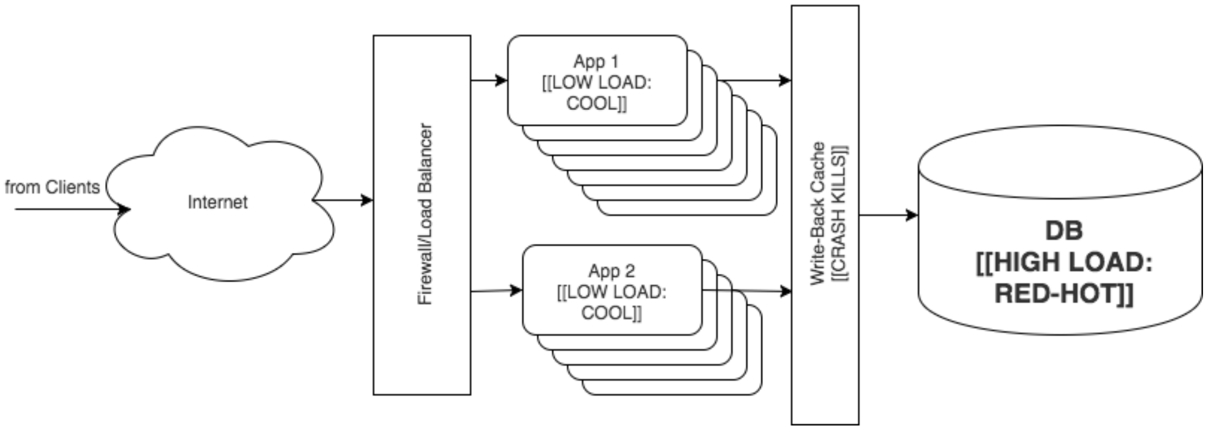
At this point, we have to note that strictly speaking, having an In-Memory State somewhere in the system does NOT necessarily imply that it is our Apps which need to be Stateful.

[42] in practice, if (a) the number of App instances running on a single server, is high, and (b) the load is restricted by number-of-players (opposed to number-of-observers) – achieving reasonable balance is going to be pretty simple.

[43] or create them artificially, as mentioned above for social farming games, and for *Uber*

Instead of using Stateful Apps, to save on the DB load while keeping our Apps stateless, we can have a centralized In-Memory write-back(!) cache sitting between our apps and DB, as shown on Fig 8.3:

Fig. 8.3



From the point of view of scaling, this model is a kind of "hybrid" between the Stateless-App and Stateful-App models. In particular, with such a Stateless-App-plus-In-Memory-Write-Back-Cache model:

- Like with Stateful-Apps, we do reduce DB load a lot.
    - As discussed above, this simplifies scaling DB greatly
- Like with Stateful-Apps, we do need to identify our Game Events, and to ensure DB writes at the end of Game Events (though with In-Memory Cache, it will be done by write-back cache on instructions of our App)
- Like with Stateful-Apps, we do sacrifice Durability between Game Events (i.e. crash of Write-Back Cache kills all the stuff which wasn't written to DB yet)
- Like with Stateless-Apps, we can Load-Balance only the incoming requests (or players) – and there is no need to Load-Balance the Stateful-Apps.

- Scaling In-Memory Cache is rarely a problem.

This approach tends to work pretty well at least for social games (in particular, those using Web-Based Deployment Architecture, which we'll discuss in Chapter 9) – and *may* work for medium-paced games such as casino games too. On the other hand, for really fast-paced games (especially simulations) this model won't really work because of (a) latencies to retrieve the state, and (b) because of enormous traffic between Stateless Apps and In-Memory Cache.

This approach tends to work pretty well at least for social games in Web-Based Deployment Architecture – and *may* work for medium-paced games such as casino games too.

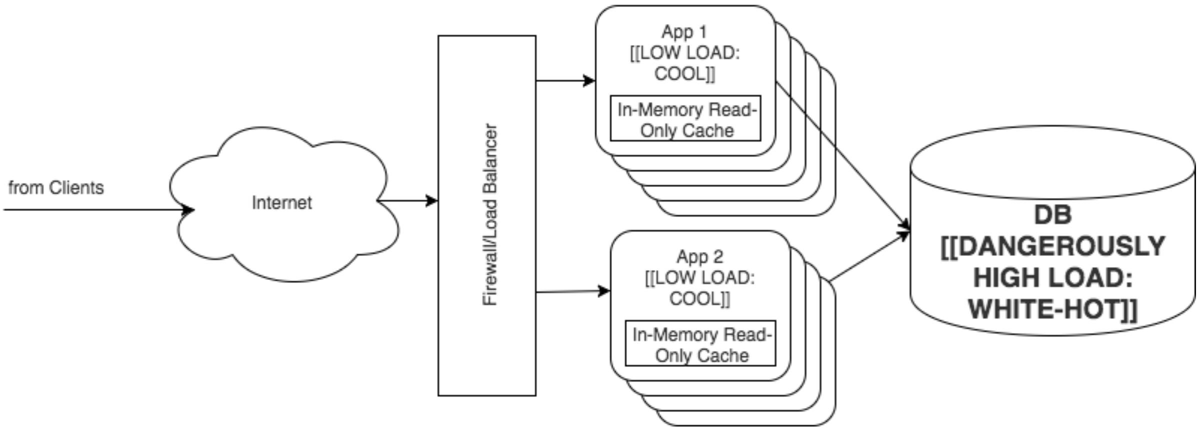
Scaling System Based on Disposable-Stateful-Apps

As mentioned above, in some cases (in particular, for stock exchanges) there is a firm requirement to have *all* the modifications to the state of our system Durable (which means that *all* modifications should go to DB, there is no way around it).

In such cases, and if the latencies are important – a kind of "Disposable-Stateful-Apps" can be used. The point here is to have a more-or-less usual Stateful App, but in this case our Stateful App will be merely serving as a read-only cache for DB information; as a result – in a case of crash it becomes trivial to restore the data from DB (which in turn makes such Stateful Apps disposable (in Docker-speak – *ephemeral*)).

An example of such a system is shown in Fig 8.4:

Fig. 8.4



This approach is very similar to the one shown in Fig 8.1 (the one for Stateless Apps) – except that Apps are no longer stateless <wink />. However, while Apps in Fig 8.4 are Stateful – their state is merely a read-only app-level cache, so in case of App crash (or App relocation/creation) the state can be easily reconstructed from the Database. While this approach does not reduce DB load compared to purely Stateless Apps <sad-face />, it does improve latencies significantly (which can be a Big Plus for stock exchanges etc.).

BTW, in a certain sense this approach shares some ideas with Front-End Servers as discussed in Chapter 9 (in a sense, Front-End Servers can be seen as read-only caches of "master" state published by the source, too).

From scaling point of view, this model can be seen as an another "hybrid" between the Stateless-App and Stateful-App models. In particular, with such a Stateless-App-plus-In-Memory-Cache model:

- Like with Stateless-Apps, we cannot reduce DB load <sad-face />
- Like with Stateful-Apps, we do reduce latencies
- Like with Stateless-Apps, we do NOT sacrifice Durability.
- Like with Stateless-Apps, we can Load-Balance only the incoming requests (or players) – and there is no need to Load-Balance the Stateful-Apps (they're both disposable and interchangeable).

Choosing Stateful vs Stateless

With all the different scalability models discussed above, *and* - as we have observed - each of them has its own niche when it is The Right Thing To Do™, it would be nice to have a simple guideline to know where to start. Not pretending that I have a definite answer which will work in all the scenarios, from my experience, I'd say that the following qualifies as a reasonably good starting point for your analysis:

If limitations to Durability are not a concern, and we can save at least 3-5x of DB load by using In-Memory State – we should go for it!

- If limitations to Durability are not a concern, and we can save at least 3-5x of DB load by using In-Memory State – we should go for it! For anywhere sizeable project, headaches related to scaling DB under unnecessary load, are the worst ones you will have, so reducing the load by factor of 3-5x (and as we've seen above – it can easily go all the way to 10x-1000x) is an Extremely Good Thing™.
    - One obvious solution in this direction is to use Stateful Apps. This approach does work – but has quite a few complications
        - In a sense – when moving from Stateless Apps to Stateful Apps, we're trading DB scaling complications (which are typical for Stateless Apps) for App scaling complications (typical for Stateful Apps). From my experience, such a trade-off is well-worth it.
    - On the other hand, as discussed above, in some cases (in particular, if the game is not too fast) we can *both* reduce DB load, *and* avoid Stateful Apps (via using an In-Memory Write-Back Cache). Still, it is not a silver bullet (and won't really work for most of fast-paced games such as simulations)
- If 100% Durability is a requirement (such as for stock exchanges) – then the choice becomes less obvious.

- If optimizing latency is a requirement – some kind of Disposable-Stateful-Apps is likely to be necessary. Personally, I've co-architected a stock exchange on top of such Disposable-Stateful-Apps (which can be seen as being along the lines of a usual game, but with DB commits on each trader action) – and with a very significant success too
- If optimizing latencies is not really needed (which includes pretty much all polling architectures) – then a classical Client-Server web architecture (the one with Stateless Apps) will do.

Scalable Components for the Scalable System

One thing to keep in mind, is that all those beautiful architectures on Fig. 8.1-8.4, are only *potentially* scalable; to make them *really* scalable – we need to make sure that all their components are scalable.

In this Chapter, we won't go into too many details of scaling individual components, but instead will give a very high-level overview, with pointers to additional discussions.

As it was noted in the *What to Scale?* section above, in an MOG there are two traditional entities which need scaling – (a) Game Worlds, and (b) database. Let's take a very cursory look at each of them.

Scaling Game Worlds

When speaking about scaling Game Worlds – making them scalable depends on two factors: (a) nature of the game, and (b) which of scalability approaches shown on Fig. 8.1-8.4 is used.

First, let's consider those games which consist of many rather small and well-isolated matches (areas, …); examples of such games include many of First-Person Shooters and MOBAs. For such games, as a rule of thumb, the game will be naturally scalable even if we're forced to use one of Stateful approaches (the one on Fig. 8.2). For such games, the smaller the matches are – the easier it will be to Load-Balance the whole thing; however, with an "average" game being able to handle about 1000 players per server box (~=100 players/core), *and* typical match/area being limited to 10 players or so – gives us a hundred of causing-mostly-the-same-load-matches, and under these circumstances Load-Balancing rarely causes any trouble.

When this "rather small and well-isolated" requirement doesn't stand (which is almost-universally the case for "seamless" MMOs) – we'll likely need to have Stateful processing too. As a result - we'll likely need to do some advanced Load Balancing (see, for example, discussion in Vol. I's chapter on Communications, [TODO: Baryshnikov], and [TODO:Beardsley] for discussion on scaling of such "seamless Game Worlds").

Scaling Matchmaking Server

In some cases (="if we managed to become Really Large™" <smile />), we may need to scale our Matchmaking Server too. From what I've seen in this regard – it really becomes too big of a problem, but – unfortunately, solutions tend to be way too game-specific to discuss them in a general way.

Scaling Database

The second big thing we'll need to scale – is database. We'll discuss scaling DB in nauseating detail in Vol. VI's chapter on Databases, so here I'll provide only a very high-level outline:

- The only part of the DB which needs to be ACID-compliant – is OLTP DB (the one where all the operational decisions are made).
    - As a rule of thumb, one single instance it can be scaled up to 100 transactions/second fairly easily, and up to 1000 transactions/second – with a significant-but-doable effort.
    - If going beyond (very roughly) 1000 real-world writing ACID transactions/second – as a rule of thumb, we *will* need to take significant effort to ensure scalability. In Vol. VI, we'll discuss how it can be achieved in a perfectly-linear-scalable Shared-Nothing manner (very briefly – it will be based on Inter-DB Asynchronous Transfer algorithm discussed in Vol. I's chapter on Communications); we'll also discuss a way to perform *gradual* migration from simple single-write-connection DB to a full-scale perfectly-scalable multi-node one (and it worked like a charm in practice too).
- As for the tons of non-operational read-only requests to the database (including both reports-for-support *and* all kinds of analytics) - they can and should be satisfied from asynchronous replicas.[44] While organizing such replicas (especially heterogeneous ones) is quite a headache – it is perfectly doable; more on it in Vol. VI.
    - And for read-only requests – they scale really trivially, just by adding yet another replica when it becomes necessary; once again – this is Shared-Nothing scaling, so the system scales in an almost-linear manner.

And as soon as we scaled *both* OLTP DB *and* replicas – we have our whole DB perfectly scalable; moreover, we made it a perfectly-scalable Shared-Nothing manner <sic! />.

Load Balancing

One topic which is very closely related to scalability, is load balancing. Even if our game is *scalable*, it only means an *ability to scale.* To make sure that our system exercises this ability – we usually need to distribute/balance the load across the different cores/server boxes/datacenters. This (not really surprisingly) is known as Load Balancing.

[44] ideally – heterogeneous replicas, as some of requests are better performed from traditional RDBMS, and some – from NoSQL

Two flavors of Load Balancing

In web world, Load Balancing is usually understood just as “how to distribute Clients across web servers”. For multiplayer games, this kind of balancing may also be present; in particular – the architectures shown on Fig. 8.1, Fig 8.3, and Fig 8.4 include Load Balancing.[45] Moreover, as we’ll see in Chapter 9 (section on Front-End Servers), for the architectures such as the one shown on Fig. 8.2, such Clients-to-Servers Load Balancing *may* also be necessary (in particular, if you want to allow spectating, and spectators are distributed unevenly across your Game Worlds).

For Stateless (more precisely – *ephemeral*) request handlers, Clients-to-Servers Load Balancing is all we’ll ever need. However, as we already briefly noted above, soon as we introduce Stateful Objects, we’ll be facing a very different flavor of Load Balancing – namely “how to distribute Game Worlds across different Server boxes/CPU cores”.

Such Worlds-to-Servers Load balancing arises because we need to distribute our Game World Server across CPU cores/Server boxes/etc. – *and*, our objects being Stateful, we cannot move them as easily between cores/boxes as we’d like to. Worlds-to-Servers Load Balancing is ubiquitous for games – and is conceptually significantly different from Clients-to-Servers Load Balancing.

Let’s take a closer look at both of these different flavors of Load Balancing.

Load Balancing of Stateful Objects. Worlds-to-Servers Balancing

As noted above, Worlds-to-Servers Load Balancing becomes necessary for Stateful architectures (such as the one shown on Fig 8.2). Most of the time, Worlds-to-Servers Load Balancing, while being quite unusual (especially for those coming from web background), is not too difficult.

In fact, for quite a few games out there, it is performed by Matchmaking Server. *If* our game is essentially a series of independent matches (or some other independent encounters) – then our Matchmaking Server can:

- Keep track of the load on different Server boxes
    - In the simplest case, this can be a number of currently running matches on different Server boxes. Moreover, however naïve this approach may look - I’ve seen it to work in real-world; actually it is not too bad as long as all the matches consume about the same amount of CPU and RAM, which is often the case.

[45] note, however, that “Load Balancer” box on these Figures is *optional*; as we’ll discuss in [[TODO]] section below – Load Balancer as a box-sitting-before-our-Servers is just one of the ways to implement Load Balancing (and usually is *not* my favorite one)

        - Note that for this to work, you'll need to report when the match ends, back to the Matchmaking Server (while creation of the match is known to Matchmaking by definition, the end of the match is not that obvious).
        - This method tends to work well as long as all Game Worlds are inherently similar, and number of Game Worlds is large (i.e. at least several dozens of them running per Server)
    - In a more elaborate case, we can make our Server boxes (or more precisely – some kind of agent running on the Server boxes) report back to Matchmaking Server with data about CPU etc. In practice, these systems are somewhat more difficult to manage than simpler ones based on number-of-Worlds. This happens, in particular, due to an inherent delay between action (Game World creation) and reaction (an update on CPU load) in such systems, we need to be careful to avoid self-induced oscillations (which, in theory, *may* lead us all the way up to the stability analysis according to Nyquist Stability Criterion – though usually reducing the delay by more frequent reporting of the load is sufficient for Load Balancing purposes).

Nyquist Stability Criterion

https://en.wikipedia.org/wiki/Nyquist_stability_criterion

In control theory and stability theory, Nyquist stability criterion is a graphical technique for determining the stability of a dynamical system.

- When creating a new instance of the Game World, Matchmaking Server can take this per-Server-box load into account, with the possible actions being at least the following:
    - Create new Game World on the least loaded Server box
    - If all the Server boxes are loaded – requesting a new Server box from cloud provider.
        - When time of procurement of the new Server box is long enough (which is especially the case for "baremetal servers", which are required for fast-paced games, more on it in Vol. VII's chapter on Preparing to Launch) – we'll need to request the Server *anticipating* the growth of the load; in practice – keeping one "spare" server box at all times is usually enough, but YMMV.
    - In extreme cases – suspending/delaying creation of less important games (while this *should not* normally happen, "never say never" adage applies to real-world deployments in spades).

A completely separate Load Balancer (which may include moving Game World instances around) – is also possible. Such schemas are of particular importance for Load Balancing of "seamless" MMO worlds (for a brief discussion on seamless worlds – Vol. I's chapter on Communications, and also [TODO:Beardsley] and [TODO:Baryshnikov]).

Clients-to-Servers Load Balancing

As mentioned above, Clients-to-Servers Load Balancing is necessary for Stateless and kinda-Stateless architectures (such as those on Fig 8.1, Fig 8.3, and Fig 8.4).

In addition, Clients-to-Servers *may* be introduced for really Stateful architectures (such as the one on Fig. 8.2) – for example, via Front-End Servers (more on them in Chapter 9). For configurations of Stateful objects combined with Front-End Servers – we need *both* Worlds-to-Servers Load Balancing *and* Clients-to-Servers Load Balancing; however – in spite of additional complexity, such things can solve a problem or three in quite a few real-world scenarios (including, but not limited to, inherently better resilience to DDoS, and better handling of spectating with unpredicted spectating patterns – more on benefits of Front-End Servers in Chapter 9).

Clients-to-Servers Load balancing in general is quite a big topic at least over last 20 years. Three most common and distinct techniques out there are the following: DNS Round-Robin, Client-Side Random Balancing, and Server-Side (usually hardware-based) Load Balancer Appliances. With the industry producing those Load Balancer Appliances making good money on them - there is no wonder that they will keep explaining that it is The Only Viable Option™ (and they succeeded with convincing most of IT industry about it too). Still, let's take a closer look at available load balancing solutions.

DNS Round-Robin

DNS Round-Robin is based on a traditional DNS requests.

First, a very short intro into DNS. Regardless of any DNS Round-Robin in place, whenever a Client requests address *frontend.yoursite.com* to be resolved into IP address, Client's PC (console, etc.) sends DNS request[46] to your (or "your DNS provider's") DNS server. Your DNS server translates *frontend.yoursite.com* into IP address – and sends it back to the Client; now Client has an IP address which can be used for further communication.

Enter DNS round-robin. If your DNS server is configured for DNS round-robin, it simply returns different IP addresses to different DNS requests, in a round-robin fashion[47] hence the name.

[46] this happens with or without DNS round-robin

[47] strictly speaking, it is a little bit more complicated than that, as DNS packets contain a list of servers. However, last time I've checked - virtually everybody out there ignored all the entries in returned packet except for the very first one, so it became 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"

one of these returned IPs can get cached by a Big Fat DNS server, and then get distributed to many thousands of Clients

It all looks very simple and nice on paper – but in practice DNS Round-Robin suffers from two major problems. First, there is a problem with caching DNS servers along the path of the request (and DNS caching happens all the time). That is, even if your DNS 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 your 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 <sad-face />. The second big problem of DNS Round-Robin, is that it lacks server fault tolerance; in other words - if one of your servers is down, a Client which relies purely on DNS Round-Robin (such as web browser), won't try another server on the list, leading to a service interruption for at least some of your players <sad-face />.

Fortunately, as for MOGs we DO have our own Client (and usually do not need to rely on functionality of the web browser), we can solve both these problems quite 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). Enter Client-Side Random Balancing.

Client-Side Random Balancing

To improve on DNS Round-Robin, a very simple approach can be used:

- We won't rotate/round-robin anything on the Server-Side; instead, we will distribute *exactly the same* list of Server IP addresses 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 use - doesn't matter much to us now, but we'll discuss "DNS-vs-hardcoded-IPs" in Vol. VII's chapter on Preparing to Launch
- 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 (and whenever connection is lost, etc.) - Client gets another random IP from the list and tries connecting again.

Client simply takes a random item from the IP list, and tries connecting to this randomly chosen IP.

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

```
int myrand() {//DON'T DO THIS!
  srand(time(0));
    //NOTE: crypto PRNG seeded with time(0) at this point
    //  will NOT be an improvement!
  return rand();
}
```

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 choose their IP addresses based on *myrand()* function; if all the Clients attempt to re-connect within 5 seconds after the disconnect, with such a bad *myrand()* function you'll get merely 5 different IPs. Other than such extremely bad cases, pretty much any RNG should be fine for this purpose. Even a trivial and very-poor linear congruential generator such as *srand()*,[48] seeded with *time(0)* at the moment when the program was launched (but NOT at the moment of request, as in example above), should do in practice, though adding some kind of milliseconds or some other Client-specific data to the mix is advisable "just in case".

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, though in practice it fails badly due to Big Fat DNS caches), Client-Side Random Balancing relies on the statistical Law of Large Numbers to achieve flat distribution of Clients between the Front-End Servers. What the law basically says is that for independent measurements, the more experiments you're performing - the more flat distribution you'll get.

Law of Large Numbers

https://en.wikipedia.org/wiki/Law_of_large_numbers

According to the law, the 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.

More formally, probability distribution of Server load for Client-Side Random Balancing is binomial; among other things, it means that standard deviation of this distribution is proportional to $\sqrt{N}$, and relative standard deviation is proportional to $1/\sqrt{N}$. In other words, if we multiply number of Clients by 100 – then standard deviation increases 10-fold, but relative standard deviation *decreases* by factor of 10. For example, if we have 100 Clients randomly balanced to 10 Servers, we'll get 10 ± 3 Clients per Server (which means that imbalance of load between Servers is of the order of 30%); however, if we have 10000 Clients randomly balanced to the same 10 Servers – we get 1000 ± 30 Clients per Server, reducing imbalance of the load between Servers to much more acceptable 3%.

In practice, despite being "non-ideal" in theory, Client-Side Random Balancing achieves much more flat distribution than DNS Round-Robin one. The reason for it is two-fold. First, as discussed above, as soon as the number of Clients is large (at least a few hundred), Client-

[48] honestly, I prefer NOT to use *srand()* and linear congruential at all – preferring simple crypto-PRNGs such as AES-CTR across the board – but here it is beyond the point.

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 any valid caching (unlike with DNS Round-Robin) doesn't change Client distribution at all.

Also we need to mention that in contrast to DNS Round-Robin, Client-Side Random Balancing can be easily made fault-tolerant (with respect to failures of balanced Servers): whenever Client detects that Server is down (for discussion on how to do it - see Vol. IV’s chapter on Network Programming) – then Client merely reconnects, using a different random IP from the list it has.

To summarize: personally, I would not use DNS Round-Robin for production Load Balancing (in particular, because of lack of fault tolerance). 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.[49]

Server-Side Load Balancer Appliances

An approach which is very different from both Round-Robin DNS and Client-Side Random Balancing, is to use Server-Side Load Balancer Appliances. Load Balancer Appliance is usually an additional box, sitting in front of your servers, and doing, as advertised, Load Balancing.

Server-Side Load Balancer Appliances 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).

In addition (similar to Client-Side Load Balancing), Server-Side Load Balancer Appliances do provide fault tolerance with regards to the balanced Servers. On the other hand, being a Single Point Of Failure (SPOF)[50] – they need to be fault tolerant themselves, and fault tolerance at this level is well-known to cause quite a bit of trouble <sad-face />. In particular, all the discussion in Chapter 10 with respect to failure of heartbeat link – applies to Server-Side Load Balancer Appliances in spades (and from what I’ve seen, these appliance boxes tend to be negligent when it comes to analysis of such failures, which – alongside

[49] 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 didn’t see it necessary in practice

[50] NB: with Client-Side Load Balancing, there is no SPOF at all(!)

with infamous “misconfigured failover scripts” - tend to cause a lion share of failures in real-world supposedly-fault-tolerant systems).

Now let’s compare Server-Side Load Balancer Appliances with the Client-Side Random Balancing. As noted above – the list of disadvantages of Load Balancer appliances is pretty long:

(a) pricing (these boxes tend to be Damn Expensive – that is, unless you’re using HAProxy, but see below about it itself becoming a bottleneck),[51]

(b) such an Appliance is an inherent Single Point of Failure a.k.a. SPOF; this means that it itself needs to be made fault-tolerant (and fault-tolerance is rarely handled good enough with these boxes, see above)

(c) configuration and operational issues with these boxes (especially “failover scripts”, see above) tend to be significant.

(d) Last but certainly not least - as all the relevant traffic needs to go via such an appliance – the load-balancing appliance itself can potentially become a bottleneck (and then the only way to scale will be scaling up, which is not really feasible). While this doesn’t look *too likely*, it can happen: while such appliances can easily handle multiple gigabits/second – their ability to handle tons of small packets is usually significantly lower (in other words – when speaking about games, you should look for packets/sec or requests/sec rather than for GBit/sec). And while HAProxy’s reported 100K requests/second may sound as a lot for a website – for a 500K-simultaneous-players game it is pretty much nothing. Moreover, while Scaling Out by adding a few other servers won’t usually be a problem for such a game – Scaling Up your Load Balancer by finding more powerful one can become Really Nasty <sad-face /> (see *Scaling Up – Doesn’t Help Much for Game World Servers* section above for a discussion – exactly the same logic applies to Load Balancer Appliances too). For highly-loaded games, if using Load Balancer appliances – your best bet will be ASIC-based ones, but while they will provide you with necessary performance and reliability (with a lower latency on top) – they will cost you arm and leg (and probably even more); while if your monetization is good – the cost of the order of quarter a million dollars for a pair of such boxes isn’t likely to kill your game, but IMO it is still *much* nicer to save them for your company (and maybe even get a bit of bonus for doing it <wink />).

[51] 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 Balancer Appliances in an 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 an active/active configuration, then you still need to have 100% redundancy to be able to cope with the load if one of them fails <sad-face />

On the pro side for Server-Side Load Balancer Appliances, there are better load balancing capabilities (i.e. beyond relying on the Law of Big Numbers). However, from what I've seen, these additional balancing capabilities are usually unnecessary for games (where Law of Large Numbers tends to stand very firmly). This makes these four cons above a deciding factor why I usually recommend to stay away from Load Balancer Appliances – at least in the context of games.

Additional balancing capabilities provided by Load Balancer Appliances, are usually unnecessary for games (where Law of Large Numbers tends to stand very firmly)

To summarize: for game load-balancing purposes I didn't see practical use cases for Server-Side Load Balancer Appliances (as always, YMMV and batteries are not included). One exception: if you're using Web-Based Deployment Architecture (in the way described above) – then HAProxy MIGHT be able to shift the balance towards HAProxy-based Load Balancer Appliances. First - HAProxy is free software running on commodity server boxes, which removes the "damn expensive" argument. Second, while for already-loaded JS code it is perfectly possible to use Client-Side Random Load Balancing, it is not the case for original HTML+JS – and HAProxy will address this problem too. Still, even for Web-Based architectures I'd seriously consider using HAProxy for static content (="to load initial HTML+JS"), and Client-Side Random Balancing for game traffic (in particular, because of significant problems of naïve heartbeat-based fault tolerance configurations, see discussion above) – but this choice admittedly is not as clear as for non-web-based (and especially UDP-based) games.

Load Balancing Summary

From my experience, Client-Side Random Balancing worked really good, and I didn't see any reasons to use something different. Round-Robin DNS is almost universally inferior to Client-Side Balancing, and hardware-based Server-Side Load Balancer Appliances are too complicated and expensive, usually without any real reason to use them at least in MOG environment. As noted above, one exception when you MAY need Server-Side Balancers, is if you're using Web-Based Deployment Architecture, but beyond purely static content even this is debatable.

And 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 and way too game-specific to discuss them in this book.

Bibliography

Baryshnikov, Maksim. n.d. "Engineering Decisions Behind World of Tanks Server."

Beardsley, Jason. n.d. "Seamless Servers: The Case For and Against." In *Massively Multiplayer Game Development*.

Bray, Brandon. n.d. *The .NET Framework 4.5 includes new garbage collector enhancements for client and server apps.* https://blogs.msdn.microsoft.com/dotnet/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps/.

Corbet, Jonathan. n.d. *"NUMA scheduling progress".* https://lwn.net/Articles/568870/.

Cybersource. n.d. *"Linux vs Windows. Total Cost of Ownership Comparison".* https://static.lwn.net/images/pdf/cybersource-tco-study.pdf.

n.d. *DPDK.* http://dpdk.org.

Duquette, Patrick. n.d. "6.2 Implementing a Seamless World Server." In *Game Programming Gems 5*.

IDC. n.d. *"Windows 2000 Versus Linux in Enterprise Computing".* https://www.cetic.be/IMG/pdf/TCO.pdf.

n.d. *Introduction to Receive Side Scaling.* https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/introduction-to-receive-side-scaling.

Klitzke, Evan. 2013. *Migrating Uber from MySQL to PostgreSQL.* https://www.yumpu.com/en/document/view/53683323/migrating-uber-from-mysql-to-postgresql.

—. 2016. *Why Uber Engineering switched from Postgres to MySQL.* https://eng.uber.com/mysql-migration/.

Lameter, Christoph. n.d. *"NUMA (Non-Uniform Memory Access): An Overview".* https://queue.acm.org/detail.cfm?id=2513149.

Lightstreamer. n.d. http://www.lightstreamer.com/.

Ligoum, Dmitry. n.d. "private communications with."

n.d. *London Stock Exchange gets the facts and dumps Windows for Linux.* http://www.itwire.com/opinion-and-analysis/the-linux-distillery/28359-london-stock-exchange-gets-the-facts-and-dumps-windows-for-linux.

n.d. *netmap - the fast packet I/O framework.* http://info.iet.unipi.it/~luigi/netmap/.

n.d. *New techniques to develop low-latency network apps.* https://channel9.msdn.com/Events/Build/BUILD2011/SAC-593T.

'No Bugs' Hare. n.d. *"Memory Leaks and Memory Leaks".* http://ithare.com/memory-leaks-and-memory-leaks/.

Noyes, Katherine. n.d. *"Five Reasons Linux Beats Windows for Servers".* http://www.pcworld.com/article/204423/why_linux_beats_windows_for_servers.html.

n.d. *Predicting the Performance of Virtual Machine Migration.* https://www.cl.cam.ac.uk/~sa497/akoush-mascots10.pdf.

Redis.CAS. n.d. http://redis.io/topics/transactions#cas.

RFG. n.d. *"TCO for Application Servers: Comparing Linux with Windows and Solaris".* http://www-03.ibm.com/linux/whitepapers/robertFrancesGroupLinuxTCOAnalysis05.pdf.

n.d. *Scaling in the Linux Networking Stack.* https://www.kernel.org/doc/Documentation/networking/scaling.txt.

StackOverflow.C#LambdaLoop. n.d. *"Captured variable in a loop in C#" where="StackOverflow".* http://stackoverflow.com/questions/271440/captured-variable-in-a-loop-in-c-sharp.

StackOverflow.PythonLambdaLoop. n.d. *"What do (lambda) function closures capture in Python?".* http://stackoverflow.com/questions/2295290/what-do-lambda-function-closures-capture-in-python.

Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli. n.d. "Architectural Breakdown of End-to-End Latency in a TCP/IP Network."

Verma, Abhishek. 2016. *Cassandra on Mesos Across Multiple Datacenters at Uber (Abhishek Verma).* C* Summit 2016. https://www.slideshare.net/DataStax/cassandra-on-mesos-across-multiple-datacenters-at-uber-abhishek-verma-c-summit-2016.

Wikipedia. 2017. *Cluster Launch Failure.* https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure.

Zubek, Robert. n.d. *"Engineering Scalable Social Games".* http://gdcvault.com/play/1012230/Engineering-Scalable-Social.

—. n.d. *"Private communications with".*

Chapter 9. Server-Side Architecture

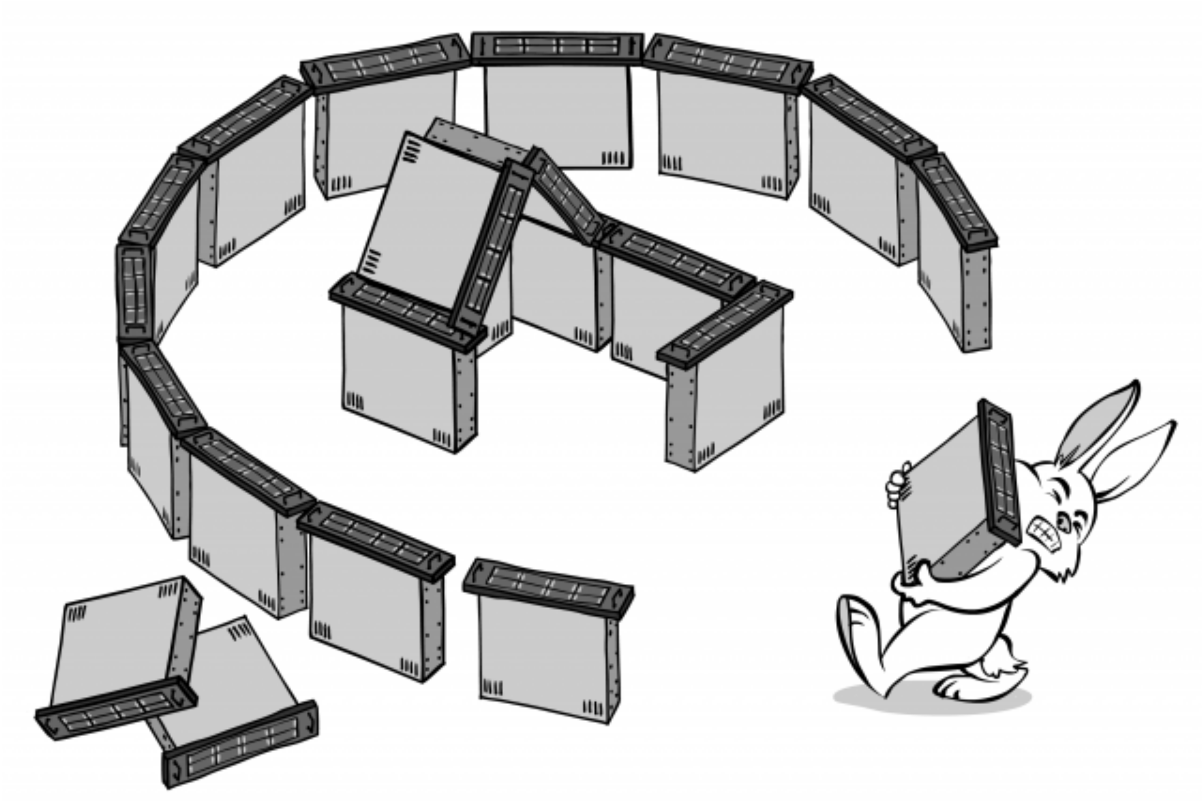
After we're done with discussing Scalability (and closely related issues of In-Memory State), we can start drawing our architectural diagrams for the Server-Side. And the very first thing we need to do in this direction, is to start thinking in terms of

"how we're going to deploy our Servers, when our game is ready?"

Yes, I *really* mean it – overall architecture starts certainly *not* in terms of classes, and for the Server-Side – not even in terms of processes or (Re)Actors. Rather, 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.[52] 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. 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.

Worst of all, there are *lots* of reasons why your Server-Side can fail badly (including such very different things as poor Scalability, lack of database coherency, failing PCI DSS audit, being wide-open to cheaters, and poor code maintainability) – and it is our job as architects to make sure that *all* the potential caveats are accounted for; sure, it is not always possible to handle *all* the troubles in advance (it is especially true as we're time-pressed by definition) – but at least we should *try*.

[52] Strictly speaking, for larger games – there is even-higher-level diagram, the diagram showing interactions between different datacenters; however – until Vol. IX we're not there yet, so for the time being we'll keep our deployment datagrams to single datacenter only.



Last thing before we start: let's note that for your very first deployment, you may have much less physical/virtual Server boxes than shown on the diagrams below; in practice – you're likely to combine apps from quite a few of these Server boxes together. On the other hand, you should be able to increase the number of your Servers quickly, so you need to have the software which is able to work in the deployment-architectures-shown-below, from the very beginning. This is important, as demand for increase in number of Servers can develop very quickly if you're successful. As for details of your very first deployment – we'll discuss them in Vol. VII's chapter on Preparing for Deployment.

Now, we can start our discussion about different deployment architectures; more specifically - we'll start with a deployment architecture you SHOULD NOT aim for.

Don't Do It: Naïve Game Deployment Architecture

Quite often, when faced with development of their very first multi-player game, developers start with something like the following Fig 9.1:

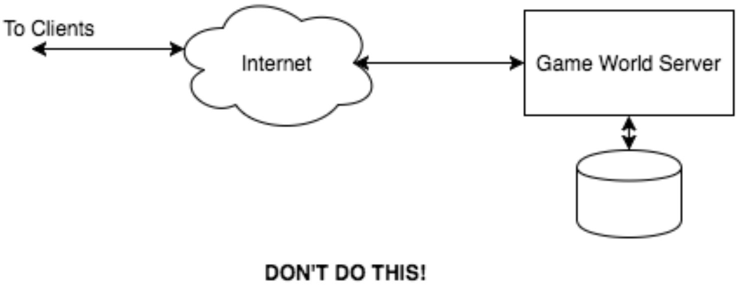


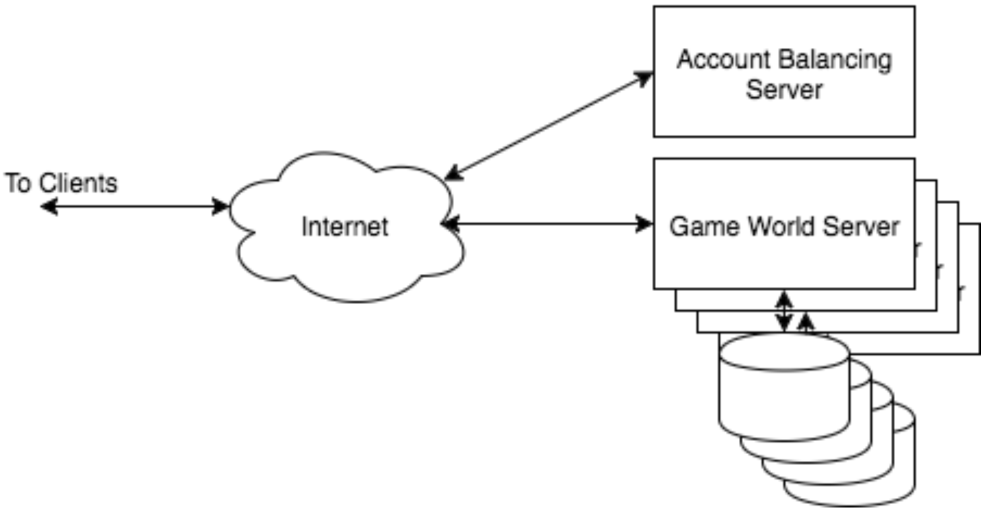
Fig VII.1

[[TODO/fig: VII.1 -> 9.1]]

It is dead simple: there is a Game World Server, and there is a database to store persistent state. However, there is a ticking bomb within this simplistic (and admittedly extremely straightforward) approach.

Later on, as one single Game World server proves to be insufficient to run all the players – the architecture above often naturally evolves into something like the diagram on Fig 9.2:

Fig. VII.2



[[TODO/fig: VII.2 -> 9.2; add "DON'T DO THIS EITHER!"]]

Here, in a naïve attempt to scale out a simplistic architecture shown on the diagram on Fig 9.1, each of the Game World Servers is just cloned (including its own database(!)). It means that we have several Game World Servers, each with its own database (completely independent from all the other databases), and each of the players get *permanently* assigned to one of Game World Servers. To handle these assignments – often (though not universally) an Account Balancing Server may be added.

If it is present, the logic of Account Balancing Server in such naïve-and-NOT-recommended architectures usually works as follows:

- First time players come to Account Balancing Server – which simply forwards them to a Game World Server.
    - There are different ways to do it – but give or take, it is usually some kind of "sharding". Whether you're merely pushing all new accounts to each new Server until it becomes full, or are using real sharding such as the one based on a hash of user-ID – doesn't matter much for our discussion now.
- From this point on[53] – player becomes permanently assigned to the respective Game World Server (and all the player's data is stored within Game World Server's DB).

[53] Actually – the assignment actually happened even earlier, as soon as the player decided on his user-ID.

- In particular, it means that the *player cannot possibly play with anybody-except-for-those-who-happened-to-be-assigned-to-the-same-Game-World-Server(!)*

My word of advice about such naïve deployment architectures:

DON'T DO THIS!

In the long run, such a naïve approach won't work well for most of the games out there; moreover, even if it works at first – more likely than not, it will create severe obstacles to your marketing/monetization efforts later.

To demonstrate problems which are inherent to such simplistic architectures – let's start with an all-important observation:

You game *will* need inter-player interaction just because two players decided they want to play together. If not now, then a bit later.

We already discussed this issue in Vol. I's chapter on GDD, so I'll provide only a quick recap here. The point here is that (a) most of multiplayer games require at least some kind of socializing with real-world people, and (b) as soon as we need real-world socializing – even the simplest "I want to play with my Facebook friend" feature means that more often than not, players will want to play with each other *just because they feel like it*.

To implement this inter-player interaction just-because-players-feel-like-it – the architecture on Fig 9.2 is *woefully inadequate*; in short – you just won't be able to implement it, plain and simple. With an architecture on Fig 9.2, when[54] you need to implement even that simplest feature of letting player to play with her Facebook friends – you will end up in producing all kinds of really weird solutions, including, but not limited to: (a) restricting arbitrary inter-player interactions, replacing real-world interactions with the inter-Server pseudo-communities (which in turn makes "invite Facebook friend" feature pretty much useless, with all the potential social viral effects going out of the window); (b) inter-Server player transfers (doesn't help much as soon as more than two players are involved – not to mention ID collisions and all kinds of weird things on the way); and (c) trying to *guess* which players are likely to play together (which is a pretty much a non-starter, but those guys were *really desperate* in trying to save their naïve-and-unworkable architecture).

The problem here lies in an observation that as soon as it is players who decide who they want to play with, (a) players and (b) Game Worlds MUST NOT be hard-tied together (otherwise choosing who-I-want-to-play-with right now, won't be possible). As a result - any simplification which forces these two very different things together – more often than not, becomes a surefire way to a disaster. I've seen such naïve architectures causing significant problems even for a farm-like game (and farming games on the first glance *seem* to be a very natural fit to such simplistic architectures) – and it only gets worse for any other game genre.

[54] Not "if"!

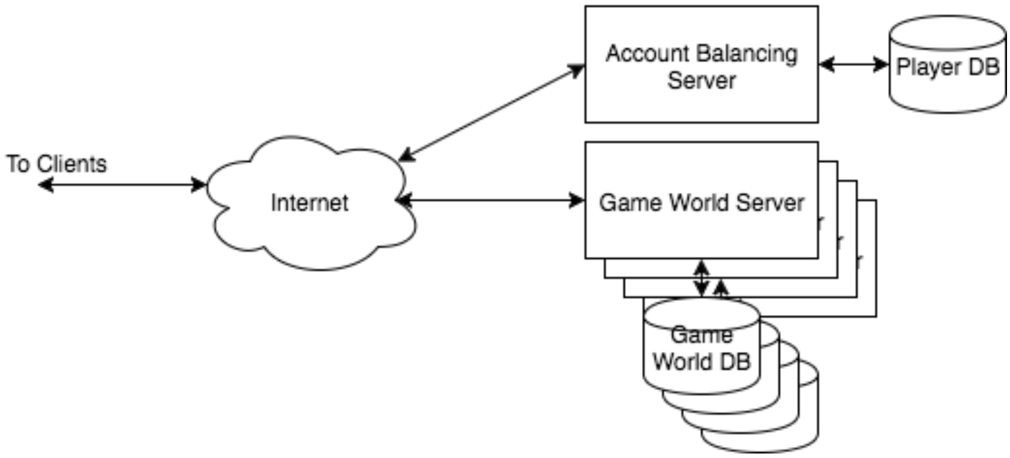
However, while taking a firm stand against Naïve Architectures, I need to make two all-important clarifications:

- The problem with the naïve approach shown on the diagrams above, is all about having c*ompletely separate user DBs* (which essentially tie your players to Game Worlds). If you change Naive Architecture to have not only a per-server DB, but also a centralized player DB - say, with per-Server DB storing Game World State of the players which are *currently* residing in this Game World, and with centralized player DB storing *all* the players - you *can* get a rather viable architecture; more on it in "Semi-Naïve Deployment Architecture" section right below.
- If your game calls for natural restrictions on the Server/Datacenter locations (for example, due to latencies you're going to have separate "NY Server", "LA Server", "London Server", and so on) – you MAY have completely separate user DBs per location/Datacenter (as players from different locations cannot play with each other by design).
    - However, in this case – user database SHOULD NOT be per-Server-box, but rather SHOULD be per-location/Datacenter (and each location/Datacenter, in spite of being colloquially named "Server", will have at least a few dozens of physical Server boxes); the point is that having a separate user DB per each such Server box is a Bad Thing™. In other words – in such cases, we're essentially speaking about having Classical (or at least Semi-Naïve) Deployment Architecture within each of Datacenters – and cloned for each of the Datacenters involved.
    - In addition, even in this case, you SHOULD take an effort to ensure that at least userIDs are *guaranteed* to be unique across different DBs. This is not a problem for internal IDs (as we can always say that globally-unique-ID is a tuple (Datacenter_ID,userID_within_Datacenter)), but if your game relies on players being able to identify each other by their visible ID/name – then enforcing such a globally-unique-ID can become a significant headache.

Semi-Naïve Deployment Architecture

As noted above – the main problem with Naïve architectures is related to creation of a permanent bond between player and Game World. This can be avoided by using the following "Semi-Naïve" architecture is shown on Fig 9.3:

Fig. VII.3



[[TODO/fig: VII.3 -> 9.3]]

Here, the only apparent difference from Fig 9.2, is that Account Balancing Server gets its own "Player DB". However, this seemingly small difference will cause a lot of changes at all the levels of your system – and more importantly, unlike Naïve Architecture shown on Fig 9.2, it does allow to build viable and scalable games.

In the Semi-Naïve Architecture shown on Fig 9.3, the idea is to keep our players in "Player DB" – and move them (or more strictly – some of their attributes, such as "in-game money" or "health", but usually not attributes such as "payment history") only temporarily to that Game-World-DB where the player is *currently* playing.[55] NB: when speaking about "moving" the attributes to Game-World-DB – usually, we'll be actually copying them, while marking them as "allocated" in main Player DB; this will allow to recover (at least to some extent) if one of the Game World DBs crashes.

While I am still advising against such Semi-Naïve Architectures (preferring instead a separate DB Server as discussed below in "Classical Game Deployment Architecture" section) – I have to say that Semi-Naïve architectures such as shown on Fig 9.3 (and unlike naïve ones discussed in "Don't Do It: Naïve Game Deployment Architecture" section above) MAY be made viable.

In particular – Semi-Naïve architectures *look* conceptually very similar to the perfectly-viable (and *highly* recommended in the longer run) Shared-Nothing DB Scaling discussed in detail in Vol. VI's chapter on Databases; however – there is a significant difference between the two. For those perfectly-viable Shared-Nothing DB Scaling architectures discussed in Vol. VI – there is usually *much* more than one "Game World Server" for each of "Game World DBs"; in fact – according to Vol. VI, the whole process of separating DBs should start *only* when the need arises (and, if you do things properly, this is not going to happen until you have *at least* 100K simultaneous players). On the other hand, having a separate DB for each of

[55] Note that for some games (think casino), player can play in several Game Worlds simultaneously; in such cases – only a *part* of the player's account (as in "$50-which-he-wants-to-play") is moved to a specific table.

physical boxes, while theoretically possible, happens to be too much trouble in practice: Server-boxes-which-run-DBs tend to be much more critical than no-DB Server boxes, they have to be backed up, you should have a non-trivial plan of “what to do if the DB Server box fails”, and so on. As a result – I still to NOT recommend going *exactly* along the lines of Fig. 9.3 from the very beginning.

Let’s take another look at the same thing from a bit different angle. Overall, a Semi-Naïve Architecture on Fig 9.3 is a Shared-Nothing system – and I’m consistently arguing *for* Shared-Nothing architectures. It just so happens that Classical Deployment Architecture also can be split into Shared-Nothing boxes (along the lines discussed in Vol. VI’s chapter on Databases) – but at the same time Classical one tends to provide better layering and better evolution path (dealing with problems as they start to bite, not earlier). Also - usually, when using Share-Nothing DB architectures, it still happens to be better[56] to have several Game World Servers per one non-user-DB-Server (and that’s what we’ll get when a Classical Deployment Architecture evolves along the lines discussed in Vol. VI’s chapter on Databases), but in theory even 1:1 relation (as with a Semi-Naïve Architecture shown on Fig. 9.3) can potentially fly – that is, as long as you have that centralized Player DB.

Bottom line on Semi-Naïve Deployment Architectures:

- Unlike Naïve one – Semi-Naïve Deployment Architecture MAY work
    - One all-important thing in making it work is to keep centralized user/player DB (which can be sharded if necessary), and avoid assigning players to Game Worlds on the permanent basis
- I am still not a big fan of Semi-Naïve Deployment Architecture (preferring Classical Deployment Architecture instead). In particular, Classical Deployment Architecture will usually provide:
    - better layering
    - better separation of concerns (including better separation between different teams)
    - more natural evolution path (with solving problems after they start to be felt – which in turn helps to solve them much better)
    - better performance

Web-Based Game Deployment Architecture (Web Stack)

Naïve and Semi-Naïve Architectures Aside – we can get to the real stuff. If your game satisfies two conditions:

- *first*, it is on the slow-paced side of things (in other words, it is not an MMOFPS) and/or "asynchronous" (as defined in Vol. I’s chapter on GDD, i.e. it doesn't need players to be present simultaneously),

[56] Performance-wise, and more importantly, maintainability-wise

- 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; a simple example of it is shown on Fig 9.4:

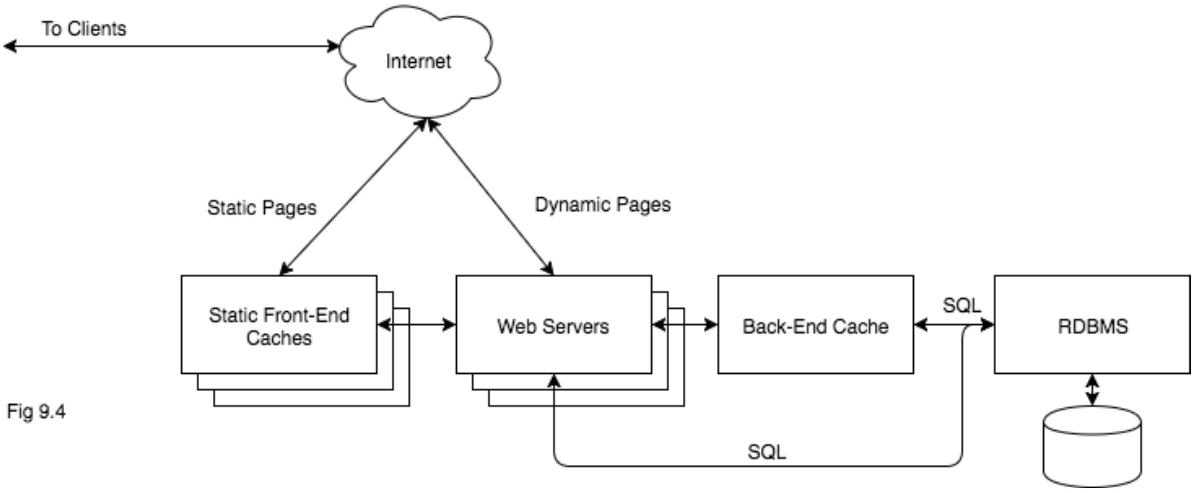


Fig 9.4

This diagram looks more or less as your usual web app, with a few added caches; this directly corresponds to “Web stack” as described in (Zubek, "Engineering Scalable Social Games" n.d.).

Web-Based Deployment Architecture: How It Works

As we can see, the diagram on Fig 9.4 looks pretty much 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 (to cache results of DB requests). However, if we take a closer look at it, we’ll find that there are also significant differences from classical web apps (special thanks to Robert Zubek for sharing his experiences in this regard, (Zubek, "Private communications with" n.d.)).

Peculiarities of the Web-Based Game architectures are mostly related to caching.

Caching, More Caching, and Even More Caching

To make the game perform well under anywhere-significant load, caching needs to be rather elaborated.

First of all, we can see that in Web-Based Game Architecture shown on Fig 8.4, both front-end caching and back-end caching is used. Front-end caching is pretty much your usual page caching (like *nginx* in reverse-proxy mode, or even a CDN); however, there is a game-specific caveat even here. 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 two different things (Zubek, "Engineering Scalable Social Games" n.d.):

a. your static assets (.SWFs, CSS, JS, etc. etc.) which almost-never change, and can (and should) be cached in a Front-End Cache, and

b. 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).

At the back-end, the situation is significantly more complicated. For games, you will often want not only to use your Back-End Cache as a cache to reduce number of DB *reads* (which is common for usual web apps), but also will want to make your Back-End Cache 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 (Zubek, "Private communications with" n.d.).

Taming DB Load: Write-Back Caches and In-Memory States

One Big Advantage™ of having write-back cache (and of the in-memory states in general) is related to the huge reduction in number of DB writes, and as it was discussed in Chapter 8, DB writes usually represent The Biggest Obstacle™ on the way to achieving scalability.

For example, if we'd need to save each and every click on the simulated farm with 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.

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 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.

And given that from the point of view of recovery from server failures and behavior which is usually required by players, write-back caches and in-memory states also tend to work very well (see discussion on "'No Bugs' Rule of Thumb" in Chapter 8) – I don't see any reasons not to use them <smile />.

Write-Back Caches: When to Write Back?

From the point of view of Web-Based Architectures, In-Memory States discussed in Chapter 8, are implemented as write-back caches. On the other hand, to be consistent with the logic discussed in Chapter 8 (and to provide good recovery from server crashes),

It is important not just to have write-back caches with *some* kind of "lazy writes" – but to have write-backs synchronized with the end of your "game events".

One important benefit of such write-back caches (the ones where you control write times yourself) is that they tend to play well with handling server failures (that is, as long as your game is one of those which require rollback in case if "game event" is disrupted).

Write-Back Caches: Locking

Ok, so we've established that in-memory states in general and write-back cache in particular, are Good Things™. Now it's time to discuss how to implement these write-back caches for the Web-Based Architecture we're currently considering.

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 (and well-known) approaches: "optimistic locking", "pessimistic locking", and transactions. Let's consider them one by one.

Optimistic Locking

Optimistic locking (as well as pretty much anything out there) can be implemented in different ways; however, here we'll discuss a specific implementation based on memcached's CAS operation.[57] The idea of using CAS for optimistic locking goes along the following lines.

Classical CAS (Compare-And-Swap a.k.a. Check-And-Set) is an atomic operation taking two parameters for a specific variable X: an "old" value and a "new" value. Then, CAS operation does the following:

- it Compares current value of the variable X with the "old" value supplied, and sees whether "old" value is the same as "current" one. Alternatively, we can say that it Checks that current-X == old-X

CAS

https://en.wikipedia.org/wiki/Compare-and-swap

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 that memory location to a given new value.

[57] a supposedly equivalent optimistic locking for Redis is described in (Redis.CAS http://redis.io/topics/transactions#cas), but in general, optimistic locking doesn't necessarily require CAS

- if “current” was found to be equal to “old”, then it Sets “current” value of X to be equal the “new” one (in academy circles, it is also commonly referred to as Swap).
- And last but not least, it tells the caller whether the assignment has happened.

The key point of CAS is that it performs all these things *atomically*. In other words, nothing can possibly happen between the comparison/check, and assignment. This simple guarantee allows to implement *lots* of different synchronization algorithms (in particular, all the critical section/no-locking algorithms in multi-threading are CAS-based); however, from our current perspective, we will consider only one thing: how to implement optimistic locking based on.

In *memcached*, CAS has an API which is a bit different from classical CAS, while the idea is still the same. With *memcached* CAS, to update our variable X (identified by its key) in an atomic manner, we need first to get the tuple of *(data, cas_token)* – usually this done by *gets* command; here *cas_token* is merely an opaque thing supplied by *memcached* (though it can be considered as a kind of “version number” for our *data*). Then, we can issue *cas* command, supplying a tuple of *(new_data, cas_token)*. *Memcached* will compare *cas_token* coming from the *cas* command, with the current value of *cas_token* associated with our piece of data, and will update the data (and increment associated *cas_token* too) if and only if the value of current *cas_token* associated with our data, is the same as the value of *cas_token* supplied in the *cas* command. On the other hand, if somebody has modified the data since you read it – the data will have different *cas_token* associated, so then our *memcached* transaction will fail (indicating that a “mid-air collision” has occurred).

Then, our CAS-based implementation of the optimistic locking will go along the following lines. To process some incoming request, our Web Server will do the following:

- receive the incoming request and realize which of Game Worlds it should go to.
- read whole Game World State as a single blob from *memcached*, alongside with *cas_token*.
- As it follows from the name of “optimistic locking”, we're optimists (for the time being that is <wink />)! As a result, our Web Server processes incoming request ignoring possibility that some other Web Server also got the same Game World and is working on it
- IMPORTANT: at this point, our Web Server is NOT allowed to send any kind of reply back to player (yet). Actually, under optimistic locking, Web Server is NOT allowed to send out ANY results of its processing at this point.
    - It is a requirement, because with optimistic locking we cannot be sure that another Web Server doesn’t process another request at the same time. While both Web Servers can prepare their replies, only one of their replies may be valid. For example, if both players are competing for the same resource – only one such request can be satisfied, so only one player should receive a reply “you’ve got it!” – and at this point we do not know yet which of Web Servers will win the race.
- Now, Web Server issues CAS operation with both new-value-of-Game-World-blob, and the same *cas_token* which it has received when it read Game World State

- if *cas_token* is still valid (i.e. nobody has written to the blob since current Web Server has read it), *memcached* writes new value, and returns ok.
    - At this point (due to atomicity which is guaranteed by *memcached*), our Web Server can be 100% sure that no one else has won the race for updating our Game World State.
    - As a result, it is only at this point when our Web Server is allowed to send reply back to the player (or whoever-else-who-requested-the-operation)
- if, however, there was a second Web Server which has managed to write since we've read our blob (i.e. our *cas_token* is “stale”) - *memcached* will return a special error (indicating a “mid-air collision”)
    - in this case, our Web Server MUST discard all the prepared replies
    - in addition, it MAY (and actually SHOULD) read new value of Game World State (with new *cas_token*), and try to re-apply incoming request to this new value of Game World State
        - let’s note that such a treatment is perfectly valid: it is just "as if" incoming request has come a little bit later (which can always happen). Let’s also note that new reply can very well be completely different from the reply which we’ve prepared when we processed the request for the first time (and that’s exactly why we needed to refrain from sending original reply back). For example, if two players were competing for the same resource at about the same time – two Web Servers might have got the same Game World State and prepare “you got it!” replies, but then only one of them will win the race to supply their version of the Game World first – and the second one will be forced to retry the operation again (with the new version of the Game World State, which already says that the resource is not available).

Optimistic locking is simple, is lock-less (which *is* important, see below why), and has only one significant drawback for our purposes. Optimistic locking works fine as long as collision probability (i.e. two Web Servers working on the same Game World at the same time) is low. However, as soon as collision probability grows (beyond, say single-digit percents) - we will start getting a significant performance hit (for processing the same incoming message twice, three times, and so on and so forth). For slow-paced asynchronous games it is quite unlikely to become a problem, and therefore by default I'd recommend optimistic locking as a starting point for web-based asynchronous games, but you still need to understand limitations of the technology before using it.

Pessimistic Locking

Pessimistic locking is conceptually very close to 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. Oh, and Web *mutex* can be implemented on top of *memached* CAS pretty much the same way critical section (or userland *mutex*) is implemented based on CPU CAS operation – and there are libraries out there which do it for you too.

In case of pessimistic locking, Game World State (once again, usually stored as a whole in a blob) is protected by a Web *mutex* (so that two web servers cannot access it concurrently). When using pessimistic locking, Web Server acts as follows:

- obtains lock on *mutex*, associated with our Game World (as it says on the tin, with Pessimistic Locking we're pessimists <sad-face />, 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 <sad-face />.

Let's consider the following scenario. We have dozens of Web Servers (which we usually have), and each is operating along the lines outlined above. Everything works perfectly, unless (more precisely – until) one of our Web Servers (or processes running there) crashes after obtaining *mutex* lock, and before releasing it <sad-face />. In such a scenario, all future interactions with the object which is protected by such a *mutex*, will be blocked *forever-and-ever* <very-sad-face />.[58]

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*-lock comes later to modify the object, 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, our Web Server MUST NOT send any replies back while it holds the lock (it still may prepare them, but they should be delayed until later); instead – it should prepare the replies and hold them for the time being. Only *after* the lock is released (more strictly – after the last write is successfully completed), Web Server is allowed to send out replies. The logic here is pretty much the same as for optimistic locking: until we know for sure that our *mutex* transaction succeeded – we cannot let any information out, as it still MIGHT change.
- third, we need to handle scenario when *mutex*-lock just got broken on timeout – and then that server who previously hold the lock, comes back to life (for example, it didn't crash, but was merely affected by temporary heavy swapping). To deal with it, whenever owner of the broken mutex-lock comes back, our *mutex* need to detect such *mutex*-lock-got-broken cases and report them back to the server-whose-lock-

[58] this, BTW, reminds me of the nasty problems from the early-90ish pre-SQL FoxPro-like file-lock-based databases. To deal with these Really Bad Problems™, the whole concept needed to be changed from file-lock-based stuff into modern DB Servers.

was-broken as an error. On receiving such an error – the Web Server App needs to discard all the prepared replies, and to retry the whole thing (obtaining lock – processing – releasing lock – sending replies).

- fourth, 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 World State are not allowed either, even if they're intended to be replaced "very soon" under the same lock (while this is fine with usual inter-thread *mutexes* where locks are never broken, it is not acceptable under breakable-lock paradigm).

In practice, these issues are certainly solvable, so *memcached can* be used for *mutex*-based pessimistic locking. On the other hand, as for *memcached* we'd need to simulate *mutex* over CAS, I still suggest using optimistic locking (just because it is simpler and causes less *memcached* interactions) – at least as a first implementation until collisions grow too high.

Caches and Transactions

Classical DB transactions are extremely 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 while developing your Game Logic (how to avoid these issues for databases - will be discussed in Vol. VI's chapter on, well, Databases).

However, there is a very different type of transactions out there, it is cache-level transactions (currently the only implementation of such transactions I know about, is the one by *Redis*, but the concept is not restricted to *Redis* for sure).

Redis transactions are completely unlike classical DB transactions and are coming without all the burden described above. In fact, *Redis* transaction is merely a sequence of operations which are executed atomically; compared to *memcached* CAS – it provides a significantly wider spectrum of the potential interactions. 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) – that is, until you realize that you DO need something more complicated than that.

Examples of using *Redis* transactions beyond *memcached*-style CAS, include such things as:

- transactions over multiple objects using optimistic locking and cas-tokens.
- Multiple-*mutex* all-or-nothing locks (which is *different* from locking several mutexes in sequence).
  - One Big Advantage™ of such atomic multiple-*mutex* locks is that they (unlike locking of several *mutexes* in sequence) are inherently immune to deadlocks (that is, IF you always get not more than one lock at a time, whether it is a multiple-*mutex* one or a single-*mutex* one).

Handling Timers

In addition to caches, one thing which is not-that-obvious for Web-Based Architectures, is handling timed events. As HTTP as such is a request-response protocol (and even if we take into account push protocols such as WebSockets, Web Servers App are still operating only when there is a client on the other side) – events-happening-while-there-is-nobody-to-see-them can be a nuisance. In this regard, there are two general approaches:

- an obvious one. Let's have a separate entity which calls timed events. In the simplest case, it can be something like *cron*, but usually it is significantly more elaborated for games (with timed events for different objects written to the database, our own daemon reading the database and invoking those events for relevant objects, and so on).
- Apparently, there is an alternative approach. As the whole problem occurs *only* when there is *nobody* there to see the result – we can ignore all the timed events while there is nobody to see them – and to recalculate (and reapply) all of them as soon as the Game-World-where-the-event-should-have-happened, is accessed (and at this point – we DO have a Web Server App running).
    - If your Game Worlds are deterministic (as discussed in Vol. II's chapter on (Re)Actors) – such re-calculation and re-applying of the events can be done entirely at infrastructure level (i.e. completely transparently for your Web Server Apps), and becomes trivial.
    - One potential caveat on this way is that if your CSRs need to run certain reports – this approach won't show them the latest greatest data; however – from what I've seen, it is more of theoretical concern than a practical one.

Web-Based Deployment Architecture: (Re)Actors

Looking at the title of this section, you may ask: how (Re)Actors (which we've discussed in Vol. II's chapter on (Re)Actors in nauseating detail, and which are usually associated with game loops, simulation loops, and UI-driven apps) can possibly be related to the web-based Server-Side stuff? They seem to be different as night and day, don't they?

Actually, Web-Based Apps and (Re)Actors are not *as* different as they look on the first glance. In particular, if we take a look at both optimistic and pessimistic locking above, we'll see that processing our Game World State under these schemas consists of three steps: (a) taking the whole Game World State, (b) generating new Game World State out of current one based on some input, and (c) storing this new Game World State.

Also, we can observe that out of these 3 steps – (a) and (c) are the same for *all* the games involved, and it is only step (b) which has some substantial app-specific logic.

Now, a flash back to (Re)Actors from Vol. II. Even a very cursory look at it will show that step (b) above *exactly* corresponds to our *(Re)Actor::react()* function; moreover – steps (a) and (c) can be seen as an *implementation detail* of our Infrastructure Code (the one which supports our (Re)Actor and calls *(Re)Actor::react()*). Bingo! If we have a proper (Re)Actor – we *can* easily run it even in such an exotic environment as within a Web-Based Architecture (and without *any* changes); moreover – the choice between Optimistic Locking and Pessimistic one becomes a deployment-time choice (and can be made without any changes to the Game Logic which sits within *(Re)Actor::react()*). This comes *in addition* to an option to use *exactly the same* (Re)Actor for a Classical Deployment Architecture.

In other words, even for a Web-Based Architecture, we can (and IMHO often SHOULD) implement our processing in an event-driven manner, essentially taking current Game World State and incoming events as inputs, and producing resulting Game World State and issuing replies as outputs.

Note that even in this case, our (Re)Actors, in spite of being run from different threads (or even on different computers), will still have a monopoly over the Game World State– and therefore can still be seen "as if" they're running within one single thread; in other words – we still do NOT need *any* inter-thread synchronization within our (Re)Actors.[59]

Overall, (Re)Actors can (and SHOULD) be implemented with absolutely no knowledge of implementation details of the surrounding infrastructure. And

as soon as we have such an infrastructure-agnostic (Re)Actor – we can easily switch not only between pessimistic and optimistic locking, but also between Web-Based Deployment Architecture and a Classical One – all without changing a single line of our (Re)Actor code![60]

Such an ability to re-deploy your game in a different configuration (without rewriting Game Logic) tends to come *very* handy in large-scale projects; while you're developing your game – you never know what kind of issues you will run into when deploying it, and when you're already at the deployment stage – you don't have an option to rewrite your code. So, keeping your options wide open, as a rule of thumb, qualifies as a Really Good Idea™.

Web Deployment Architecture: Scaling and Load Balancing

From the point of view of Scalability - Web Deployment Architecture shown on Fig 9.4, directly corresponds to Stateless-Apps-plus-In-Memory-Cache discussed in Chapter 8. While it is also possible to run a Web Deployment Architecture as Stateless-Apps without In-Memory-Cache – as discussed in Chapter 8, it is rarely a good idea besides scenarios when we *both* need Durability *and* don't care about latencies (which is a very unusual combination for games).

As it was discussed in Chapter 8, such Stateless-Apps-plus-In-Memory-Cache architectures are not too difficult to scale. In particular, as each request is handled on the Web Server

[59] And this, as it was discussed at length in Vol. II, is a *major* advantage for writing Game Logic

[60] BTW, to have this property your (Re)Actor doesn't even need to be non-blocking or deterministic – as a matter of fact, strictly speaking, it doesn't even need to be a (Re)Actor <wink />, though (Re)Actor-style interface tends to simplify such infrastructure-agnostic coding greatly

where it comes (and the requests are pretty much independent) – we already got a scaling which is very close to linear one. Keep in mind though that this near-perfect scalability stands only as long as collisions between different requests (when they're retrieving the Game World State from Back-End Cache) are almost non-existent; if such collisions become frequent – you may need to give up Web Deployment Architecture partially or completely, switching into a "hybrid" Web-Based+Classical architecture (briefly discussed in [[TODO]] section below) or into "Classical" Deployment architecture. Fortunately, if you were using (Re)Actors for your development – such migrations can be done without rewriting the whole thing.

As for scaling database for Web-Based Deployment Architectures – it is also not *that* much of a trouble; the key here is that if we're using that write-back Back-End Cache – we're reducing DB multiple-fold (with numbers such as 10x-100x being pretty normal), so it is usually possible to avoid dealing with scaling for a while (note, however, that if scaling does rear its ugly head – it is MUCH simpler to implement if we have a DB Server App, as discussed in [[TODO]] section below).

When it comes to the Load Balancing – it also tends to be very simple for Web Deployment Architecture. As we do NOT bind Game World state to a single core/thread/server box/… (and instead just pull Game World state wherever the request comes in) – Web Deployment Architecture described above scales pretty much as any other web application (and can be Load Balanced in a pretty much the same manner too). In terms we've defined in Chapter 8 section above – Web Deployment Architecture requires only Clients-to-Servers Load Balancing (and doesn't need Worlds-to-Servers Load Balancing).

On the other hand, for the purposes of Clients-to-Servers Load Balancing in the context of Web Deployment Architecture – we're not really restricted to usual Server-Side Load Balancer Appliances (neither to DNS Round-Robin, phew). Rather – we have an option to use any of the Client Load Balancing methods discussed in Chapter 8; this includes my personal favorite "Client-Side Random Balancing".

On Specific Web Servers

As we're speaking about web servers, we should answer the question of "which web server is better for our purposes?", with a potential to run into a deathly debate of *Apache*-vs-*nginx*-vs-*lighttpd*. However, I don't want to risk entering such a risky waters, and instead will just say that

if you've implemented your game properly[61] – then most likely, choice of specific web server doesn't matter too much

There is, however, one potential exception in this regard. 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 ((Lightstreamer)). I didn't try it myself, so I cannot really vouch for it, but what they're doing with regards to improving

[61] ="as described above" <wink />

interactivity over TCP, is really interesting (we've already discussed some of their tricks in Vol. I's chapter on Communications). As usual, YMMV (and keep in mind that if you're using their specific latency-oriented features, you'll become Locked-In to them), but IMO it is certainly worth considering.

Other than that – to run your game processing code, feel free to choose *Apache*, or *nginx*, or *lighttpd*, and it is rather unlikely that you will see much difference for handling your game traffic (mostly because most of the work will be done in your processing code anyway). However, when serving static data – things are different, and usually *nginx* or *lighttpd* come out comfortably on top for serving static pages/images/etc.[62]

Enter DB Server App

Up to now, we discussed "traditional" Web-Based Deployment model shown on Fig 9.4. As discussed above – it might work (and has been seen working in practice) – but IMNSHO, it has one significant weak point: it is that those request handlers which run on Web Servers, are running SQL directly. This, in turn, means that our Web Server Apps (those handling HTTP requests) become *tightly coupled* to database (and to its structure).

From what I seen, this approach (which corresponds to 20-years-old client-server architecture) very severely limits our options for optimizing/scaling DB side later, when we reach decent amounts of players (as it was mentioned in Chapter 8, very roughly – hundreds of thousands of simultaneous players is a point when it will likely become necessary). Even earlier we're likely to observe that such a tight coupling tends to have *shared* responsibility for our database to performing well – and with "everybody" responsible for DB performance, it almost-universally leads to *nobody* being responsible for it <sad-face />, so your DB is likely to choke (forcing you to undertake severe scalability/optimization efforts) *much* earlier than necessary.

To avoid this kind of problems (and to ensure proper decoupling from the very beginning) – I am advocating for a separate DB Server App, which is responsible for taking incoming requests over DB Server API (which MUST be expressed in terms of game logic, without any SQL), and performing all the manipulations with DB (this includes converting requests into SQL or NoSQL requests, issuing transactions etc. etc.), as shown in Fig 9.5:

[62] BTW, both *nginx* and *lighttpd* use event-driven processing model which is actually the same as (Re)Actors – the model which I'm trying to push ahead throughout this book. Moreover, performance-related advantages of our (Re)Actors over massively multithreaded stuff are of the same nature as the advantage which *nginx*/*lighttpd* holds over *Apache*.

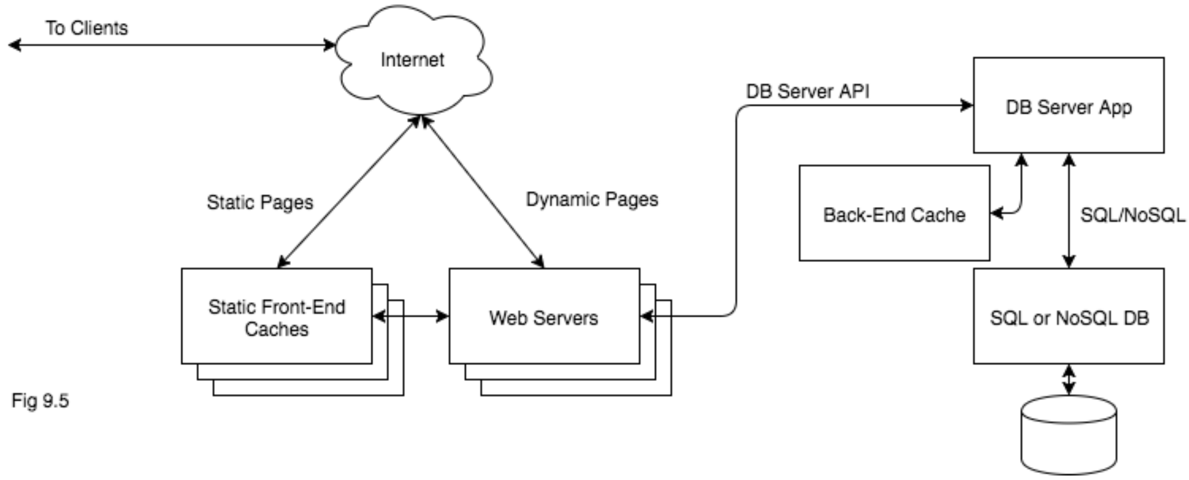


Fig 9.5

As we'll see in [[TODO]] section below, such a separated DB Server App provides many advantages, including, but not limited to such things as ability to change DB structure (use of caches, even switch some parts from SQL to NoSQL and vice versa) completely transparently for Web Servers, better separation of concerns, better specialization of the teams (Web Server App team doesn't need to care about SQL, and DB Team can take full responsibility for the DB), and so on.

If DB Server App ever becomes a bottleneck – it can be changed to behave as a simple "proxy" for incoming requests, which simply forwards requests to "real" DB Server Apps sitting behind it; as such a "proxy" DB Server App can be made to handle at least 100K requests/second[63], overloading it becomes quite difficult to put it mildly.

One further elaboration occurs when we want to simulate In-Memory States for Web-Based Architectures (which is consistent to the Optimistic/Pessimistic Locking schemas discussed above). In many of such cases – storage of the Game State is not really related to our main database, but rather acts as a temporary and separated storage, as shown on Fig 9.6:

[63] taking typical load curves, it roughly corresponds to 1500 billion(!) transactions/year, which is very well beyond anything I ever heard of (except, maybe, for a post-2007 NASDAQ)

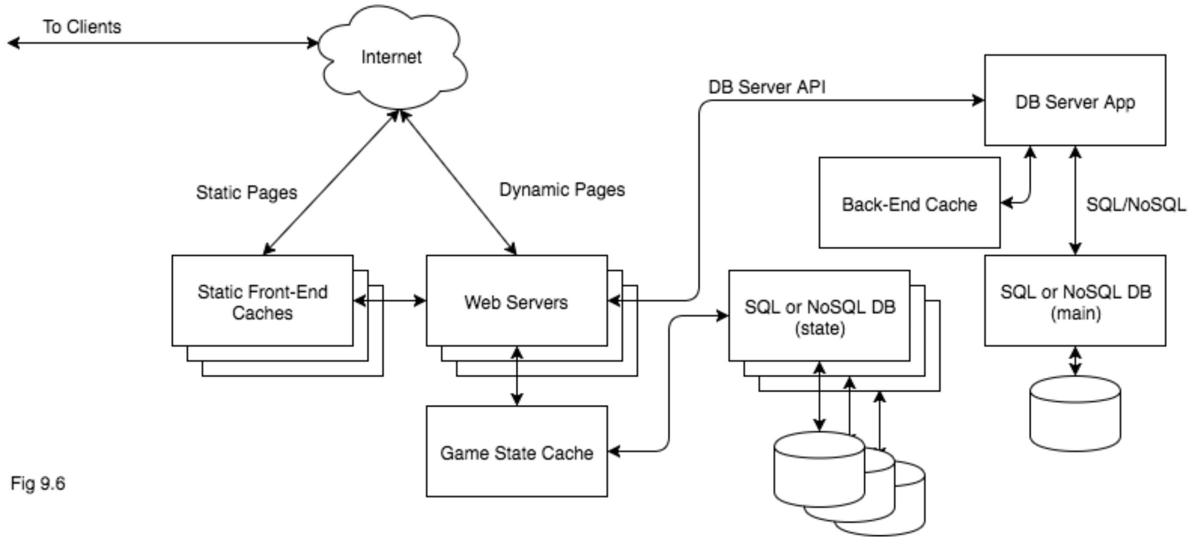


Fig 9.6

As we can see, in this case – we don’t really need to store all the temporary Game World States in our main DB, but instead can delegate this very limited and very-well defined task to a completely separate path, so it doesn’t interfere with main DB (and sharding game states is trivial by design).

Web-Based Deployment Architecture: Merits

Phew, we’ve spent quite a bit of time discussing the Web-Based Deployment Architecture, so now we need a short summary of its pros and cons.

First of all, unlike the naïve approach above, Web-Based systems may work (for relatively slow-paced games, that is). 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 above), 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[64], of course, also has downsides, though they may or may not matter depending on your game (and also note that quite a few of these downsides do not apply if you’re using (Re)Actors as I advocate above):

- Unless you’re doing it (Re)Actor way (see above), 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. *NB: this downside is NOT applicable*

[64] as well as any other one for that matter

if your Web Server App is (Re)Actor-Based (because you can re-use exactly the same (Re)Actors for Classical Deployment Architecture)

- it is pretty much HTTP-only (with an option to use Websockets); migration to plain TCP/UDP is generally rather complicated (that is, unless you're (Re)Actor-Based from the very beginning). *NB: it is NOT that much of a problem if your Web Server App is (Re)Actor-Based for the same reason as above; while migration to a different protocol MAY require changes in marshalling – they're usually not as drastic as "rewriting the whole thing"*
- as the number of interactions between players and Game World grows, Web-Based Architecture becomes less and less efficient (as neither optimistic locking, nor distributed-mutex-locked accesses to retrieve whole Game World State from the back-end cache and write it back as a whole, scales well). Even medium-paced "synchronous" games such as casino multi-players, are usually not good candidates for Web-Based Architecture. *NB: if (Re)Actor-based approach is used, this can be addressed by migrating your game to Classical Deployment Architecture when/if the need arises.*
- you need to remember to keep all the accesses to any shared states synchronized; if you miss one - it will work for a while, but will cause very strange-looking bugs under heavier load. *NB: in (Re)Actor-based approach, it can be implemented within "infrastructure code" and outside your Game Logic, ensuring consistency.*
- 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 locking), and make your own research.

In spite of these issues,

if your game is relatively slow/asynchronous and inter-player interactions are simple and rather infrequent, Web-Based Architecture may be the way to go

This is especially true if you're using (Re)Actors to implement your Web-Based stuff; in particular, using (Re)Actors often allows to postpone making decision on "what we'll use – Web-Based Deployment or Classical one" – until you know your actual requirements much better.

While Classical Architecture described below can also be used for slower-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,

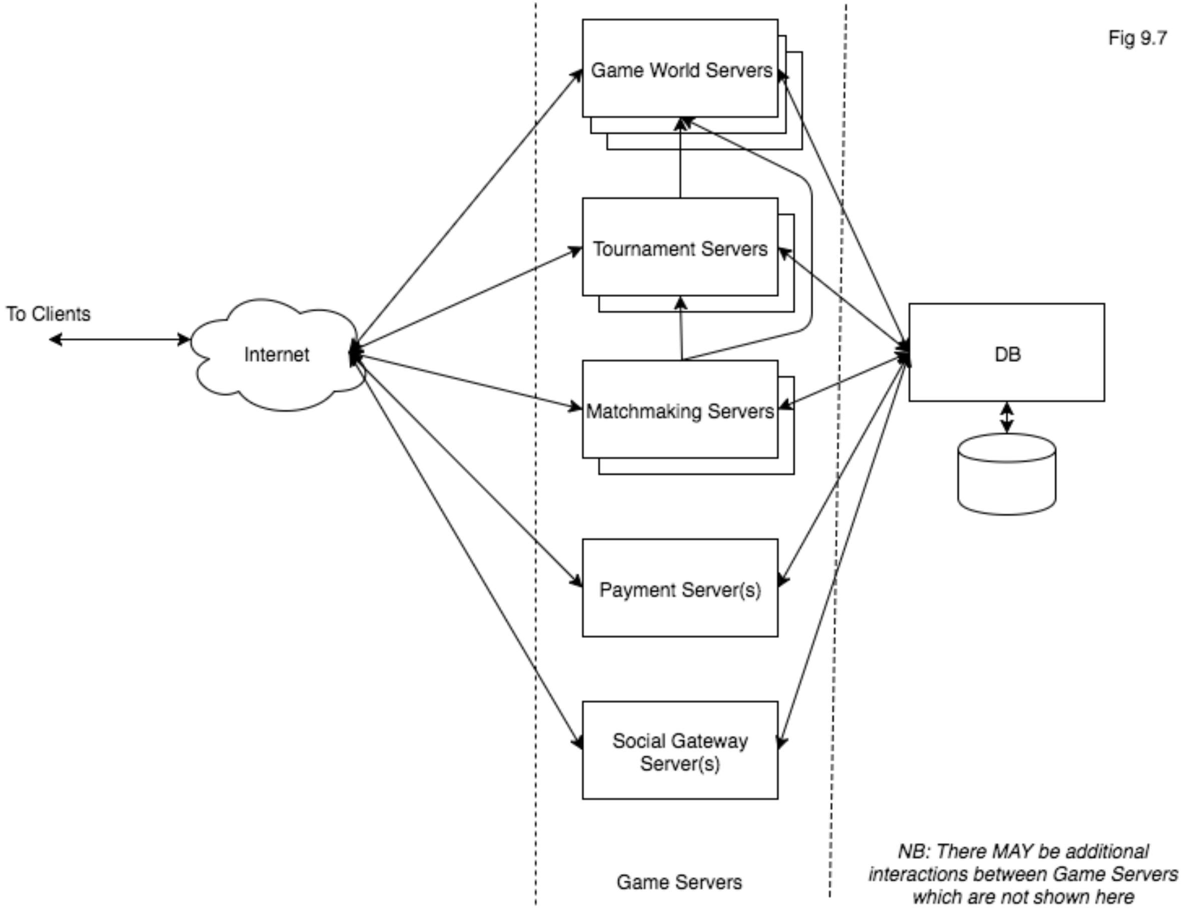
even for medium-paced synchronous multi-player games (such as casino-like multi-player games) Web-Based Architecture is usually not a good candidate

For medium-paced synchronous kind of games, usually the costs of keeping stuff in a single cache and transferring it back and forth between Back-End Cache servers and Web Servers, ends up to be higher than costs of just keeping our state in-memory. And the faster the game – the higher the cost of Web-Based Architecture grows <sad-face />.

Which leads us to what I call "Classical Game Deployment Architecture".

Classical Game Deployment Architecture

Fig 9.7 shows the way how most of the faster-paced games are deployed these days. Let's name it a "Classical Game Deployment" diagram.



In Classical Deployment Architecture, Clients are connected to Game Servers directly, and Game Servers are connected to a single DB, which hosts system-wide persistent state. Each of Game Servers MIGHT (or might not) have its 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 database.

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.

Usually, different types of Game Servers (more strictly – different types of functionality hosted on Game Servers) should correspond to the entities on your Entities&Relationships Diagram described in Vol. I's chapter on GDD. As game entities vary from one game to another one – so do Game Servers. As an example, let's take a look at a few of typical Game Servers which roughly correspond to some of the game entities we've discussed in Vol. I's chapter on GDD (while as always, YMMV, these ones 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 a 50'000-feet architectural point of view, all these seemingly different things are very similar. All of them represent a certain State (we usually name it Game World State) which is affected by player's actions in real time, and changes to this Game World State are shown to all the players.[65]
  - One potential type of Game Servers which is closely related to Game World Servers, is AI Servers. As a rule of thumb (noted in Vol. I's chapter on Communications), if your game has non-trivial AI, it is better to separate AI into separate processes/threads/(Re)Actors; it will allow to keep things clean (and allow to split load to different boxes more easily). In case of AI processes – they can run on the same server boxes as Game World Servers – or on different ones, becoming AI Servers.

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 assigning players to one of your multiple Game Worlds. In practice, Matchmaking Servers can be pretty much anything: from lobbies where players can join teams or select Game Worlds, to completely automated matchmaking (*LoL*-style or something). Usually it is Matchmaking Servers that are responsible for creating new Game Worlds, and placing them on the physical boxes (and sometimes even creating new server instances in cloud environments).

[65] restrictions may apply to which parts of the state are shown to which players. One such example is a Server-Side fog-of-war (and more generally – all kinds of Interest Management), as it was discussed in Vol. I's chapter on Communications

- **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, and so on), 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 servers 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.
    - As noted above, 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 – or they can have their own DB), 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 with their own peculiarities, you can count on it) – is quite 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-manual 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 Vol. VI’s chapter on Payment Processing.
    - 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 an 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 (without any knowledge of *how* the bonus was earned).

Payment Server and Social Gateway Server are necessary to provide interaction of your game with the real world.

We need to note that while in Classical Deployment Architecture, different Game Servers are usually meant as “server boxes” – it is neither an exact requirement, nor a universal rule. I’ve seen very different mix-n-match configurations of different Game Servers distributed over available server boxes; what we should aim for when developing our software – is to allow different deployment configurations without changing code. In particular, we SHOULD NOT write software which assumes that two different Game Servers run on the same box – and neither we should write software which assumes that two different Game Servers run on different boxes. These things should be deployment-time decisions, not development-time ones.

Implementing Game Servers under (Re)Actor-fest architecture

In general, Game Servers can be implemented in whatever way you prefer. In practice, however, I *strongly* suggest to have at least Game Logic implemented under (Re)Actor-fest model described in Vol. II’s chapter on (Re)Actors. Among the other things, (Re)Actors:

- provide very clean separation between different modules
- enable replay-based debug and production post-factum analysis
- allow for different deployment scenarios without changing the (Re)Actor code (this one becomes really important for the Server Side), and
- completely avoids all those pesky inter-thread synchronization problems at logical level
- for further discussion of (Re)Actor benefits – see Vol. II.

Before delving into a more detailed discussion of (Re)Actor-fest architecture, let’s recap some basic points from Vol. II:

- under (Re)Actor-fest paradigm, all the logic (such as Game Logic) resides in (Re)Actors.
    - Each (Re)Actor is just an event-processing machine, with *Reactor::react()* function taking and processing incoming events
    - (Re)Actors do NOT deal with thread sync – and all calls to *Reactor::react()* function are serialized
    - I am arguing for having (logic) (Re)Actors:
        - deterministic (at least exhibiting Same-Executable Determinism as defined in Vol. II)
        - *mostly*-non-blocking (i.e. non-blocking to a reasonable extent, see Vol. II’s chapter on (Re)Actors for further discussion). At least – *all* communications over the Internet MUST be non-blocking. While communications over LAN SHOULD be non-blocking, and disk/DB accesses MAY be kept blocking in certain cases (in particular, when we cannot do anything useful while waiting for reply, anyway).
- Besides (Re)Actors, there is so-called Infrastructure Code. Very simply, it is a thing which feeds incoming events to *Reactor::react()* function of our (Re)Actors – and provides facilities to communicate with the outside world (to send replies and other messages, to write to DB, etc. etc.).

- One of the major benefits of (Re)Actors is that the same (Re)Actor can work with *any* valid infrastructure code; for example – exactly the same Game Logic (Re)Actor can work either under Web Deployment Architecture – and under Classical Deployment Architecture (and without (Re)Actor noticing any difference too).

Minimal (Re)Actor-fest

As noted above, when implementing Game Servers via (Re)Actor-fest architecture, we should at least implement Game Logic (such as Game World Logic) within (Re)Actors:

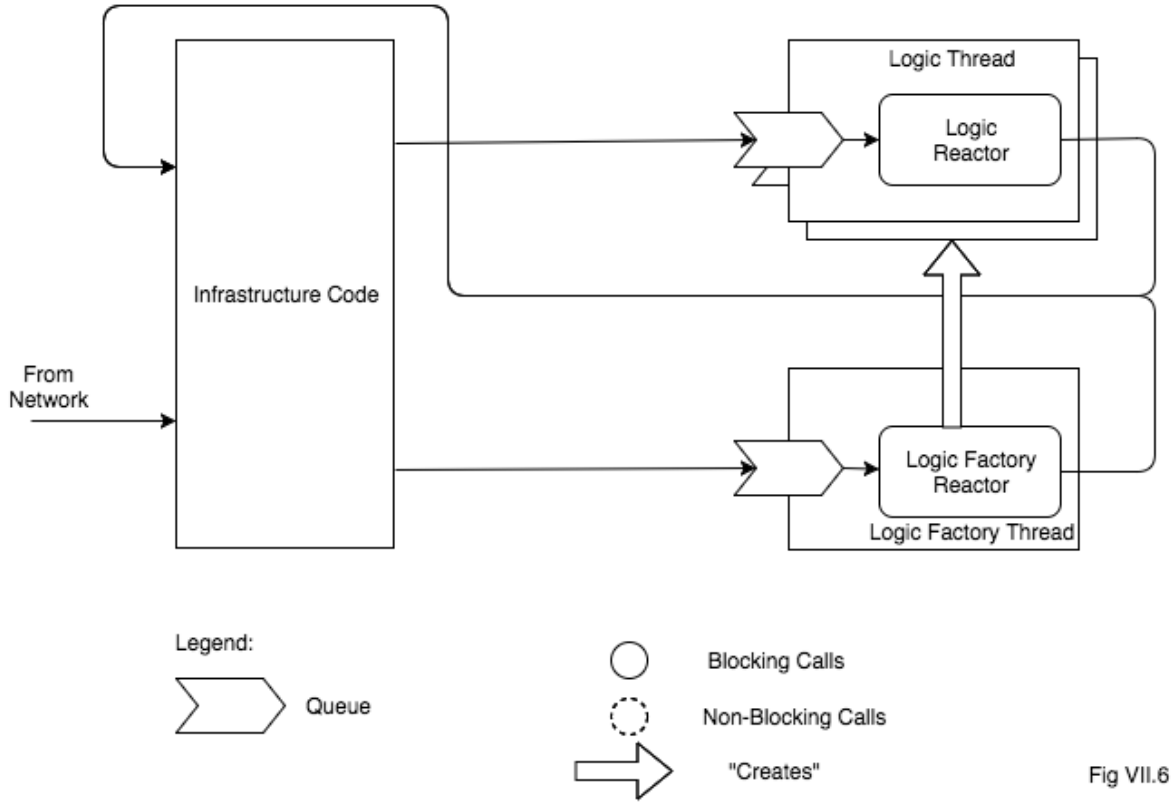


Fig VII.6

[[TODO/fig: VII.6 -> 9.8; rename Reactor->(Re)Actor]]

With a “minimal” (Re)Actor-fest architecture shown on Fig 9.8, we do *not* specify how to implement that Infrastructure Code which feeds events to our Logic (Re)Actors. In this regard, there are many options; in some cases, infrastructure code can be (Re)Actor-based (as discussed in [[TODO]] section below), but in some examples – it can be multi-threaded, using classical mutexes etc. etc. for inter-thread synchronization (that is, as long as mutexes and threads are used for Infrastructure Code only, and we’re NOT allowing to use mutexes and other thread sync mechanisms from within (Re)Actors).

While personally I don’t like multi-threaded code in Infrastructure Code either – still, I have to admit that

Infrastructure Code is very much an implementation detail, and how to implement it - is orders of magnitude less important than having your Game Server Logic implemented as a (Re)Actor.

In other words, as soon as you made your Game Server Logic a (Re)Actor – everything else is not *that* important; after all, you can rewrite all your Infrastructure Code without rewriting the Game Server Logic quite easily.

What is also important, is that even with "Minimal" (Re)Actor-fest architecture, we will be able to get ALL the benefits from our (Re)Actors (that is, with respect to the logic within (Re)Actors, but most of the time it is this logic which changes frequently and requires lots of maintenance); in particular, as long as your Game Logic is a (Re)Actor – you can use production post-factum analysis, and can run replay-based regression tests.

Now let's consider those two all-important (Re)Actors which are present on the diagram on Fig 9.8.

Logic (Re)Actor

The core of the game itself is Logic (Re)Actors. Specifics of those Logic (Re)Actors are different for different Game Servers you have, and can vary from "Game World (Re)Actor" to "Payment Processing (Re)Actor", with anything-else-you-need in between.

It is worth noting that while for most Logic (Re)Actors you won't need any communications with the outside world except for sending/receiving messages (as shown on the diagram), there *may* be gateway-style (Re)Actors (such as Payment (Re)Actor or Social Gateway (Re)Actor) where you will need some kind of external API (most of the time they go over outgoing HTTP, though I've seen quite strange things, all the way up to X.25-over-IP).

(Re)Actors using External Blocking APIs

These external APIs, when present, don't change the nature of those gateway-style (Re)Actors too much, and you will still have all the (Re)Actor goodies (as long as you "wrap" all the calls to that external API, see Vol. II's chapter on (Re)Actors for details). On the other hand, depending on the specific libraries you're using for these external interfaces, you might need to deviate from purely non-blocking (Re)Actors (while you still SHOULD aim for making them entirely deterministic); for gateway-style (Re)Actors blocking calls are usually ok, as long as:

- you have a dedicated thread (either created on-demand, or used from the thread pool) for each instance of your non-blocking (Re)Actor. While you CAN run multiple instances of purely-non-blocking (Re)Actors within the same thread – gateway (Re)Actors are usually blocking, so you don't really have this luxury anymore <sad-face />
- you keep the number of such (Re)Actors which can run simultaneously (and therefore – the number of threads), low. Overall, only a very few systems I've seen, required more than one gateway (Re)Actor per payment provider, and all the systems were fine with a total of a-few-dozens of separate gateway (Re)Actors per social system you're interacting with, but as with anything else, YMMV.
    - If you cannot handle all the requests to a certain gateway within one (Re)Actor – make sure to simply use multiple (Re)Actors for the job (usually –

it will be one "proxy" (Re)Actor dispatching requests to "worker" (Re)Actors, it is indeed *this* simple)

Logic Factory

In addition to obvious Logic (Re)Actors, on Fig 9.8 we can also see a Logic Factory (Re)Actor. It is necessary to create new instances of Logic (Re)Actors (and if necessary, new threads or even processes) when demand for them arises.

For example, when our Matchmaking Server needs to create a new Game World on server X, it sends a request (such as a message or a non-blocking RPC call) to the Logic Factory (Re)Actor which resides on the server X, and this Logic Factory (Re)Actor creates a Game World (Re)Actor with requested parameters.

When our Matchmaking server needs to create a new game world on server X, it sends a request to the Logic Factory (Re)Actor which resides on the server X, and Logic Factory (Re)Actor creates a Game World (Re)Actor with requested parameters.

Deployment-wise, usually there is only one instance of the Game Logic Factory per physical (or virtual) server, but technically there is no such strict requirement.

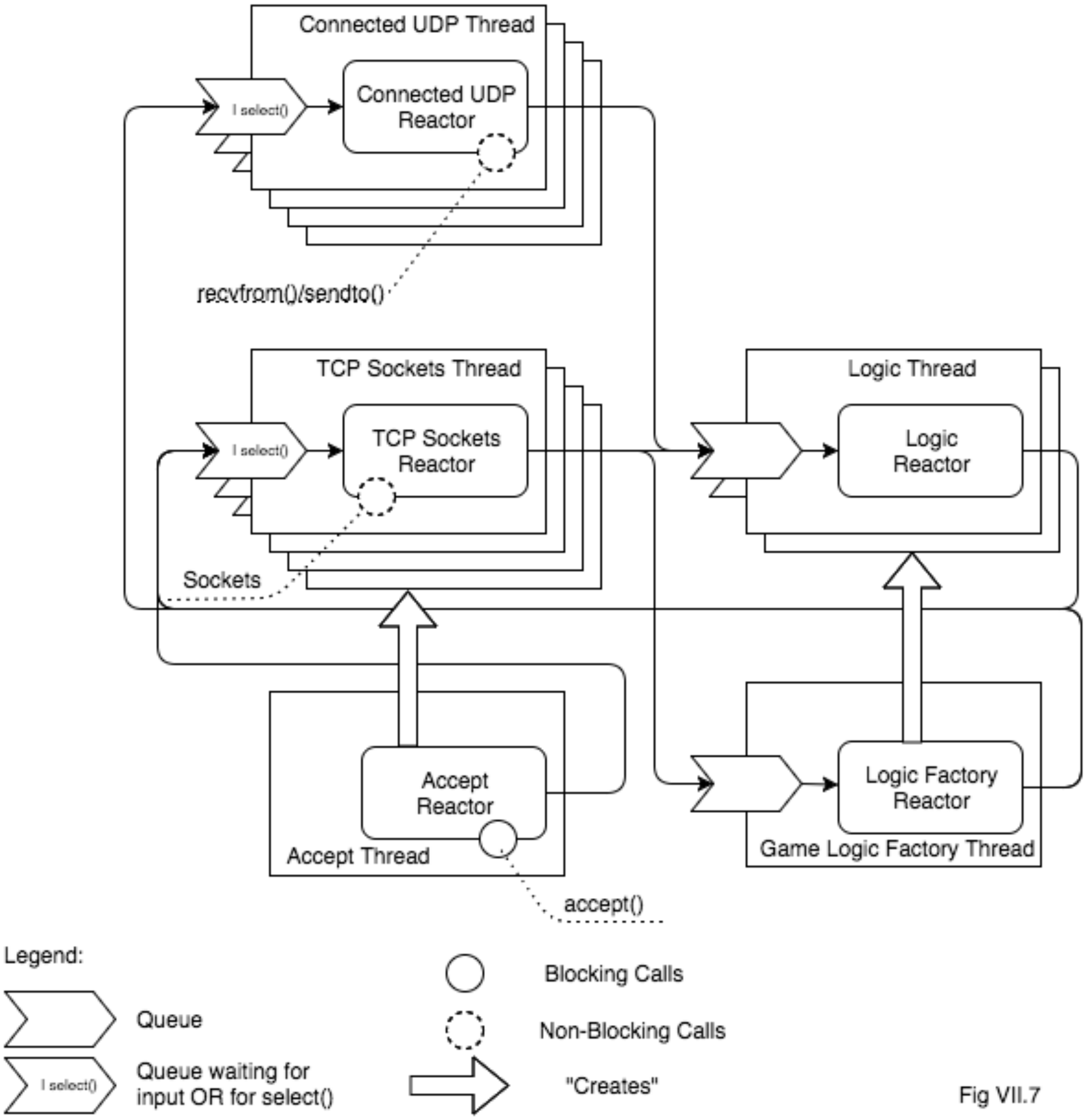
Full-Scale (Re)Actor-fest

As discussed above – the only thing which *really* matters, is to have your Game Logic implemented as (Re)Actors. On the other hand, if you want to be completely faithful to the idea of (Re)Actors[66] – you can implement your Infrastructure Code as (Re)Actors too, as outlined below.

With such a Full-Scale (Re)Actor-fest, you'll be able to get all the (Re)Actor-related goodies (this includes replay testing, production post-mortem etc.) not only for Logic (Re)Actors, but also for all your (Re)Actors which implement infrastructure code. In certain cases, you *might* need some (hopefully minor) deviations from pure (Re)Actors for your infrastructure code – and it is ok, as long as your Game Logic is separated from your infrastructure code by an extremely clean and very straightforward (Re)Actor APIs.

The following Fig 9.9 shows a (much more detailed) diagram with an implementation of the whole generic Game Server under (Re)Actor-fest model (this includes Infrastructure Code implemented as (Re)Actors):

[66] as I am usually want - guilty as charged <wink />



[[TODO/fig: VII.7 -> 9.9, all Reactor->(Re)Actor]]

If it looks complicated at the first glance – well, it should; task of creating scalable servers is never simple. That being said, let's note that the diagram represents quite a generic case, and for your specific game you may not need all of that stuff – at least for first stages of deployment; we'll briefly discuss a way to start development without the need to implement *all* of these (Re)Actors, in the *Starting Small* section below.

Now let's take a closer look at those (Re)Actors which were added to the diagram on Fig 9.9 (compared to Fig. 9.8). As it is shown on Fig 9.9 - it is just TCP-related (Re)Actors and UDP-related ones which were added; however, pretty much any protocol can be handled in a manner similar to one of these two protocols, see, for example, "Websocket-related (Re)Actors and HTTP-related (Re)Actors (not shown)" section below.

TCP Sockets (Re)Actors and TCP Accept (Re)Actor

In Fig. 9.9, TCP-related stuff is represented by TCP Socket (Re)Actors/Threads and Accept (Re)Actor(s)/Thread(s).[67] Here the things are relatively simple: we have classical *accept()* thread, that listens on the socket, accepts incoming sockets, and passes them to TCP Socket Threads (creating TCP Socket Threads as it becomes necessary).

We'll discuss specifics of working with TCP sockets in Vol. IV's chapter on Network Programming; the only really important thing to be mentioned here is that each TCP Socket Thread[68] SHOULD normally handle more than one TCP socket – and to do it, sockets MUST be in non-blocking mode. Usually number of TCP sockets per thread for a typical 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()*,[69] you're likely to hit the wall at around 30 sockets per thread (see also discussion in Vol. IV), 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 thread 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 modern non-overloaded[70] server I wouldn't expect additional latencies and jitter of more than double-digit microseconds, which should be non-observable even for the most fast-paced games.

UDP-related (Re)Actors

From scalability point of view, UDP is quite a weird beast; in many cases, you can use really simple things to get UDP working, but to get to high-performance scalable implementations, you may need to resort to some trickery – or to quite heavy solutions to achieve scalability. The solution on Fig 9.9 is on the simpler side, so you MIGHT need to get into more complicated things to achieve performance/scalability (more on it below).

Let's start explaining UDP-related stuff from Fig 9.9. 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; at the very least, we will need it to know where to send all those updates happening within our Game World State. 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.

[67] of course, if your game is UDP-only, you can skip it, but more often than not, at least some of the Game Servers *do* have some TCP – at least for inter-server communication (as it was noted in Vol. I's chapter on Communications – usually, TCP is preferred for Server-2-Server interactions).

[68] and accordingly, Socket (Re)Actor (unless you're hosting multiple Socket (Re)Actors per Socket Thread, which is also possible)

[69] which IMHO provides the best balance between performance and implementation complexity (that is, if you need to run your servers on Windows)

[70] = "having at least one idle core at all times"

KISS principle

https://en.wikipedia.org/wiki/KISS_principle

KISS is an acronym for 'Keep it simple, stupid' as a design principle noted by the U.S. Navy in 1960

This is where those "Connected UDP (Re)Actors" come in; while they're 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 (Re)Actors and associated threads – as with TCP, 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, of the order of 16-128 UDP connections per thread.

On the other hand, even 128 UDP connections won't be enough for your Server. To deal with it, the simplest (and very popular too) way is to say that:

- your server uses multiple UDP ports
- each of Connected UDP Threads is assigned one single UDP port
    - as a result, it has its own UDP socket, and waits for it
- to balance players to different threads – Matchmaker just gives them different ports on the Server
- Bingo! We have a Share-Nothing architecture, and it is perfectly scalable too.
    - Each of ports/sockets is dedicated per-thread, so there is no contention there
- On the negative side for this approach, I can list only the following:
    - It doesn't allow for easy moving users from one thread to another one. In some (admittedly rather rare) scenarios it can affect how well the balancing can be performed.
    - It exposes implementation details (threads) to the Client. This exposure can potentially have some strange security implications – though I've never heard of any practical schema abusing it.[71]

The downside of this approach is that this UDP Handler Thread can easily become a bottleneck <sad-face />.

IMO, for quite a few games out there, positives of the "port-per-thread" approach above outweigh the negatives. On the other hand, if for whatever reason you don't like it – a few other approaches can be possible:

- An additional UDP Handler Thread reading our single UDP socket – and dispatching the packets to an appropriate Connection UDP Thread. The downside of this approach is that this single thread can easily become a bottleneck <sad-face />.
- Using multiple threads to *recvfrom()/sendto()* from/to the same UDP socket; in this case – scalability is back (and without exposing ports and allowing to move users between the threads too). However, implementation complexity can be substantial. Most importantly, as we still need to access the *state* of our UDP connection – it means that we *either* need to pull this state from some common pool (which has to be a mutex-synchronized container – with lots of contention, ouch!), *or* (in a true (Re)Actor spirit) to pass the incoming UDP packet to

[71] yet to see such an abuse?

the thread which has corresponding state. In this latter case, answering a question “where to store the mapping of incoming-packet-IP/port-pairs to threads”, is not that trivial; shared states (inviting potential contention <sad-face />) are not really desirable, so the message-exchange-based solutions similar to that of Routing&Data Factory (Re)Actor (described in “Routing&Data Factories” section below), might be necessary.

NB: of course, for most/all of these models, platform-specific UDP optimizations can be applied (one popular example being *recvmmsg()*); however, for the time being, we’re not going to discuss socket peculiarities, postponing this discussion until Vol. IV’s chapter on Network Programming.

Websocket-related (Re)Actors and HTTP-related (Re)Actors (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 (Re)Actors as for TCP, but implementing additional header parsing and frame logic within your Websocket (Re)Actors.

Long polling

https://en.wikipedia.org/wiki/Push_technology#Long_polling

With long polling, the client requests information from the server exactly as in normal polling, but with the expectation the server may not respond immediately.

BTW, if you think you need to support pure HTTP (i.e. without Websockets) even for a medium-paced synchronous game such a casino one – think again. While implementing interactive communications over request-response HTTP is possible (in particular, using “long polling”) – it is notoriously difficult to get it work reliably, and tends to cause lots of unnecessary server load. As a result - 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 Vol. IV’s chapter on Network Programming. On the other hand, for asynchronous and/or rrreeeeaaallllyyyy-sssslloooowwww games, real-time *push* is not required, so HTTP (with simple polling) MAY be a reasonable choice.

GPGPU (Re)Actor (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 (Re)Actor (not shown on Fig 9.9) to communicate with your *CUDA*/*OpenCL*/*Phi* GPGPU. A few things to note in this regard:

GPGPU

https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

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

- We won't discuss how to apply GPGPU 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.
- Implementing determinism for GPGPU (Re)Actors, as a rule of thumb, is not trivial, especially if inter-thread interactions are involved (which is usually the case for GPGPUs). For example, a simple change in the order of floating-point additions may easily lead to rounding-related differences in the last digit (with both results being practically the same, but technically different). However, as it was discussed in Vol. II's chapter on (Re)Actors (and in "DIY Fault Tolerance in case of Almost-Determinism" section below), even such *almost-determinism* has some practical uses (albeit much less of them than real determinism); one of such uses is discussed in the "DIY Fault Tolerance in case of Almost-Determinism" section below.
- 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 Vol. VII's chapter on Preparing to Launch, for further discussion)
    - As soon as you agree that it is not your servers, but leased ones or cloud ones (see also Vol. VII), it means that you're completely dependent on your server ISP/CSP on supporting whatever you need.
    - Most likely, with 3rd-party ISP/CSP it will be Tesla or GRID GPU (both by NVidia), so as a rule of thumb, you should be ok with CUDA rather than OpenCL.

CSP

https://www.sdxcentral.com/cloud/definitions/what-are-cloud-service-providers/

Cloud service providers (CSP) are companies that offers network services, infrastructure, or business applications in the cloud.

    - The choice of such ISPs which can lease you GPUs, is limited, and they tend to be on an expensive side <sad-face />. As of the mid-2017, the best I was able to find was a server with 2x Tesla M60 GPUs (i.e. 2x 4096 CUDA cores) rented at ~$1500/month (that's for 17TFLOPs or so). With cloud-based GPUs, things weren't any better, and started from around $500/month per Tesla K80 (the one with 2496 total cores). <ouch! />
        - One interesting exception at that time was an offering from ovh.co.uk; they seemed to offer a server including 4x GTX1070 (4x 2048 cores, around 23TFLOPs) for mere GBP565/month (+VAT). While not directly comparable to Teslas (in particular, double-precision is traditionally MUCH slower on GTX line, but chances are that you

won't need double precision for games anyway) – it looks very interesting, as it is about 7x more price-efficient[72] than the M60-based ones mentioned above (YMMV, batteries not included – and make sure to ask ovh folks how many of such servers they can provide).

- If you are going to co-locate your servers instead of leasing them from ISP,[73] 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 mid-2017, Tesla M60 costs around $3500(!); at this price, you get 2xGM204 cores, 16GB RAM@5GHz, clock of 930/1180MHz, and 2x2048 CUDA cores – resulting in overall single-precision performance of 9.6TFLOPS. At the same time, desktop-class GeForce Titan X Pascal is available for about $1200, has a newer GP102 core, 12GB RAM, clock of 1417/1531MHz, and 3584 CUDA cores – resulting in single-precision 11TFLOPS. In short – for single-precision calculations Titan X Pascal gets you even more power than Teslas M60/P100 at less than 30% of the price. It might look as a no-brainer to use desktop-class GPUs, but there are several significant things to keep in mind:

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.

    - 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 and double-precision math can be very nasty depending on specifics of your code
    - 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 almost-never a viable option (see Vol. VII's chapter on Preparing to Launch)
    - 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 (especially those 1U ones) are extremely tight, and air flows are traditionally very different from the desktops, so throwing in additional 250W or so with a desktop-oriented air flow to a non-GPU-optimized server isn't likely to work without throttling for more than a few minutes.

[72] For single precision, i.e. for 4-byte *floats*

[73] this potentially includes even assembling them yourself, but I generally don't recommend it

- IMVHO, 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. Unfortunately, I don’t know of any major vendor which is officially supporting desktop-class-GPGPUs-in-servers (and those few medium-class vendors which did support desktop-class GPGPUs – are now Tesla- and GRID-only, and priced accordingly <sad-face />). Feel free to try whichever-vendor-you-can-find, but don’t hold your breath over it. At the very least, make sure to rent/buy one such a box ASAP and run it for a month 24x7 under top load to see whether it copes well with load and cooling (and ideally – you should run it in the datacenter where cooling conditions can differ from those in your office); also make sure to double-check if your colocation provider is ready to host these not-so-mainstream boxes (some of them are rather non-standard, including non-standard height[74]).
- Alternatively – you can keep trying combinations of standard rack servers with desktop cards, hoping to find a combination which works for you. On this way, I’d rather try larger rack boxes (at least 2U ones, better 4U ones) – but my wild guess is that you’re going to spend LOTS of time until you find some server+desktop-GPU combination which can work for many days without overheating.

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

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 GPGPU on servers will take much more effort than simply making your software run on your local Titan X or GTX1080 <sad-face />. 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.

Starting Small

Taking another look at the diagram on Fig 9.9, it becomes obvious that if your server doesn’t need to support UDP or TCP – of course, you won’t need corresponding threads and (Re)Actors. However, keep in mind that usually your connection to other servers (such as DB Server App) 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 (one exception is if you’re using an already-working 3rd-party “reliable UDP” library; in this case, you can have the whole thing UDP-only without spending lots of time on figuring out UDP-related bugs over Server-2-Server connections).

[74] I’ve seen 4.75U box (with 8 GPGPUs), it is indeed very strange – though it should fit into the standard rack, so your colo ISP *might* allow it.

On the other hand, our (Re)Actor-fest architecture provides a very good separation between protocols and logic, so usually you can safely start[75] with a TCP-only server even for a really fast-paced game. This will almost-certainly be enough to test your game intra-LAN (where packet losses and latencies are negligible), and implement UDP support later (and without the need to change your Game Logic (Re)Actors). Still, make sure that you implement UDP layer ASAP – there *will be* quite a few of nasty caveats for sure.

Strictly speaking, it is also possible to start with a UDP-only server to add TCP later – but as implementing reliable inter-server connections is a headache for UDP, this is rarely the best option.[76]

(Re)Actor-Fest Architecture on the Server Side: Flexibility and Deployment-Time/Run-Time Options

When it comes to the available deployment options, (Re)Actor-fest is an extremely flexible architecture. We already briefly mentioned this flexibility in Vol. II's chapter on (Re)Actors; however, in Vol. II our discussion was more about (Re)Actors in general. Now, we're going to discuss your deployment and run-time options provided by (Re)Actor-fest on the Server-Side (note that these goodies come *in addition to* generic (Re)Actor benefits discussed in Vol. II, including such beauties as production post-mortem and replay regression testing using real-world data).

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. In just one example, I've seen people almost-running into the wall trying to identify which of the (Re)Actors causes memory corruption – and stopping short just inches before the wall, due to ability to run (Re)Actor in a separate process (identifying the problem very nicely).

In general, Deterministic (Re)Actors 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.

Philosophical ranting aside – let's take a more practical look at the very practical benefits of the (Re)Actors on the Server-Side.

[75] As a Really Big Fat Rule of Thumb™, we SHOULD NOT intend to use TCP in production for fast-paced games such as FPS or MOBAs – it will turn out to be a suicide much more often than not. However, for a very preliminary over-the-LAN testing it will work.
[76] once again – unless you're using a RUDP library which already works.

Threads and Processes

First of all, you can have your (Re)Actors deployed in different configurations depending on your needs. In particular, (Re)Actors can be deployed as multiple-(Re)Actors-per-thread, one-(Re)Actor-per-thread-multiple-threads-per-process, or one-(Re)Actor-per-process configurations (all this without changing your (Re)Actor code at all).[77]

(Re)Actors can be deployed as multiple-(Re)Actors-per-thread, one-(Re)Actor-per-thread-multiple-threads-per-process, or one-(Re)Actor-per-process configurations (all this without changing your (Re)Actor code at all)

One real-world example. 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-(Re)Actors-per-thread, another bunch of Game Worlds (intended for mature players) – deployed as a single-(Re)Actor-per-thread (improving latencies a bit, and providing an option to raise thread priority for these (Re)Actors), and those Game Worlds for pro players – as a single-(Re)Actor-per-process (additionally improving memory isolation in case of problems, and improving memory locality and therefore performance a further tiny bit); all these (Re)Actors were using absolutely very same (Re)Actor code, but it was compiled into different executables (which were using different Infrastructure Code) 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 have this (Re)Actor to a dedicated thread, and then pin this thread to a single core (preferably the same where interrupts from you NIC come on this specific server box) and pin other processes to other cores, keeping your latencies to the absolute minimum.[78]

CPU pinning

https://en.wikipedia.org/wiki/Processor_affinity

Processor affinity, or CPU pinning, enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU.

Underlying Communication Protocol as an Implementation Detail

With (Re)Actor-fest architecture, exact-communication-protocols-you're-using become an implementation detail (which can be delegated to your Network Team). For example, you can have the same Game Logic (Re)Actor to serve both TCP and UDP connections. Not only it comes handy for testing

[77] Restrictions apply, batteries not included. If you have blocking calls from within your (Re)Actor, which is common for DB-style (Re)Actors and some of gateway-style (Re)Actors, you shouldn't deploy multiple-(Re)Actors-per-thread

[78] this will further reduce latencies in addition to any benefits obtained by simple increase of thread priority, because of per-core caches not being evicted by threads accidentally running on the same core

purposes (as mentioned in the *Starting Small* section above, but it also may enable some of your players (those who cannot access your servers via UDP due to firewalls/weird routers/using-web-client etc.) to play over TCP[79], 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 (Re)Actor-fest architecture 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 is specific to Server-Side (Re)Actor-fest architecture, is to allow moving your Game Worlds (or more generally – (Re)Actors) from one server to another one (though with some additional headache, and a bit of additional latencies).

One simple way to do it, is to:

- We have our (Re)Actor with serialization
- Whenever need to migrate (Re)Actor arises – we:
    - Stop processing messages in the (Re)Actor
    - serialize your (Re)Actor on source server (see Vol. II's chapter on (Re)Actors for a brief discussion on serializing (Re)Actors)
    - transfer serialized state to a target server's Game Logic Factory, and to
    - deserialize our (Re)Actor there.
    - Resume processing messages in the (Re)Actor

Bingo! Your (Re)Actor now runs on the target server right from the same logical moment where it stopped running on the source server. In practice, however, moving (Re)Actors around is not that easy, as you'll also need to notify your clients about changed address where this moved (Re)Actor can be reached, but despite being an additional chunk of work, this is also perfectly doable if you really want it.

In this regard, the most obvious way is to notify your Clients about IP/port change of your (Re)Actor (so that Clients can reconnect to this new IP/port). On the other hand, IF you're using Front-End Servers (as discussed in [[TODO]] section below) – it is perfectly possible to handle this movement-of-(Re)Actor without Clients realizing it, so all the changes are limited to communications between Front-End Servers and Game Servers. For more details – see also discussion on "re-connecting at source" in Chapter 10.

Low-latency (Re)Actor Migration

While simplistic serialize-then-deserialize approach discussed above, does work – it has a potentially significant drawback: latency. In the model above, while we're serializing-transferring-deserializing, there is no chance to process any messages (plus after deserialization, we'll have that cost of "re-connecting at source" or equivalent). In general, it *is* possible to improve latencies for (Re)Actor migrations, making such migration *very* fast

[79] BTW, in Vol. IV's chapter on Network Programming we'll discuss how to make TCP almost-as-responsive as UDP – while it is not a picnic, and causes quite a bit of server load, it *seems* to be doable.

(bringing observable migration times below 1ms); unfortunately, it comes at a cost of significant complication of the migration process:

- We have our (Re)Actor with serialization
    - In addition, our (Re)Actor is *deterministic*
- Whenever need to migrate (Re)Actor arises – we:
    - Do NOT stop accepting and processing incoming messages (yet)
    - Serialize the state of our (Re)Actor (if necessary – it can be even done in an incremental manner)
    - Transfer serialized state to the target Server Box – *and* start duplicating to the same Server Box all the input messages coming to the source (Re)Actor, to the target (Re)Actor[80]
    - On the target Server Box – deserialize received state, creating a target (Re)Actor, and start feeding it all the duplicated input messages. At this point, target (Re)Actor works in a "rollforward" mode, ignoring all the direct inputs from the Clients, and processing *only* duplicated input messages coming from the source (Re)Actor.
    - Instruct all the Clients (which are connected to the source (Re)Actor) to create another connection to the target (Re)Actor; in this mode – Clients are sending all their data to *both* source and target (Re)Actors, and processing replies from *any* of them
    - After all (or *almost*-all) the Clients have both connections – source (Re)Actor is deactivated, and then the target (Re)Actor is switched from "slave" mode to normal mode of operation. This delay – from disabling source (Re)Actor to enabling target (Re)Actor – is the *only* delay which is visible to the Clients in this process;[81] as such – it can be easily brought to below-1ms range.
    - Now, we can say Clients to switch back to a single-connection mode (using only target (Re)Actor now).

As we can see – this process, while being significantly more involved, *does* allow to reduce latencies; whether its benefits are worth it for you – depends on your specifics.

Online Upgrades with (Almost-)Zero Downtime

Yet another two options provided by (Re)Actor-fest architecture, 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 (Re)Actors (while existing (Re)Actors are still running with the old code). This works as long as changes within (Re)Actors are minor enough so that all external inter-(Re)Actor interfaces are 100% backward compatible, and the life time of each (Re)Actor is naturally limited (so that at some point you're able to say that migration from the old code is complete). This technique is reminiscent of good ol' Blue-Green deployments, though (unlike traditional

[80] in case of incremental serialization – input messages usually have to be duplicated after *the very first portion* of the serialized state is sent.

[81] Actually, as we run target (Re)Actor for a while before switching to it – even cache population latencies don't apply in this model.

Blue-Green techniques) it works on a per-(Re)Actor basis which in turn enables gradual migration (as opposed than all-or-nothing migration typical for traditional Blue-Green environments); in other words – during migration, while some of our already-running Game Worlds are still “Blue”, newly creates ones are already “Green”.

When implementing this Blue-Green-like gradual upgrades, we DO need to ensure compatibility of our Game Worlds with all the dependencies; usually, the most critical dependency is our DB Server App. On the other hand, most of the time, functionality of DB Server App is not *changed*, but rather it is *extended*; in all such cases, it becomes possible to upgrade DB Server App first – and to upgrade Game World Apps in Blue-Green-like manner later; as new DB Server App (the one with extended functionality) should be compatible with *both* “old” and “new” Game World Apps – this migration path will work quite smoothly.

Blue/green deployment

http://searchitoperations.techtarget.com/definition/blue-green-deployment

A blue/green deployment is a software deployment strategy that relies on two identical production configurations that alternate between active and inactive. One environment is referred to as blue, and the duplicate environment is dubbed green.

-- TechTarget

The second of the online-upgrade options allows to upgrade (Re)Actors while the Game World is still running. The idea behind it goes the route of serialization (Re)Actor’s state – replacing the code – deserialization of (Re)Actor’s state. This serialize-replace-deserialize option, however, tends to be much more risky than the first one, and potential migration problems may be difficult to identify and/or to test. Formally – for the second option to work, we need to *guarantee* (a) that serialization and deserialization (which in this case are performed between *inherently different* executables), are perfectly compatible, and (b) that regardless of the point where we serialize-replace-deserialize – the behavior will be sane.

In practice – unless we’re using IDL-with-backward-compatibility-support (which we should, but rarely do; more on it in Vol. I’s chapter on Communications) - even (a) often becomes an insurmountable problem. Worse than that, testing for *all the potential switch points* in (b) is rarely feasible (and predicting which of the points are of potential interest, is very difficult for a generic upgrade). Therefore, if you cannot rely on Blue-Green-like deployments and need to go serialize-replace-deserialize route – the following precautions are advisable:

- Make sure to use IDL with an explicit support for backward compatibility. As discussed in Vol. I – there are many reasons to do it anyway, but if you want to migrate via serialization – it instantly becomes a must-have.
    - Make sure to enforce backward compatibility for serialization of your (Re)Actor’s state
- Try to limit the number of potential states where migration can occur. Quite often, it is better to wait with migration for a few minutes (until (Re)Actor reaches a “stable” state) than try to test all the potential migration points
- Make sure to use severe automated testing specifically for the migration. In particular – make sure to use “replay” technique (also provided by (Re)Actor, see Vol. II’s chapter on (Re)Actors for details) to test randomly-happening migration. Such testing should use big chunks of the real-world data, and should simulate online upgrades at the random moments of the replay.

Implementing Infrastructure Code for Server-Side (Re)Actor-fest

Coming back to our diagrams on Fig 9.8 and Fig 9.9, we can see that from implementation point of view – *all* our (Re)Actor threads are always the same:

- Each thread[82] have *exactly one* queue to wait on
- They wait on this queue, then whenever *something* comes into the queue – they process this *something* – and go back to waiting on the queue.

As a result, the main question for implementing Infrastructure Code *at least* on the Server-Side – becomes “how to implement this queue”.

*nix: sockets and pipes, that’s it

Fortunately enough, on *nix even the most straightforward implementation tends to work very well in practice. The very basic idea goes along the following lines:

- For TCP/UDP communications – use good ol’ Berkeley sockets
- For inter-(Re)Actor communications – use anonymous pipes
- Implement each of our queues as a set of sockets/pipe handles
- Implement waiting on a queue as waiting for something to happen on these sockets/pipe handles (using *select(), poll(), epoll(),* or *kqueue()*[83]) – then we get *both* “just queue” and “queue which can also wait for socket” very easily.
    - As we can see from Fig 9.9, even for a full-scale (Re)Actor-fest, these two queues are *all* we will probably need. While strictly speaking, for 100% non-blocking processing, there can also be data coming from the disk or from DB – but in practice, under our *mostly*-non-blocking paradigm, disk and/or DB operations rarely need to be handled in a non-blocking manner[84], so waiting for sockets and pipes happens to be sufficient.

That’s it, we’re quite unlikely to need anything else under *nix.

Windows: *WaitForMultipleObjects()* vs *IOCP*

> *"Ave, Imperator, morituri te salutant"*
> *(“Hail, Emperor, those who are about to die salute you”)*
> *-- quoted in Suetonius, “The Twelve Caesars”, 121 AD*

[82] if there are multiple (Re)Actors per threads, it is still exactly one queue *per thread*

[83] discussion on “which of these functions is better”, is beyond this Chapter, and belongs to Vol. IV’s chapter on Network Programming

[84] moreover, often it doesn’t make any sense to handle disk or DB operations as non-blocking; this routinely happens whenever we *cannot* handle any new operations while we’re waiting for the result of the current one.

I know for sure that I will be beaten Really Hard™ by hardcore zealots of IOCP (="Input/Output Completion Ports") on MS Windows, but being honest with myself and you, I have to say that

At least for (Re)Actor-like load[85], "traditional" uses of IOCP tend to behave *worse* than WaitForMultipleObjects()

Yes, I am arguing to build Infrastructure Code not along the lines of ubiquitous-on-Windows IOCP (traditionally used with multiple reading threads per queue), but along exactly the same lines as on *nix. Sure, implementation details are going to be very different (on Windows, we'll need to wait using *WaitForMultipleObjects()* on *OVERLAPPED* events instead of waiting for file handles using *epoll()*/etc. under *nix), but overall idea still remains the same: to specify set of events we're waiting for[86] – and to wait for them, it is this simple.

Moreover, I am not just *arguing* to do it – I also done it myself, and this architecture-I-wrote-over-15-years-ago, is still in use, and is still working like a charm for a billion-dollar-company (beating any major competitor at least 10-fold on players/server metric). To make it even more compelling – I can say that at some point, they tried to rewrite this simplistic architecture into multiple-threaded IOCP; and guess what? – IOCP-based architecture performed *worse* than original one[87].

The reason for it is rather simple. For (Re)Actors, the processing is inherently *stateful* – so we DO need to access the state of our (Re)Actor. On the other hand, with traditional uses of IOCP – we have multiple threads waiting on the same IOCP queue (i.e. we're using IOCP queue as a Multiple-Writers-Multiple-Readers queue a.k.a. MWMR queue) – and essentially let IOCP decide which thread will execute after the operation is completed. It means that to process the request while accessing the state of our (Re)Actor, we'll still need to synchronize on the state of (Re)Actor (for example, using mutex), and then to call *Reactor::react()* while the state is protected by our lock on mutex.

Apparently, this approach is severely sub-optimal. First, synchronizing on mutex creates a potential for contention, which in turn will cause extra thread context switches (and as context switch can take up to 1M CPU cycles [TODO: ref to the article] – we don't really want to have any unnecessary ones); moreover, this contention tends to behave *worse* as the load increases (causing pretty bad non-linear behavior exactly when it matters most). Second – even in the case of no-contention, CPU-randomly-selected-by-IOCP-to-run-the-request will still need to "pull" the state of our (Re)Actor into caches of the current CPU core; this, in turn, more often than not, *will* cause an observable performance hit (after all, with modern CPUs, each memory read from main RAM can cost up to several hundred CPU cycles; even reads from shared L3 cache are around 40 cycles). To make things even worse – we need to keep in mind that at least for last 20 years, vast majority of the production-level server boxes are in fact multi-socket NUMA boxes, so randomized access patterns typical for

[85] Actually, it is pretty much *any* stateful processing

[86] for *all* the (Re)Actors residing within the thread

[87] The difference was admittedly marginal, but it was perfectly clear that IOCP is *at least not performing better* (and is probably performing worse) than original implementation based on *WaitForMultipleObjects()*.

"traditional" MWMR-style IOCP, will routinely cause one CPU socket to access the memory of another CPU socket via HyperTransport/QPI, raising the cost of "pulling" the state into caches on current CPU core even further.[88]

For stateful (Re)Actor-like processing, the only tangible disadvantage of *WaitForMultipleObjects()* compared to IOCP, is a limit on the number of the sockets you can wait for in one call to *WaitForMultipleObjects()*; this, in turn, limits the number of the sockets per thread to about MAXIMUM_WAIT_OBJECTS/2 (last time I checked, MAXIMUM_WAIT_OBJECTS was still 64, so you won't be able to handle more than 32 sockets per thread; however – as mentioned above, 32 sockets/thread is very close to what-we-usually-want anyway).

Last but certainly not least:

Criticism of IOCP above applies ONLY if we're using IOCP in a "usual" way – i.e. handling incoming requests in MULTIPLE threads (i.e. using IOCP as an MWMR queue).

On the other hand, if we're using IOCP is a manner which is ideologically similar to MWSR[89]-like *WaitForMultipleObjects()* (i.e. with only ONE thread which can handle the request – and it is exactly that thread where our (Re)Actor runs, so once again we don't need any thread sync) – the whole thing will become much more similar to MWSR-like *WaitForMultipleObjects()* than to "usual" MWMR-like IOCP. TBH, I didn't try this IOCP-as-MWSR approach myself – but my educated guess is that it *will* probably work pretty well (in particular, it MAY allow to lift that 32-sockets-per-thread limitation mentioned above).

Bottom line on IOCP:

- Feel free to try it
- If using IOCP in traditional MWMR-like mode - do NOT expect miracles; most of the time, for stateful loads, multiple-threaded IOCP will *lose* to *WaitForMultipleObjects()* (that is, assuming that implementation based on *WaitForMultipleObjects()* is a reasonably good one)
    - One potential exception may occur if you have LOTS of (Re)Actors with LOTS of blocking operations. However – for Game World (Re)Actors there shouldn't be anything blocking at all, and for those-(Re)Actors-which-do-have-blocking (such as DB (Re)Actor) – you won't have too many of them anyway.
- On the other hand, MWSR-like uses of IOCP (i.e. with a hard limitation of "only one specific thread – the one containing the target (Re)Actor - is allowed to processing requests in each of IOCP queues") MIGHT be perfectly viable.
    - Note that in this case, the whole architecture will become significantly more involved (and *much* more similar to *WaitForMultipleObjects()*-based one) than usual MWMR-based IOCP systems. In particular – we'll need to maintain multiple IOCP queues (exactly as with *WaitForMultipleObjects()*, there will be

[88] In contrast, for a system based on *WaitForMultipleObjects()* – we'll have extremely good locality (both temporal and spatial) coming without any additional efforts from our side.

[89] ="Multiple Writers Single Reader"

exactly one queue per thread), and we'll need to know in advance where exactly any of the requests should go.

To further summarize it in one phrase –

For (Re)Actor-like stateful processing, you SHOULD concentrate on MWSR queues, as they tend to beat MWMR ones performance-wise; which system API to use to implement an MWSR queue – you'll need to find yourself[90].

This is consistent with my inherent dislike to use any of the MWMR-like techniques (including, but not limited to, all the techniques using thread pooling, Smartfox server, and DarkRift) for stateful processing. I have to admit that most of the time, this performance difference is not fatal (and, depending on your game, can be negligible); still, whenever I have a choice – I certainly prefer to avoid unnecessary mutexes and unnecessary context switches (which, BTW, is exactly why *nginx* outperforms *Apache* - not fatally, but sustantially).

On libevent/libev/libuv on the Server-Side

In addition to good plain old sockets, there are several event-driven libraries out there, such as *libevent*/*libev*/*libuv*. One thing I want to note is that while on the Client-Side (that is, unless you're using big 3rd-party engines) these libraries tend to be very useful, on the Server-Side I am not sure whether they are necessary. Let me explain:

- The main strength of these libs is to bring all the events of interest in the system together, and to wait for any of them happening under one single call. This is exactly what we needed (and achieved) above from socket+pipe queues (and their reasonable facsimile on Windows).
    - In general, events of interest include such things as arriving network packets, user input, app-level events, and so on.
- On the other hand, on the Server-Side, we don't have the variety of events that are present on the Client. In fact, pretty much everything we'll ever be speaking about on the Server-Side is (a) network packets, (b) timer events, and (c) app-level inter-(Re)Actor messages. And as we've seen above - all these three things can easily be handled without any intermediary libraries, both under *nix and under Windows.
- As for disk I/O (which doesn't fit into *poll()*/*epoll()*[91]) – it is fairly rare to have direct disk I/O on the Server-Side (most of the storage goes to DB anyway), so that it is rarely an issue. Moreover, even when disk access *is* necessary (for example, for local logging), for local disks – it usually can be done in a blocking manner (see discussion on mostly-non-blocking processing in Chapter 5).

As long as these observations stand (and I didn't see any Server-Side system where they didn't), I don't really feel that intermediary event libs are necessary or even convenient on the Server-Side; after all, *poll()*/*epoll()* and *WaitForMultipleObjects()* are (a) better documented, and (b) will perform at least a bit better (and if by any chance *lib** chooses to

[90] on the other hand – if doing things correctly, you cannot go too wrong with pretty much any system-level MWSR.

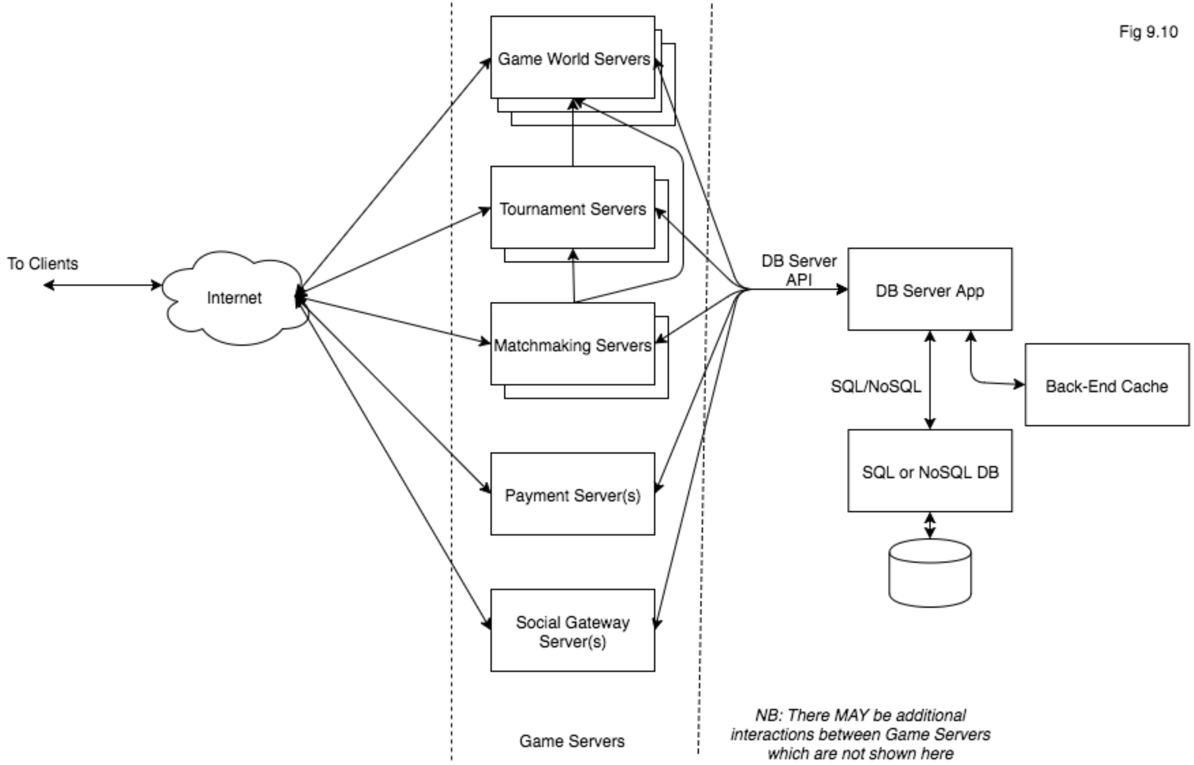
[91] BTW, *BSD's *kqueue()* is better in this regard

simulate a non-blocking call via blocking one – really non-blocking implementation will perform MUCH better).

In other words: I admire the work done by authors of these libs (especially *libuv*), but I don't feel that they will provide substantial simplification when used on the Server-Side; after all – if we're using (Re)Actors, all this code belongs to written-once Infrastructure Code (so we'll be isolated from the platform specifics anyway).

DB Server App for Classical Deployment Architecture

While we were discussing Web-Based Deployment Architectures – I was insisting on separating DB logic into a separate DB Server App, which should be the only entity speaking directly to the database and knowing about SQL[92]. For Classical Deployment Architectures, I insist on exactly the same thing, so our architecture diagram from Fig 9.7 becomes similar to the following Fig 9.10:



The idea here is *exactly* the same as the idea behind separation of the DB Server App for a Web-Based Architecture. In short – we're aiming to eliminate too-tight coupling, to isolate game logic from SQL, and to make DB structure an implementation detail of the DB Server App. In turn, it will allow us to have a very efficient DB Team, which will be able to optimize DB structure without affecting our other Apps – and, as a result, will be able to *take responsibility* for the whole database. For more discussion on the benefits of the separated DB Server App – see the *Enter DB Server App* section above.

An all-important reminder: for this approach to work as intended,

[92] the only exception to this rule MAY be read-only reports.

DB Server API (the one rest-of-the-world uses to communicate to DB Server App), MUST be expressed in terms of game logic (with no SQL/NoSQL in sight)

Given that the whole point of DB Server App is to *isolate* our game logic from the database implementation details, the requirement above should be quite obvious – however, in practice it can be forgotten way too easily (which, in turn, would lead to re-instatement of the tight coupling which we're trying to deal with in the first place).

Classical Deployment Architecture: Scaling and Load Balancing

To scale a very generic Classical Deployment Architecture – we'll need to scale all our Game Servers plus a DB Server App.

For the time being, we'll concentrate only on scaling Game Servers; scaling of DB Server App is a completely different story – but it is more local problem (i.e. it can be seen as an implementation detail of the DB Server App) – so we're able to postpone discussions on DB Server Scalability until Vol. VI's chapter on Databases.

Scaling Game World Servers - Natural Linear Scalability (except for seamless MMOs)

For most of the games out there, scaling Game World Servers is not difficult. Indeed, if our Game Worlds are small enough – we won't have problems to handle each of them with one single thread (if using (Re)Actor-fest architecture – with one single (Re)Actor), and then we can just instantiate as many of them as we need. For example, if your game is a battle arena consisting of matches (each match having like 10 players) – this very simple approach will allow for trivial scaling.

For such games, our diagram on Fig 9.10 corresponds to Stateful-App-Based System discussed in Chapter 8 (the one relying on In-Memory State). It means inheriting all the properties of Stateful-App-Based Systems discussed in Chapter 8; let's take a closer look at them, and see how these properties interplay with specifics of MOG and Game Servers:

- DB load is greatly reduced compared to simple Stateless-Apps (compared to Stateless-Apps without Write-Back Caches). In practice, this helps A LOT to scale DB to those numbers which we might need.
- Stateful-App-Based Systems have In-Memory State which is inherently non-durable[93]. It means that if our Game World Server crashes – we'll lose this In-Memory State. On the other hand – as discussed in Chapter 8, for most of MOGs out there, in case of crash rolling-back-to-the-start-of-Game-Event is *exactly* what we want anyway.

[93] At least unless we're going into fully fault-tolerant systems, which are usually not really realistic for MOG Game World Servers.

- Load Balancing can be not-so-trivial. Moreover – as discussed in Chapter 8, for Stateful-Apps there are *two* different kinds of Load Balancing: Worlds-to-Servers Load Balancing, and Clients-to-Servers Load Balancing.

Let's discuss Load Balancing for Classical-Architecture-as-shown-on-Fig-9.10, in more detail. Actually, for the architecture shown on Fig 9.10, we don't have any ability to balance Clients-to-Servers (*all* the Clients who are playing/viewing at specific Game World, have to be processed by that specific Game World, that's it). This can be acceptable for quite a few games out there, but – *if* you want to have spectators, *and* some games will have LOTS of such spectators – you'll be in trouble; this can be addressed by adding Front-End Servers (more on them in [[TODO]] section below).

As for the Worlds-to-Servers Load Balancing – it is usually achieved by measuring the load of each of the Server Boxes, and reporting it to the Matchmaking Server. And whenever Matchmaking Server decides to create a new instance of Game World – it does in on the least-loaded Server Box (potentially allocating new Server Box from the cloud if such an ability exists). Ways to measure Server Box load vary from counting number of Game Worlds per Server Box, to real CPU load measurements (BTW, the latter is not necessarily "better" than the former, due to phase shifts and potential for positive feedback loops and oscillation); for more detailed discussion on it – see Chapter 8.

Moving Game Worlds Around

Up to now, we were speaking about balancing load by *creating* an instance of Game World (such as Game World (Re)Actor) on a specific Server box. On the other hand, in some cases it may be necessary to move already-created Game World instances around to address imbalance which may occur *after* we already created all our Game World instances.

This is possible – however, from what I've seen in practice, for Game World instances it is rarely necessary as long as (a) all your servers are rented on per-month basis, *and* (b) each Game World is small enough, *and* (c) lifetime of Game Worlds is limited. If, however, *any* of these requirements doesn't stand for your game – you'll likely want to move your Game Worlds around. In particular, if you're renting some of your cloud servers on per-minute basis – you *will* want to consolidate your Game Worlds onto as-few-rented-servers-as-possible as soon as the load allows it.

And, if you need to move your Game Worlds – at least with VMs and/or (Re)Actors it is perfectly doable.

For VMs, Game World relocation is going to take a while; for example, (Predicting the Performance of Virtual Machine Migration n.d.) gives VM migration times of the order of single-digit seconds(!); still, as it is a one-time latency – even this can be bearable for quite a few games slow- and medium-paced games out there.

For (Re)Actors, this relocation latency can be reduced to single-digit milliseconds, making it viable even for the most-latency-critical games out there. For a brief discussion of implementation details for such low-latency (Re)Actor relocations – see the *Moving Game Worlds Around (at the cost of Client reconnect)* section above.

MMOGs and Seamless Game World Servers

For our Scalability analysis above, there was an all-important assumption[94] – it is that our Game Worlds are small. This immediately leads us to a question – what to do if our game is one single and seamless Game World with tens of thousands of players in it?

Of course, if we can easily split our large Game World into smaller "zones" (with no ability to see to a different zone – such as in most of RPGs out there where each house/city/… is a separate zone) – we'll be fine with just minor additions to the model above. But what if we're not (i.e. if our Game World is large enough and seamless too)?

As we've already mentioned in Vol. I's chapter on Communications, probably the most popular way of scaling MMOGs with seamless Game Worlds goes along the following lines:

- We're splitting our One Big MMOG into many "zones"
- We're saying that on "zone" boundaries there is an overlapping area (large enough to be able to show the player who resides on the edge of this overlapping area, all the potentially movable objects within his field of view)
    - Moving objects (such as PC/NPCs/…) within this overlapping area are simulated TWICE – once in each of zones. However, only one of the copies is authoritative, so that there is a constant re-synchronization going on between the overlapping "zones" – with non-authoritative copies being adjusted to correspond to the authoritative one.
        - This adjustment is roughly similar to that of described in Vol. I with respect to Client-Side Prediction. In other words – we can think that simulation of the non-authoritative copy is just a prediction (made by a Server).
    - At some point, authority over an object which crosses the boundary between zones, is transferred from one Zone Server to another one. After authority transfer, both Zone Servers still continue to simulate the object – but a different Zone Server becomes authoritative.
- As soon as we're done with "Zone Servers", we can treat them more or less the same as our usual Game World Servers which we discussed above.
    - NB: unlike usual Game World Servers, Zone Servers do need to communicate to each other; while rarely being a problem when all your Zone Servers run from one single datacenter – it can have important ramifications when spreading over different datacenters, so your Load Balancing algorithms will likely need to keep it in mind.

For much more detailed discussion about this approach – see, for example, (Beardsley n.d.), (Duquette n.d.), and (Baryshnikov n.d.).

[94] And, as all of us experienced multiple times, any assumption is a potential source of screw-ups

Scaling Matchmaking Server

Another Game Logic Server which can cause scalability issues – is your Matchmaking Server. In its regard, unfortunately, I can offer only two considerations:

- As a rule of thumb, Matchmaking Server is relatively rarely accessed; also it is usually simple enough and fast enough. As a result, you can usually avoid dealing with scalability for a long while (usually - up to 100K simultaneous players and more).
    - As always, YMMV and batteries not included <sad-face />
    - It will even work from one single thread/(Re)Actor for a long while
- When handling matchmaking from one single thread becomes a problem – it is usually possible to split it so you can handle necessary load; however, the ways of splitting which I've seen, were too game-specific and I am not able to generalize them into anything universally-usable at the moment. In other words – while I am sure that achieving scalability for your Matchmaking Server is *certainly doable*, I cannot provide any non-game-specific advice how to do it, so you'll to think about it yourself <sad-face />.

Scaling Other Game Servers

As for other Game Servers – they usually either scale similar to the Game World Servers above (one example of these is Tournament Servers), or they don't need to scale at all (i.e. all needed things can be handled by one (Re)Actor), or they're trivially scalable via some kind of balancing proxy.

Let's discuss the last one (scaling by proxy) by an example. Let's say that we have a Social Gateway Server which needs to send LOTS of Facebook updates (of course, by player's request – but number of these requests happens to be huge for our game). First, we had it implemented as one (Re)Actor; however, at some point we found that one thread cannot handle all the load. At this point, we can just create a kind of balancing proxy – which will have an interface *exactly* as our usual Social Gateway Server, and several instances of Social Gateway Server behind the proxy. All requests from the rest of our system go to this Social Gateway Proxy, which merely forwards the request to one of the instances of Social Gateway Server sitting behind it. This schema tends to work pretty well – in particular:

- proxy can be inserted at later stages without any change to the rest of the system,
- I've never seen balancing proxy to become overloaded. Indeed, with a single thread able to proxy around 100'000 messages per second – overloading such a proxy becomes a very difficult task (just to put it into perspective – the whole Twitter on average has only 6'000 tweets/second).
- The only major caveat with such proxies is that they imply that the Game Server being proxied, is essentially stateless – otherwise this kind of proxying wouldn't work; on the other hand, surprisingly large number of those-other-Servers-which-need-scaling, are indeed stateless. Overall, I didn't see a need to scale some of the "other" (i.e. non-Game-World and non-Matchmaking) Game Servers beyond balancing proxy.

Classical Game Deployment Architecture: Summary

To summarize our discussion above about Classical Game Deployment Architecture:

- It works even for most fast-paced games out there <smile />
    - In fact, for fast-paced games it is *the only* viable option
- It can and often should be implemented using (Re)Actor-fest model with deterministic (Re)Actors, see discussion above for details
    - Deterministic (Re)Actors provide LOTS of benefits, from production post-mortem to replay testing using real-world data, and so on, and so forth
    - (Re)Actor-wise, I Very Strongly suggest to write your Game Logic as (Re)Actor
    - Writing your infrastructure-level code as a (Re)Actor, while certainly possible (as described above) and I suggest to do it too – is not *that* important as implementing your Game Logic as a (Re)Actor.
- As a rule of thumb, for stateful processing (which is inherent to Classical Deployment Architectures) MWSR-based architectures tend to beat MWMR-based ones (the latter include thread pools).
- DB Server App (with DB Server API expressed in terms of game logic – and without any SQL/NoSQL details) – is *highly desirable* not only for Web-Based Deployment Architecture, but also for Classical Deployment Architecture.
- Under Classical Deployment Architecture, scaling of Game World Servers is usually trivial
    - One exception is "seamless worlds", but methods of scaling seamless worlds are known too.
- Classical Deployment Architecture as such addresses only Worlds-to-Servers Load Balancing; for Clients-to-Servers Load Balancing – we'll need to add Front-End Servers (discussed in [[TODO]] section below).

"Hybrid" Web+Classical Architecture (Mixed Stack)

As it was already noted above – it *is* possible to combine Classical Deployment Architecture with a Web-Based one. In this case, traditionally a deployment diagram looks along the lines on the following Fig. 9.11:

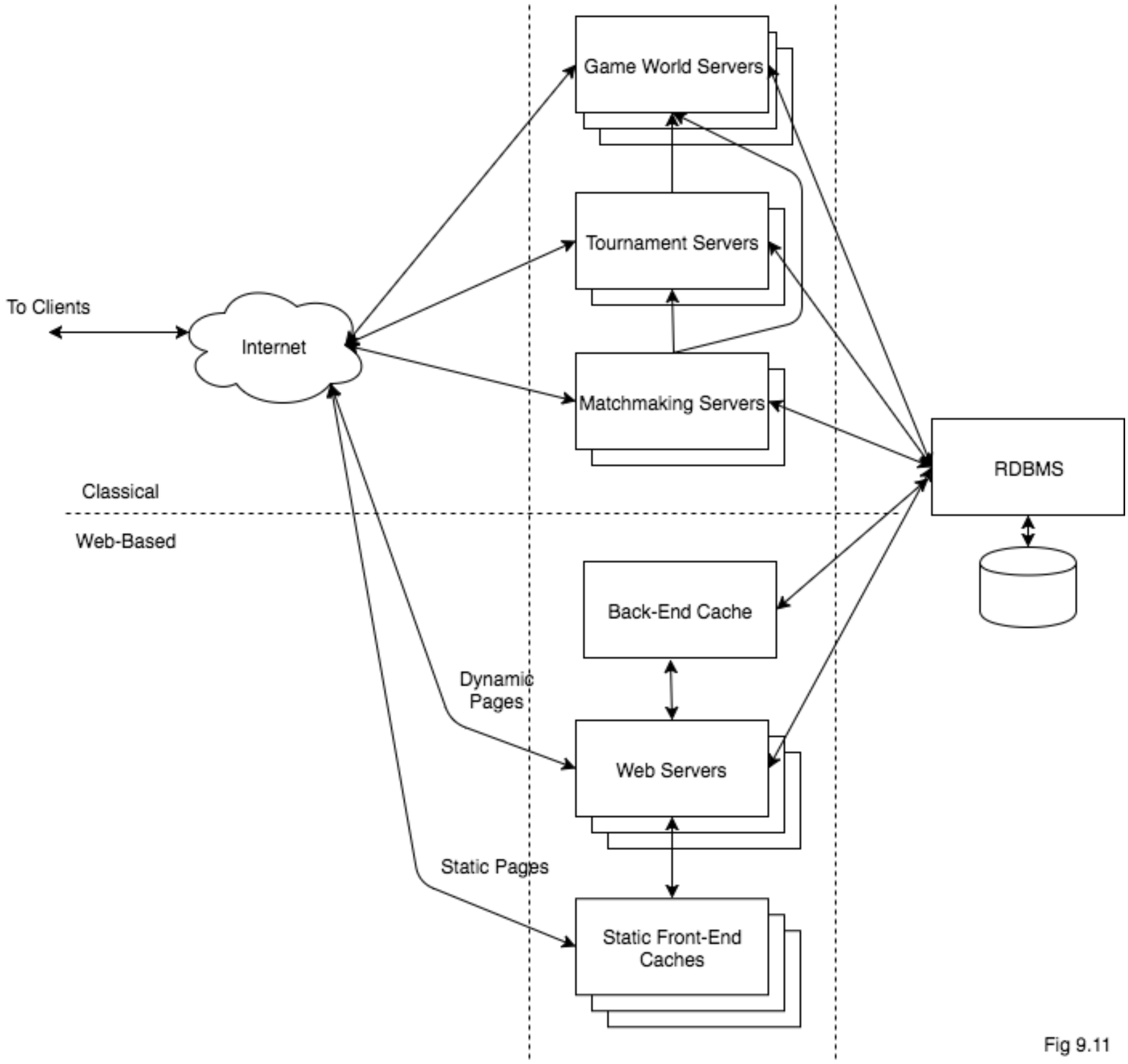


Fig 9.11

As we can see, "hybrid" architecture on Fig 9.11 is nothing but a merge between "Web-Based Architecture" from Fig 9.4, and "Classical Architecture" from Fig 9.7. Such "hybrid" architectures directly correspond to "Mixed Stack" from (Zubek, "Engineering Scalable Social Games" n.d.). Overall, I am not going to spend too much time describing such hybrid (or "Mixed-Stack") architectures; as discussed in (Zubek, "Engineering Scalable Social Games" n.d.), they surely have their merits, but these merits directly follow from description of Web Servers described under Web-Based Deployment Architecture, and from "Classical" Servers being an incarnation of Game Servers described under Classical Deployment Architecture. Very shortly – yes, you can combine Web-Based and Classical Deployment Architectures in one game, and you'll obtain Fig 9.11 <smile />.

For such "hybrid" architectures, implementing Server-Side is rather straightforward (just implement web-based part along the lines of Web-Based Architecture, and classical part - along the lines of Classical Architecture as discussed above). However, there are two important notes to keep in mind while combining these different approaches together:

- Both parts of the stack MUST speak to the same database behind. It should be quite obvious – but let's state it explicitly just in case.

    - The only exception-I-know to this rule, occurs when we want to save Game States into the database – in such cases, it is ok to have separate dedicated DBs specifically for Game States (in a manner similar to Fig 9.6 above).
- Your Client will need to work with both of these quite separated stacks. As a result – Really Ugly™ solutions (with effectively two separate Clients – one downloadable for play, and one browser-based for everything-else) are often used on the Client-Side. In Vol. II's chapter on Client-Side Architectures I already bashed these dual-Client solutions (at the very least, (a) they're ugly from player perspective, and (b) they make password phishing easier). On the other hand, it is only a Client-Side issue; moreover - it *is* possible to get browser integrated into downloadable Client (or, in some cases - to get gameplay integrated into browser-based app); see Vol. II's chapter on Client-Side Architecture for a relevant discussion.

Overall, personally I am not really a big fan of such hybrid architectures; the reason for it is mostly because keeping too many different technology stacks together – and making them cooperate too (especially on the Client-Side) - can easily become more cumbersome than DIY. On the other hand, as my concerns are relatively mild, and as I know for sure that such "hybrid" deployments worked in Quite Large™ social game deployments (ok, as it was Zynga – I have to admit it was Really Large™ <smile />), it means that if you will insist on implementing your game this way – I won't jump *too* high <wink />.

DB Server App for Mixed Stack

While, as mentioned above, I agree that in quite a few cases, such "hybrid" (Mixed Stack) architectures may be viable – there is one thing I have to insist on if you decide to go this way. As you may already have guessed <wink /> - it is using a DB Server App instead of allowing all the web servers, and all Game Servers to issue SQL statements directly (also, DB Server App will take over functions of the Back-End Cache). Such an architecture is shown on Fig 9.12:

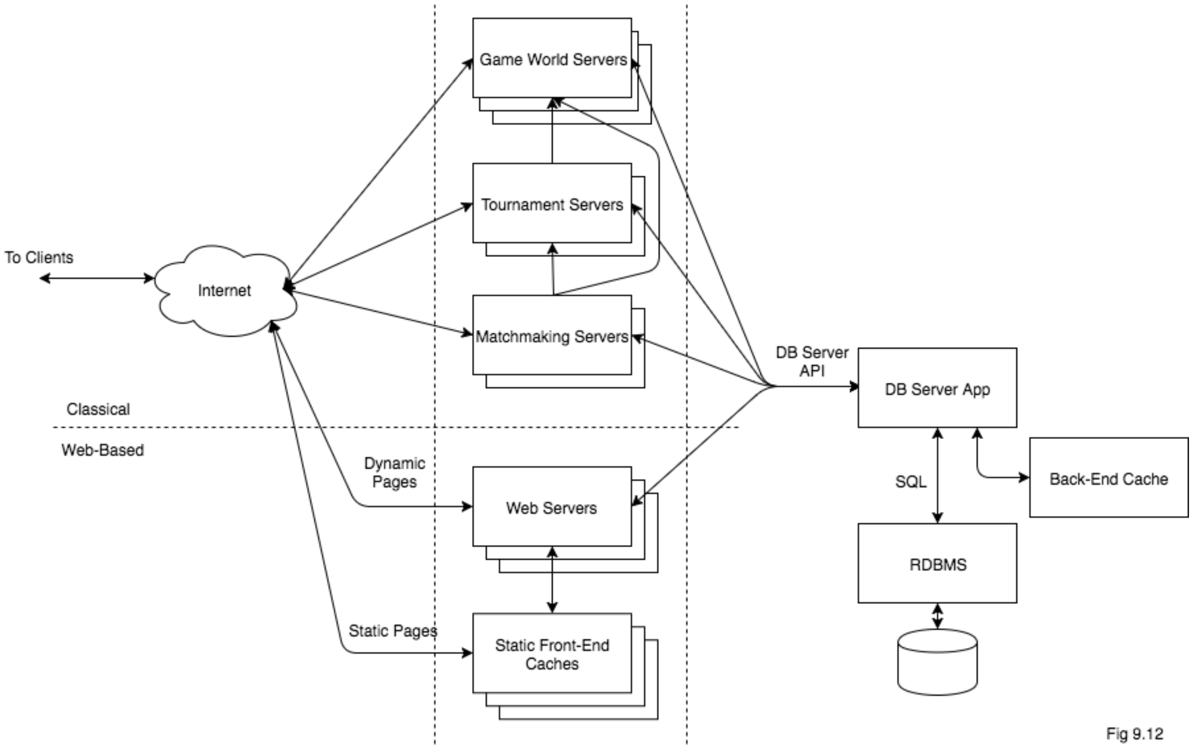


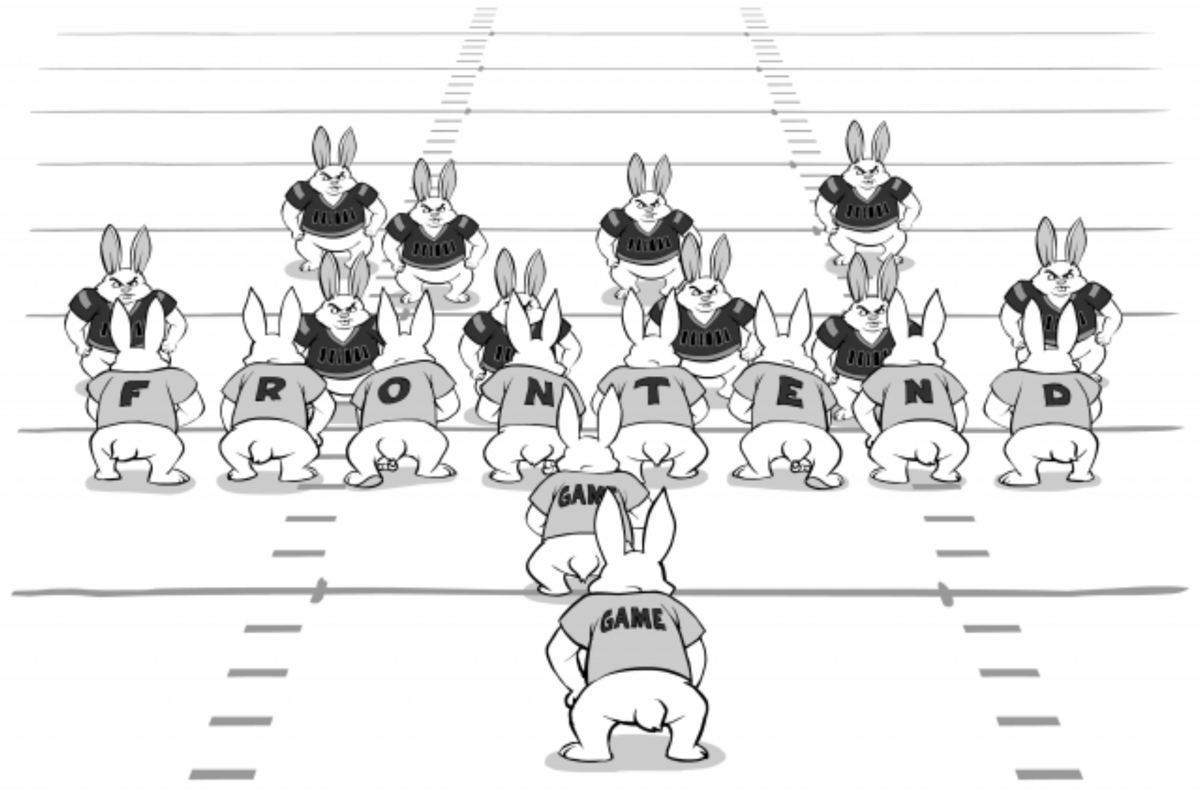
Fig 9.12

This approach is a hybrid of previously discussed ones, so I won't spend too much time on it, merely re-iterating the most import points:

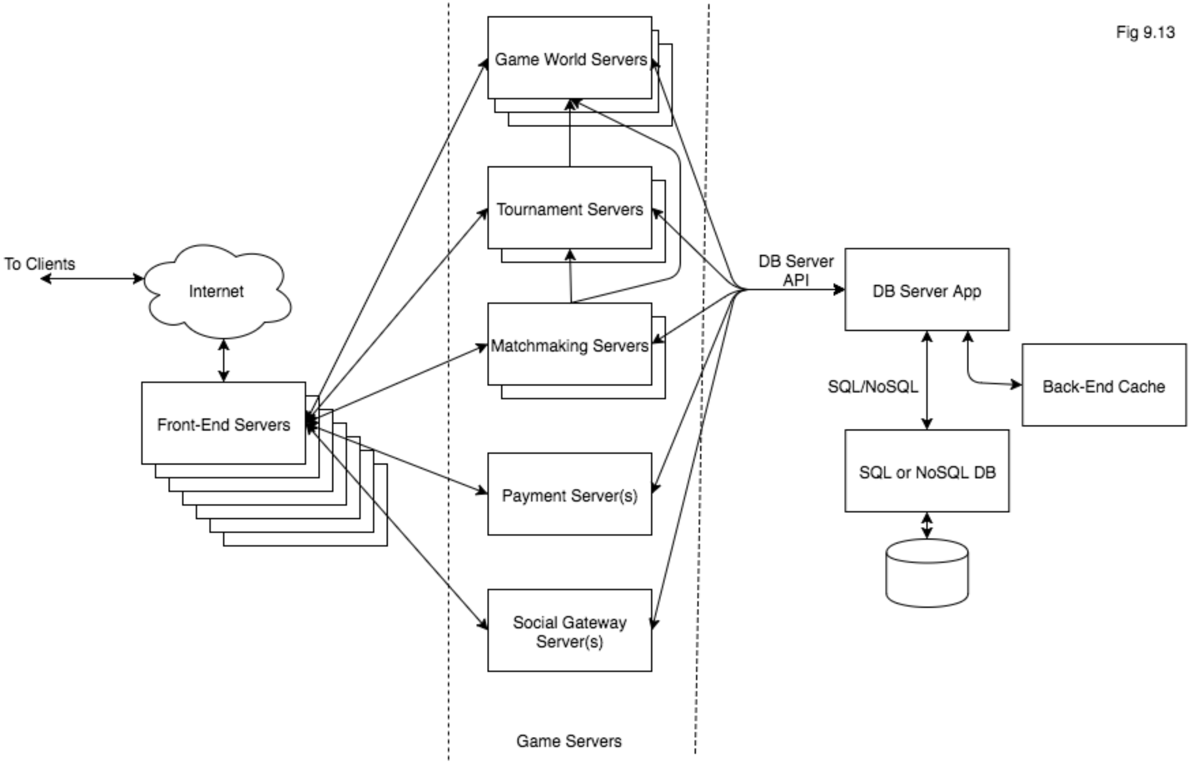
- All entities SHOULD speak to the same DB Server App (with the only potential exception-I-know being separate Game State DBs, along the lines of Fig 9.6).
- DB Server API MUST be expressed in terms of Game Logic (and *not* in terms of SQL/NoSQL).
- As discussed in Vol. II's chapter on Client-Side Architecture – your Client SHOULD NOT *look* as being split into "downloadable" and "web-based"; integrating these parts of the Client is *important* for your bottom line (especially if competition is significant).

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 on Fig 9.10 is certainly not bad, and it will work, but there is still quite a bit of room for improvement for quite a few games out there. More specifically, we can add another row of servers in front of the Game Servers,[95] as shown on Fig 9.13:



[95] BTW, we can easily do it for "hybrid" Web+Classical Architectures too

As you see, compared to the Classical Deployment Architecture (as shown on Fig 9.10 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 different kinds of keep-alives where applicable, see Vol.IV’s chapter on Network Programming 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.

We'll discuss the implementation of our Front-End servers a bit later, but for now let's note that most importantly,

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 orders of magnitude better than to see your game hang).

Front-End Servers as Concentrators

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

In addition, these Front-End Servers can store a copy of relevant Game Worlds when it is necessary, 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 of the Year 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 tens of thousand people willing to watch it; most of the time – for marketing/monetization purposes you’ll want to broadcast it *both* as a video stream (for those who don’t have your Client yet) – *and* as a game data stream residing with your Client (so in your video stream you can say “hey, if you want to see it in real 3D – please download our Client”,[96] not to mention it being

[96] and whenever you can provide a compelling reason to download your Client – you’re getting more players, it is this simple.

more convenient to existing players, and game-data broadcast using much less traffic and being much less expensive as a result[97]).

As for the protocols for these Front-End-Servers-acting-asconcentrators – they can be implemented based pretty much on the same protocols which were discussed in Vol. I's chapter on Communications. In particular:

- "concentrator" can act as if it is a "Client", subscribing to Game World Updates and keeping its own eventually-consistent copy of the Game World
- when a new Client comes to concentrator –
  - concentrator feeds the Client with an (almost-)up-to-date information from its own copy of the Game World, *and*
  - concentrator adds the Client to the list of downstream subscribers
- on receiving any Game World update from its own upstream provider – concentrator:
  - updates its own copy of the Game World, *and*
  - sends the update to all its downstream subscribers

Bingo! We've got our "concentrator" to serve thousands of subscribers – and without loading its upstream at all.

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:

- having a copy of relevant Game World(s) on your Front-End Servers (i.e. with Front-End Servers acting as "concentrators") allows to replicate your Game World State to a virtually unlimited number of observers/spectators. This comes handy in at least two common scenarios: (a) broadcast of your Big Final Game™ (with hundreds of thousands who want to watch it in real time), and (b) broadcasting lists of available games (for players to select which game they want to join) – and these can become quite large as soon as you're allowing to select your opponents directly. Best of all, any such broadcast will happen without affecting Game Server's performance (and this is the last thing you want to happen to your Big Final Game™). Moreover, with Front-End Servers usually you won't need to organize anything special for your Big Final Game™, and 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

[97] For video stream – we're speaking about *at least* 3Mbit/s to get *anywhere-decent* image; with game data stream – it is more like 100kBit/s to get *absolutely perfect* picture (plus an ability to change angles, etc. etc.). So, from the players' point of view (and players don't caere about complexity of Front-End Servers,) – the choice is a no-brainer.

    - 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 1'000'000 observers watching some specific game on this Game Server, all the additional load is spread onto your Front-End Servers, without affecting your Game Server
- Front-End Servers take some load off your Game Servers, while being easily replaceable
    - it means that you can have fewer 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 – that is, without cumbersome fault-tolerant stuff discussed in Chapter 10. In other words – with Front-End Servers, instances when some of your Game World Servers go down, will occur more rarely (!)
        - in particular, it means that 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 almost-certainly do need them for your Game Servers – at least unless you're into real fault tolerant solutions described in Chapter 10). If you're going to deploy into the cloud – it means that you may want to consider cheaper offers for your Front-End Servers (even if they're coming from different CSP).[98]
        - Note that to get this benefit from Front-End Servers, it is necessary to ensure that if one of Front-End Servers goes down – its load should be automagically redistributed among the others (in practice, Client-Side Random Load Balancing, as discussed in Chapter 8, was seen to work extremely well with Front-End-Servers).
    - Let's note that positive effect from reducing the number of Game Servers depends on the portion of the load which is took by communications compared to the Game World logic. In particular, for simulation games – the gain is likely to be rather negligible.
- they allow your Client to have a single connection point to the whole site (or at least to minimize the number of such connection points); 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"
    - hiding more implementation details of your site from the hostile world outside.
    - More on it in Vol. IV's chapter on Network Programming
- Front-End Servers allow for better security, in particular:
    - they can be separated in a kind of DMZ, see further discussion in Vol. IX's chapter on Security, Take 2).

[98] keep in mind that you still need top-notch connectivity

    - To attack your Front-End-based system (the one which also uses Client-Side Random Balancing) with a DDoS, it is necessary to overload *all* your Front-End Servers simultaneously. While this is certainly possible – I've seen *lots* of DDoS attacks being unable to get the game, exactly because of overloading just one Front-End Server.
        - Most of the time, you'll still need another layer of DDoS protection (such as BGP-based one – more on it in Vol. IX), but Front-End Servers will help you to avoid activating BGP-based protection more-often-than-otherwise-necessary. As each such activation is disruptive to your players for sure (and having too many of such activations *may* become expensive) – having to use such protection less frequently is certainly a Good Thing™.

Front-End Servers: Drawbacks and Issues to Solve

As for the potential negative sides of having Front End Servers, there are quite a few of them too (nothing in this world comes for free, sigh); we'll discuss them in more detail so you'll be able to decide whether they apply to your specific game – and how to deal with them.

Latencies and Latency Differences

The most obvious potential drawback of Front-End Servers is additional latency. More specifically, we're speaking about the time which is necessary for the packet incoming from a Client at application layer, to get deserialized from Ethernet wire by the NIC of the Front-End Server, to generate hardware interrupt, for OS kernel to receive and process this interrupt, for kernel thread to pick up results of interrupt processing and to get the packet through IP stack all the way to app level (this includes kernel->user level transition), to be processed by your Front End Server app-level, to go back into TCP stack[99] on Front-End Server side (this includes user->kernel transition), to be sent to NIC and then to be serialized to the Ethernet wire (plus the time necessary to go in the opposite direction).

[99] as it was discussed in Vol. I's chapter on Communications, I'm usually arguing for TCP connections for inter-server communications in most cases. On the other hand, UDP is also possible if you really really prefer it

It may sound scary, but let's take a closer look at this additional latency. For a 100-byte packet transmitted at 1GBit/s, serialization/deserialization times will be around 1 microsecond; for NIC-interrupt interactions (both transmit and receive combined), (Steen Larsen n.d.) gives an estimate of around 8 µs; their measurements, however, don't include kernel-to-user switches and back; these may cost us several additional thousands of CPU clock cycles, which corresponds to another several microseconds (in some cases this can be improved BTW, see brief reference to *netmap*/*DPDK* RIO in "Network Processing" section below, and more detailed discussion in Vol. IV's chapter on Network Programming). And processing within the app-level of the Front-End Server can be usually brought down to double-digit microseconds.[100] Finally, we need to multiply all these delays by two to get to RTT.

we're speaking about the additional latencies below 100 µs; in practice, I've seen it a bit higher – at 200-500 µs, but it was well below 1ms anyway.

Still, if we do all the math, we'll find that we're speaking about these additional-latencies-due-to- below 100 µs; in practice, I've seen it a bit higher – at 200-500 µs, but it was well below 1ms anyway.

Let's note though that if you're using an inefficient library (which does happen, especially with not-so-mainstream libraries, and even worse – with not-so-mainstream programming languages) – the latency can be MUCH higher. While I am comfortable to say that on a x64 box running Windows or Linux, it *is* possible to achieve delays which are well-below 1ms using plain C on top of OS calls – I cannot vouch for doing it with each and every communication library out there; all 3rd-party libraries need to be tested in close-to-real-life tests to be sure <sad-face />.

Another latency-related 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. with reasonably good Load Balancing – and as I've seen it in practice - if one of the Front-End Servers is overloaded, it means that most of the other ones are already at 90%+ of capacity, which you should avoid anyway; see also Chapter 8 for discussion of the ways to implement 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 *maybe* MMOFPS).

To summarize:

[100] note that this might become a non-trivial exercise; on the other hand, I've done it myself (reaching as little as 5 µs for some classes of traffic), so it is certainly doable at least in some practical cases. On the other hand, YMMV depending on the processing involved.

- if additional latency of around 1 millisecond is ok for you – don't worry too much about additional latencies and go for Front-End Servers; this certainly covers all genres with the only potential exception being MMOFPS
- still, make sure to test additional latencies for your own libraries before committing to Front-End Servers; while achieving 1 millisecond is certainly possible – you may run into problems with your communication library; during my career I've seen Really Weird™ performance-related things done even by popular libraries.
- 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 a MMOFPS world pretty much anything can happen) - think about it a bit more – and experiment more too. My guess is that you should still be able to reach even hundreds-of-microseconds range for Front-End Servers – but don't rely on it until you see it with your eyes in a well-conducted experiment (ideally – by measuring RTT with and without your Front-End Server, all other measurements can mislead way too easily).

Discussion on Scalability of Front-End Servers and Dealing with "N-squared". Server Groups

Another concern with regards to Front-End Servers is that as your whole game grows, your inter-server traffic and RAM usage across all the Front-End Servers will grow as N^2, so whenever we're increasing capacity by a factor of two, they will grow by a factor of 4.

While it is a theoretically valid argument (and a popular one among the serious gamedevs too), I used Front-End Servers in a quite big project, and didn't see this "N-squared problem" starting to play any role in practice. Moreover, if "N-squared dependency" ever becomes an issue, it is certainly possible to avoid N^2 completely – by effectively splitting your site into several "server groups" of N Front-End Servers by M Game World Servers, and balancing your Clients across these N Front-End Servers.

In the extreme case (and this is what quite a few guys from the industry are arguing for) – you can reduce these NxM server groups into 1xM server groups (i.e. with one Front-End Server per "server group"). However – personally I am a rather strong advocate of NxM approach with N being around 3-5 (and if not causing too much trouble – raising N up to 10-20; BTW, from my experience - a single group with N~=20 has good chances to cover your whole game, even if it runs a hundred of thousands of simultaneous players).

Let's do some very basic math (of course, for your game the numbers will be different – and make sure that you repeat the calculation using your own numbers, but the idea– though not necessarily results - will stay more or less the same). Let's say your game is an arena with 100K simultaneous players distributed over 10K Game Worlds, which Game Worlds are dispersed over 100 10-core Game World Servers[101]. Also, let's say we have 10 Front-End Servers to handle all this traffic (10K players/Front-End Server is a reasonably good ballpark

[101] as of 2017, 100 players/core or 1000 players/2S server is more or less a "de-facto industry standard number" observed across a wide range of simulation-related games

starting number). Let's further assume that each of Game World Publishable States takes 100K of RAM (BTW, when speaking about Publishable States – as defined in Vol. I's Chapter on Communications - this is quite a generous number), and generates 50 bytes per PC per network tick (with network ticks coming 20 times per second), plus it has 100 NPCs which are much less mobile and generate on average 10 bytes per network tick each.

This makes each of Game Worlds to generate 50 bytes/PC/tick * 10 PCs/GameWorld * 20 ticks/second + 10 bytes/NPC/tick * 100 NPCs/GameWorld * 20 ticks/second = 30Kbytes/second/GameWorld of traffic. With our NxM matrix being actually 10 Front-End Servers by 100 Game World Servers – these 30Kbytes/second need to be sent to 10 of our Front-End Servers, making it 300Kbytes/second per Game World, or 300Kbytes/second/GameWorld * 10K GameWorlds = 3GByte/second of total traffic between our Game World Servers and Front-End Servers. While this traffic is not going to leave boundaries of the datacenter (and you're not going to pay for it as for the traffic) - this is still quite a large number and indeed sounds scary. However, we need to keep in mind that it is a *total* traffic, and that no server in the system will ever see this much: in particular, each of 100 Game World Servers will send only 1/100th of it (i.e. 30MByte/s or 300Mbit/s), and each of Front-End Servers will receive only 1/10th of this traffic (i.e. 300Mbytes/s or around 3Gbit/s – rather high, but can be handled by 10GBit/s interfaces which are already pretty much standard as of 2017 on "Workhorse" 1U/2-socket boxes[102]). The only entity in the system which will experience this 1GByte/s = 10GBit/s traffic, is your *Ethernet switch*, but as of 2017, if you have a 128-or-so-port switch (to connect all those server boxes) – you'll usually get *much* more than 10GBit/s switching capacity anyway.

From the point of view of RAM – in the absolutely worst case, we'll use 10K worlds*100K bytes/world=1Gbyte RAM to store each all those 100K Game World Publishable States on each of our Front-End Servers; as 1G is not much to worry about by today's standards, it means that again, we don't have much to worry about in practice.

As a result, I don't envision any problems if for the game like the one we've just discussed, we're using NxM matrix of 10 Front-End Servers x 100 Game World Servers to handle 100K simultaneous players. And let's keep in mind that we've got a fairly large game to start with (100K of simultaneous players is nothing to be ashamed about even these days).

On the other hand, if we'll extend this game to handle a *million* of simultaneous players – probably the most cost-efficient way would be to avoid dealing with too much of N^2 dependencies, and to have 10 of such 10x100 server groups.

One may ask – if having server groups too large causes all this overhead, why can't we have just 1xM server groups? (and as I noted above – it is an approach which is rather popular within the industry). The reason why I am arguing for having more than one Front-End Server in an NxM server group – is to have redundancy, fault tolerance, and resilience-to-naïve-DDoS-attacks between these N Front-End Servers. To have this kind of redundancy –

[102] If your Front-End servers happen not to have 10GBit/s interfaces – you can always increase the number of your Front-End Servers or to use smaller server groups as described below; in any case, this problem is not fatal

we need at least two Front-End Servers (i.e. 2xM server group), but then each of them will need to be able to handle all the traffic of the whole server group, which leads to the need for 100% redundancy and not-so-efficient configurations; if we have at least 5 Front-End Servers – we can keep reserves within much more reasonable 25%, while having the redundancy/fault tolerance – and also Load Balancing without any Server-Side Load Balancing boxes too. In addition, if you're about to broadcast one single game (whether at real-time, with some delay, or recorded) to a large chunk of your players, it is usually better to balance this load across as many Front-End servers as possible (which calls for larger server groups); on the other hand, if the need arises, the same Front-End Servers can simultaneously participate both in smaller server groups (for usual traffic), and in larger server groups (for broadcast traffic).

An example of such deployment is shown on Fig. 9.14:

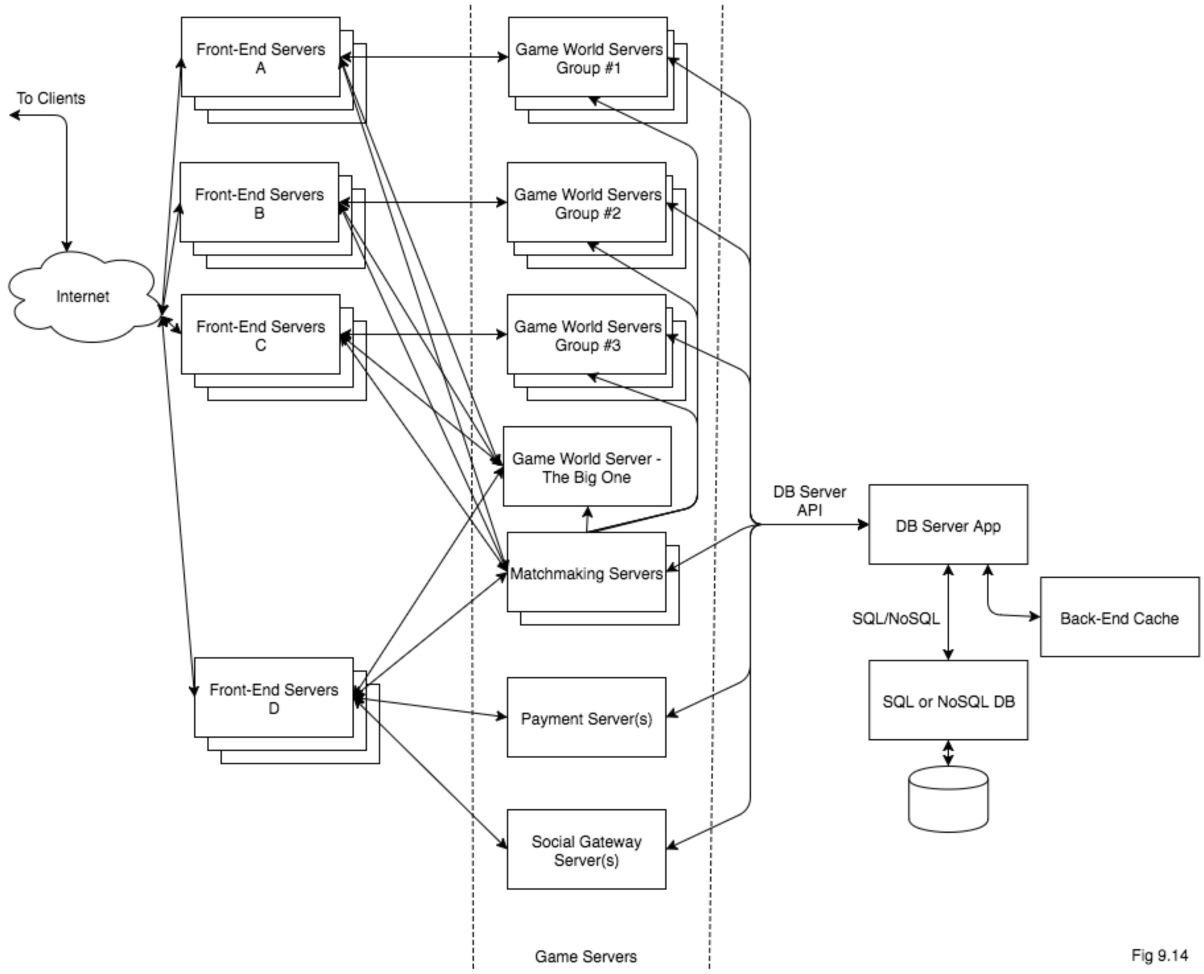


Fig 9.14

On Fig 9.14, we have 3 groups of Game World Servers (#1, #2, and #3), and four bunches of Front-End Servers (A,B,C,D). Each of groups of Game World Servers is using one bunch of Front-End Servers; Payment Servers and Social Gateway Servers are using dedicated bunch D of Front-End Servers. This allows to have some balance of the load between the Front-End Servers. However, there are two Game Servers in this system, which are (by the nature of the game) broadcasted to pretty much all the players – these are The Big One Game Server (running a Match of the Year or something else which everybody wants to watch), and

Matchmaking Servers (which for this game may publish the matchmaking data, allowing players to select games-in-the-process-of-planning themselves); these servers are connected to all the Front-End-Servers.

This whole deployment architecture allows to achieve two things: first, it balances things-which-are-of-interest-to-everybody, across *all* the available Front-End Servers; second – it avoids N^2 problem (as both broadcasted events and Game Server Groups are O(N) – the latter as soon as we put a hard limit on number of Game Servers and Front-End Servers within one group).

Overall, the number N of Front-End Servers within each server group qualifies as a deployment-time implementation detail; however, our job as developers is to make sure that this implementation detail can be changed as desired (and on per-published-State basis too) without any code rewriting.

Game-Server-to-Front-End-Server Affinity

"Server groups" can be further generalized into what can be called "Game Server to Front-End Server Affinity". In this case, players which need to connect to certain Game Servers, can use only a subset of the Front-End Servers (reducing or completely eliminating the N^2 effects). In a very generic case (covering pretty much everything you may possibly want), it could work as follows:[103]

- there is a way for the game Client to get list of IPs of Front-End Servers (the list can be either embedded into the Client, or obtainable via DNS etc.)
- to connect to the Matchmaking Game Server, Client can go to any of the Front-End Servers in the list (and request connection to Matchmaking Game Server by a well-known ID).
- when matchmaking is done – Client receives ID of the Game Server where it needs to connect, PLUS a list of IP addresses of Front-End Servers (or a DNS name which maps to such a list, see on DNS Round-Robin Balancing below) which can handle such a connection.
    - At this point, Matchmaking Game Server can balance not only Game World instances between different Game World Servers (named Worlds-to-Servers balancing in Chapter 8), but also Clients between different Front-End Servers (implementing Clients-to-Servers Balancing discussed in Chapter 8).
    - It should be noted that in this schema, I still insist on providing a *list* of IPs, with at least 3-5 different Front-End Servers on this list (="at each point, Client having at least 3-5 different Front-End Servers to select from"); this is still necessary to speed up dealing with Front-End Server failures, to deal with "point" DDoS attacks, etc.
- Then, Client connects to one of the Front-End Servers from the list (randomly, see discussion in "Client-Side Random Balancing" section in Chapter 8), and requests

[103] that is, if Client-Side Random Load Balancing or DNS Round-Robin Balancing is used; for Server-Side Appliance Balancing, pretty much the same logic can be performed by the balancing appliance

Front-End Server to connect to (or to subscribe to the information published by) Game Server by its ID.

- If the player just wants to observe – her Client can request a list of the games in progress (from Matchmaking Game Server or a separate Game Server set up just for this purpose) – and can receive the same (Game-Server-ID,list-of-Front-Server-IPs) tuple for each of the games. Once again, balancing of Clients between different Front-End Servers can happen here, allowing for broadcasts to all of your 100K players (and without introducing N^2 problems).

Also let's note when we're speaking about Game-Server-to-Front-End-Server Affinity, this "affinity" is quite different from classical web-like affinity (usually referred to as "persistence" or "stickiness") as used on Load Balancer Appliances for web servers. In the web world, affinity/persistence/stickiness is all about having the same Client coming to the same Server-sitting-behind-the-Load-Balancer-Appliance (to deal with sessions and per-client caches). For our Front-End Servers, however, affinity is very different: we do not care about the same Client coming to the same Server, but rather about the Clients-which-are-playing-in-the-same-Game-World, coming to the same Front-End Server (to eliminate N^2 effects).

Overall, full-scale Game-Server-to-Front-End-Server Affinity is usually an overkill, but well – I believe that there are cases when it can save your bacon[104]. Implementation-wise (assuming that additional latency of Front-End Servers is not a problem), I'd suggest to start with a software which can handle Server Groups (though not generic affinity); moreover – I'd suggest to deploy with one single Server Group. More often than not, you'll have enough time to implement affinity later (and changes to your app level to do it will be rather minimal).

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 authoritative Game World State on any of your Front-End Servers[105]

In other words, Front-End Servers should have only a replica of the original Game World State, with the original (authoritative) Game World State kept by Game Servers; it means that in case of failure of any of the Front-End Servers, it can be easily replaced without too much inconvenience for your players.

There is no need to worry too much about this restriction if you're using a generic subscriber/publisher (or state replication) paradigm, but you have to be extremely careful if

[104] though I have to admit that I didn't see such cases myself

[105] however, non-authoritative copies of Game World State – such as those one used by concentrators - are perfectly fine

you're introducing any custom logic to your Front-End Servers, because you may lose the all-important “easily replaceable” property above.

Front-End Servers: (Re)Actor-fest Implementation

Front-End Servers are essentially a part of our infrastructure (and not a part of our Game Logic), so even if you’re using (Re)Actor-fest architecture, they may be implemented in various ways without violating minimal (Re)Actor-fest requirements. In other words – as with all the Infrastructure Code, it doesn’t matter *too much* how you implement Front-End Servers – via (Re)Actors or using multi-threaded stuff with thread sync. That being said, personally I am still leaning towards pure (Re)Actor-based implementations, and we’ll discuss one such implementation right below.

One implementation of the Front-End Server implemented under pure (Re)Actor-fest architecture (see Vol. II’s chapter on (Re)Actors for details on (Re)Actor-fest, (Re)Actors, and queues) is shown on Fig 9.15:

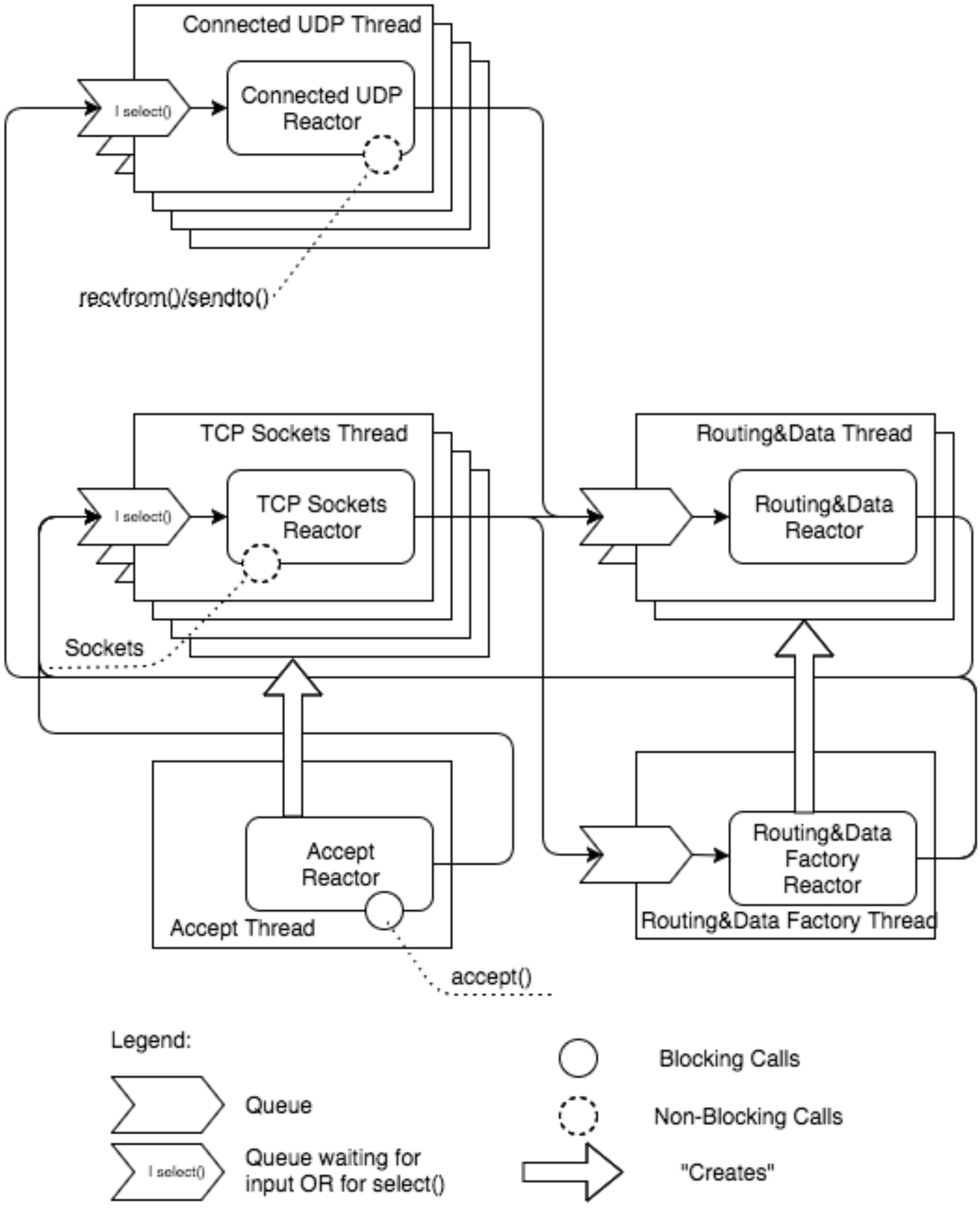


Fig VII.10

[[TODO/fig: VII.10 -> 9.15, rename all Reactor->(Re)Actor]]

Here, we have TCP- and UDP-related threads similar to those described in “Implementing Game Servers under (Re)Actor-fest architecture” section above with regards to Game Servers, and one or more of Routing&Data Threads (with each containing at least one Routing&Data (Re)Actors), which are responsible for routing of all the packets, and for caching the data (such as Game World State). Let's discuss these routing-related (Re)Actors in a bit more detail.

Routing&Data (Re)Actors

Each of Routing&Data (Re)Actors has its own data that it handles (and updates if applicable). For example, one such Routing&Data (Re)Actor may contain a state of one Game World (or several/all Game Worlds). Other Routing&Data (Re)Actors may handle routing of the point-to-point packets from players to (and from) one specific Game Server. In general, there will be three different types of Routing&Data (Re)Actors:

- 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, such as Interest Management discussed in Vol. I's chapter on Communications to avoid sending data to those players who shouldn't see it (so no hack of the Client can possibly do maphack or wallhack, and to save on game traffic too) – specific Game World handlers become a necessity.

It is possible (and often advisable) to have more than one Routing&Data (Re)Actor 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 (Re)Actors into specific threads – depends on your game significantly, but usually generic connection handlers are extremely fast and often 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 (Re)Actors per thread – the more latency and the less thread-related overhead we'll have; unfortunately, the rest depends too much on specifics of your game to discuss it here.

It is possible (and often advisable) to have more than one Routing&Data (Re)Actor within a single Routing&Data Thread

Routing&Data Factories

On the diagram on Fig 9.15, Routing&Data Factory Thread is responsible for creating Routing&Data Threads (and Routing&Data (Re)Actors), according to requests coming from TCP/UDP threads. A typical life cycle of Routing&Data (Re)Actor may look as follows:

- One of TCP/UDP (Re)Actors needs to route some message (or to provide synchronization to some Game World State), and realizes that it has no data about Routing&Data (Re)Actor, where it needs to route the message to, in its own cache.
- TCP/UDP (Re)Actor sends a request to Routing&Data Factory (Re)Actor

- Routing&Data Factory (Re)Actor creates Routing&Data Thread (with an appropriate Routing&Data (Re)Actor)
    - As any other thread-hosting-(Re)Actors, Routing&Data thread will have its own queue
- Routing&Data Factory (Re)Actor reports ID (pointer, etc.) of the Routing&Data thread's queue, back to the requesting TCP/UDP Thread
- TCP/UDP (Re)Actor (the one mentioned above) recived ID/pointer of the queue – and sends the message to the queue
    - In addition, it may cache ID/pointer of the queue, so in the future it knows where to send such messages
- Whenever the Routing&Data (Re)Actor is no longer necessary for its purposes, TCP/UDP (Re)Actor reports it to the Factory (Re)Actor
- if it was the last TCP/UDP (Re)Actor which needs this Routing&Data (Re)Actor, Routing&Data Factory (Re)Actor may instruct appropriate Routing&Data Thread to destroy the Routing&Data (Re)Actor

Phew. This looks rather cumbersome – but in fact, it is fairly easy to implement (and more importantly, debug). And as soon as such implementation is in place – it will work extremely well performance-wise and scalability-wise (as it is pure (Re)Actor-based system – it is inherently Shared-Nothing one, and such systems tend to scale extremely well).

Routing&Data (Re)Actors in Game Servers and Clients

I have to confess that personally I am positively in love these Routing&Data (Re)Actors. I love them so much that I am usually arguing for having them 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 (Re)Actor stuff on your Game Servers and/or Clients.

Front-End Servers Summary

More often than not, Front-End Servers are a Good Thing™

To summarize the section on Front-End Servers:

- For quite a few games, Front-End Servers are a Good Thing™. In particular:
    - they take the load off your Game Servers
        - which in turn *may* improve overall system reliability (as Front-End Servers are easily replaceable)
    - they allow to handle 100'000+ observers for your Big Event easily (actually, while I didn't see more that 200K of observers at the same time, at least in theory the sky is the limit); this also applies to easy handling of any broadcasted information (such as list of available games which is necessary if your matchmaking algorithm allows manual selection).

    - they facilitate single Client connection (which is generally a good thing to have, see Vol. IV's chapter on Network Programming for further discussion) – or at least limits the number of connections Client needs to keep
    - they facilitate Random Client-Side Load Balancing, and as it was discussed in Chapter 8, I prefer this kind of balancing to any other Load Balancing of the Clients
    - their drawbacks are pretty much limited to the additional latency and additional complexity
        - additional latency can be brought to sub-millisecond range
- Front-End Servers can be implemented under (Re)Actor-fest architecture, as described above; this is my personal preference most of the time, though YMMV.

Regional Datacenters to Reduce Latencies

[[TODO: pullquote capt. Obvious]]As we've already mentioned in Vol. I's chapter on GDD (and what immediately follows from geography and speed of light), for latency-critical games it is often beneficial to have some of servers closer to your players; in quite a few cases, it allows to reduce latencies. This, in turn, leads to having different locations for game's server boxes (for example, one location on each of US coasts, one for Western Europe, and so on); these locations are often referred to as "Servers", but as the term "Server" is already badly overloaded (and as each of these locations tends to have LOTS of server boxes) – we'll call them "Regional Datacenters".

There are several different architectures to support Regional Datacenters (which we'll start discussing in a jiffy); however – first let's mention a few things to be kept in mind about multi-Datacenter deployments regardless of exact schema used:

- Most of the time, even if your game is rather latency-sensitive, only a few Datacenters are maintained per continent.
    - For example, if you have "East Coast" Datacenter in NY and "West Coast" one in SF, you should be able to limit "round-trip times" of all your US players to 50ms or so;[106] for detailed discussion on RTT, see Vol. I's chapter on Communications.
    - Going into more fine-grained locations for your Datacenters, while possible, is rarely worth the trouble (except, maybe, for first-person shooters, which are by far the most latency-critical games out there).
- you MUST have very good connectivity between your Datacenters.
    - Having at least two *independent* connections for each Datacenter-to-Datacenter connection is highly advised.
        - At the very least, you should have two different inter-ISP connections (with different routing(!))
            - Having peering for these specific inter-ISP connections explicitly set by both of your ISPs (to each other) to ensure the best data flow for this critical path, is *highly* advisable too

[106] not accounting for "player's last mile"

    - Alternatives (actually, *significantly better* alternatives) to one of those independent over-the-Internet connections, include such things as end-to-end inter-datacenter non-routable links (such as Frame Relay or MPLS); without routing, they are *much* less likely to fail (and have observably smaller latencies too).
        - On the negative side – such point-to-point connections are outrageously expensive; on the other hand – at dozens of thousands of dollars per month for a 100Mbit inter-Datacenter trans-atlantic point-to-point link[107] - for certain games they're not *that much* outrageously expensive. As a result, for, say, stock exchanges it MIGHT be not too expensive compared to the risks of the whole thing going down.
        - If you're planning to use such a thing – keep in mind that non-routable links may still fail (though *much* less frequently than Internet connections), so make sure to use at least your regular Internet connection as a backup.
        - When estimating costs of such solution, keep in mind that traffic over Inter-Datacenter links can be an order (or even two) of magnitude lower than traffic-which-goes-to-the-Clients; this happens due to Front-End Servers acting as "concentrators"
- If migrating from single-Datacenter deployment to a multi-Datacenter deployment, keep in mind that Server-2-Server interactions between Datacenters tend to be VERY different from intra-Datacenter ones:
    - While for intra-Datacenter communications you can get away with relying on inter-Server connectivity to be non-disruptible,[108] for intra-Datacenter communications it won't fly <sad-face />.
        - It means that you DO need to have a policy (and a code which implements this policy) on "what we're doing if inter-Datacenter connectivity is down". As a rule of thumb, inter-Datacenter connectivity won't go down for more than 5 minutes, but transient failures 1.5-2 minutes long (corresponding to average BGP convergence time) will happen on a regular basis (very roughly - a dozen times per year, give or take an order of magnitude). Also for inter-Datacenter communications we cannot rule out rogue packets disrupting our inter-Datacenter TCP connections, etc.
    - Security is a real issue for inter-Datacenter communications. This means that you need *both* to encrypt your data[109], *and* to think about protecting your inter-Datacenter communications from DDoS attacks. More on it in Vol. IX, chapter on Security Take 2.

[107] BTW, the price didn't change too much over last 10 years, so don't expect any sharp price drops soon

[108] Well, strictly speaking, intra-Datacenter connectivity failures do happen, but with a good provider (and reasonably good hardware) they're so few and far between, that not accounting for them isn't likely to lower your overall MTBF in a significant manner

[109] at least whatever-data-which-goes-over-the-Internet

Naïve Approach – Completely Separate Datacenter DBs

Now, let's start discussing those different architectures supporting Regional Datacenters.

In a simplistic case – you can run a copy of your system (such as the one shown on Fig 9.5 or on Fig 9.7) in each of your Datacenters. This approach is frequently used in practice, though as discussed in the *Don't Do It: Naïve Game Deployment Architecture* section above, I am against such naïve approaches even in case of multiple Datacenters; in general - am arguing for at least player DB to be centralized and/or shared between different Datacenters, as discussed below (otherwise – the whole thing will become a nightmare at least for your support/monetization/security teams). While for multiple Datacenters having completely-separate DBs is not *that* absolutely-fatal as for using them merely to scale your system[110] - I am still arguing against completely unsynchronized player DBs.

Semi-Naïve Approach – Mergeable Datacenter DBs with Player Migration

While I am arguing against naïve approach mentioned above – I have to mention that it *can* be brought to the usable shape fairly easily. With separate Regional Datacenter having their separate DBs – the main problem is that *there is no single player identifier across DBs*.

Technically, this problem is easily solvable: we can simply say that global player ID is a tuple of (Datacenter_ID, player_ID_within_Datacenter), that's it. This approach (which will need to be repeated for all the other DB tables) will already allow us to merge player DBs into one single reporting DB for reporting purposes.

However, there is still a remaining caveat. We DO want to allow the same player to play on different Regional Datacenters – or at the very least to MOVE the player between them (otherwise – to bypass this restriction, players will create separate accounts etc. which will cause significant trouble in the long run).

To implement it – we'll need to keep global player ID *explicitly* in all the DBs. In other words – we cannot just *imply* that Datacenter_ID is present in all the records of the Datacenter DB (adding it only when we're merging data within the replica, but need to *store* it at least for PLAYERS table. In this case Datacenter_ID in (Datacenter_ID, player_ID_within_Datacenter) effectively becomes an ID-of-the-Datacenter-where-player-was-created. BTW, most often, this requirement to store Datacenter_ID *explicitly* applies (almost-)exclusively to PLAYERS (as vast majority of other entities don't need to be migrated from one Datacenter to another one).

[110] in particular, because player's ability to play on the same server is *naturally* limited by their location – so you can explain your player "why he cannot play with his Facebook friend X"

And as soon as we have global player ID *explicitly* in all DBs – we can migrate players between DBs rather easily. For example, we can take player with all her belongings – and transfer her using Inter-DB Async Transfer protocol[111] to another Datacenter.

Such an architecture is shown on Fig 9.16:

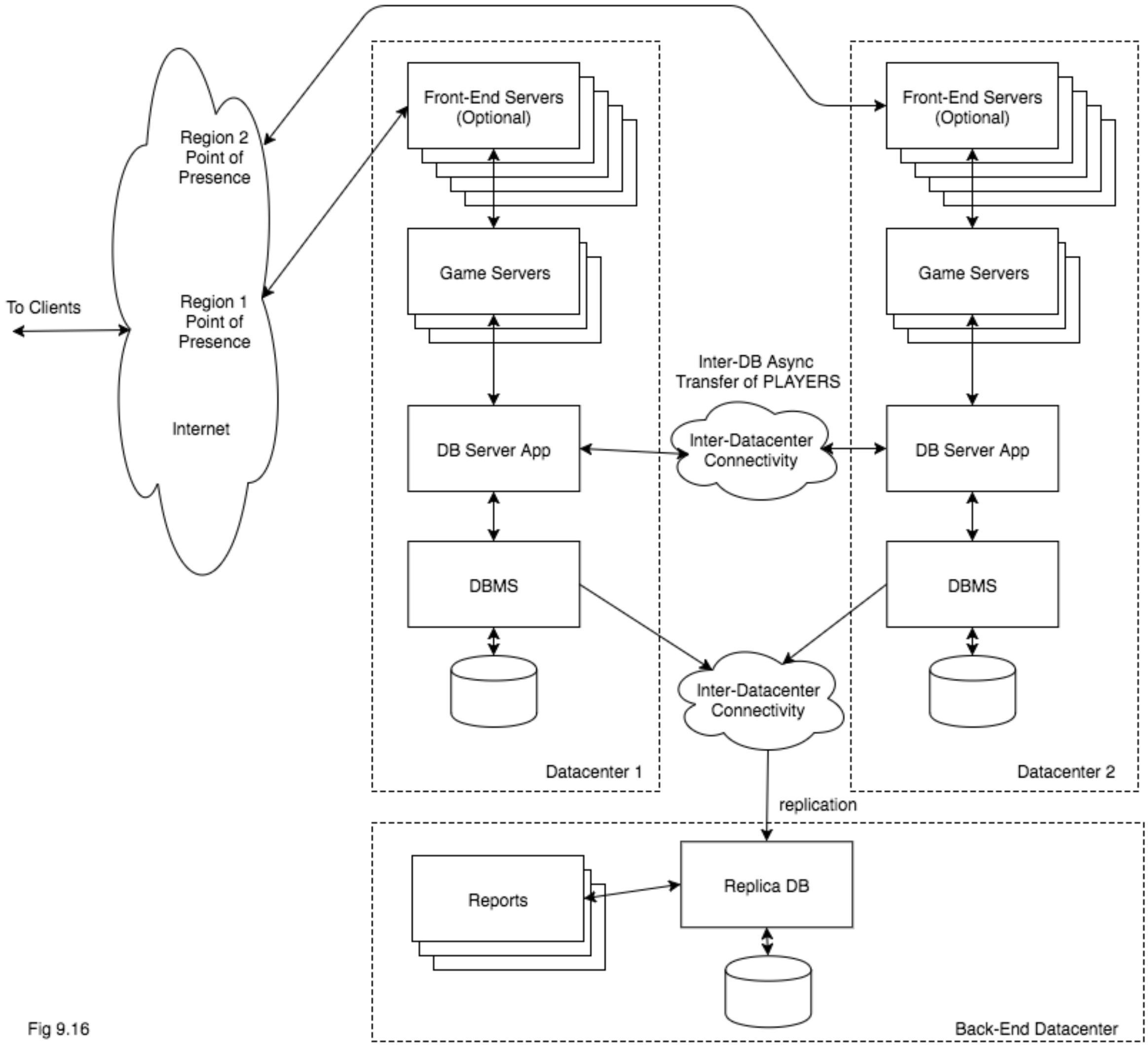

To Clients
Region 2 Point of Presence
Region 1 Point of Presence
Internet
Front-End Servers (Optional)
Game Servers
DB Server App
DBMS
Datacenter 1
Inter-DB Async Transfer of PLAYERS
Inter-Datacenter Connectivity
Inter-Datacenter Connectivity
Front-End Servers (Optional)
Game Servers
DB Server App
DBMS
Datacenter 2
replication
Reports
Replica DB
Back-End Datacenter

Fig 9.16

Notes:

- Front-End Servers are optional
- for the time being, we don't discuss exact mechanisms of implementing replicas (it can be at DB level or at DB Server App level); for more discussion on replication – please see Vol. VI's chapter on Databases.
    - what matters is that replicas are (a) asynchronous and (b) are guaranteed to be conflict-free. In other words – with ID spaces for different DBs guaranteed to be different, there is no chance of the same record to be modified by different datacenters (though keep in mind that due to player transfers also being asynchronous, there might be an occasional and temporary *perception* of the conflict on the receiving side of the replication, so you need to make sure that such occurrences don't break your replication).

[111] described in Vol. I's chapter on Communications

    - BTW, as we are guaranteed to be conflict-free – we do NOT really need "merge replication"[112].

Overall, such semi-naïve architectures are known to work in practice – though still cause their fair share of troubles. In particular, *if* your game relies on the visible-player-ID to identify player in a unique way – then guaranteeing absence of collisions between visible-player-IDs becomes impossible under this model.[113]

Datacenter DBs+Centralized DB of Player IDs

Going one step further into direction of the centralizing *at least some* data – we get to the idea of the centralized DB of Player IDs (and *only* IDs). In this case:

- We have a centralized DB which lists all the player IDs (and probably their visible_IDs, but pretty much nothing else).
- Whenever player tries to create a *new* account (or to change their ID or visible_ID) – Regional Datacenter sends a request to the centralized-DB-of-player-IDs. By doing so, our centralized PlayerID-DB can *guarantee* that our player IDs are globally unique
- Except for the creating new PlayerID or changing an old one – our Datacenters are still perfectly autonomous.

An example of such an architecture is shown in Fig 9.17:

[112] we may still use it as an underlying mechanism, but this is a completely different story

[113] technically, there is always an option to say that player-outside-of-his-home-server is always prefixed with ID of datacenter (i.e. if my account *NoBugs* belongs to USW Datacenter, then when playing at this Datacenter, I am shown as *NoBugs*, but whenever I am playing on another Datacenter – I am shown as *USW.NoBugs*), but to do it – first, we need to know that players won't object to this approach too much, and second – we need to prevent dots in player visible-IDs (or otherwise prevent collisions between names from different Datacenters) from the very beginning.

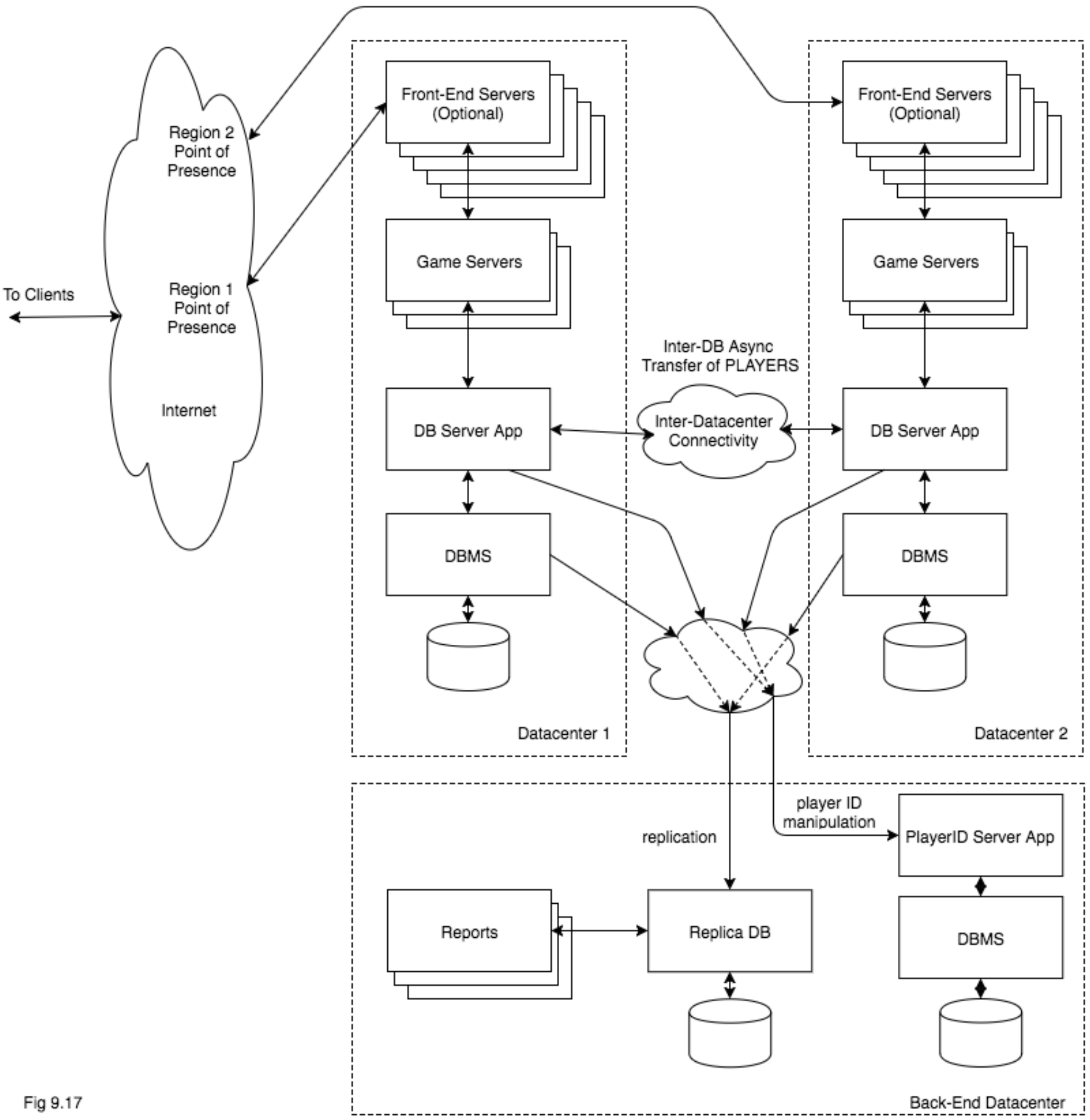


Fig 9.17

The beauty of this approach is that while it does guarantee PlayerIDs to be perfectly unique – it doesn't introduce *too much* inter-Datacenter dependencies. In particular – even if inter-Datacenter link is temporarily down, players are still able to play. Actually, the only two things players cannot do while the link is down – is to register new account, and to change ID of an existing one – and neither of these operations is *too* time-critical (if during 2-minute link downtime, 200 of your new players weren't able to register – it does qualify as an issue; however, comparing to making-your-200'000-existing-players disconnect in the middle of their match – it is pretty much nothing).

Datacenter DBs+Centralized Player DB

Going a bit further in direction of data centralization – we arrive at the model where player DB is centralized, but players are *temporarily moved* to the Regional Datacenters. Such an architecture is shown in Fig 9.18:

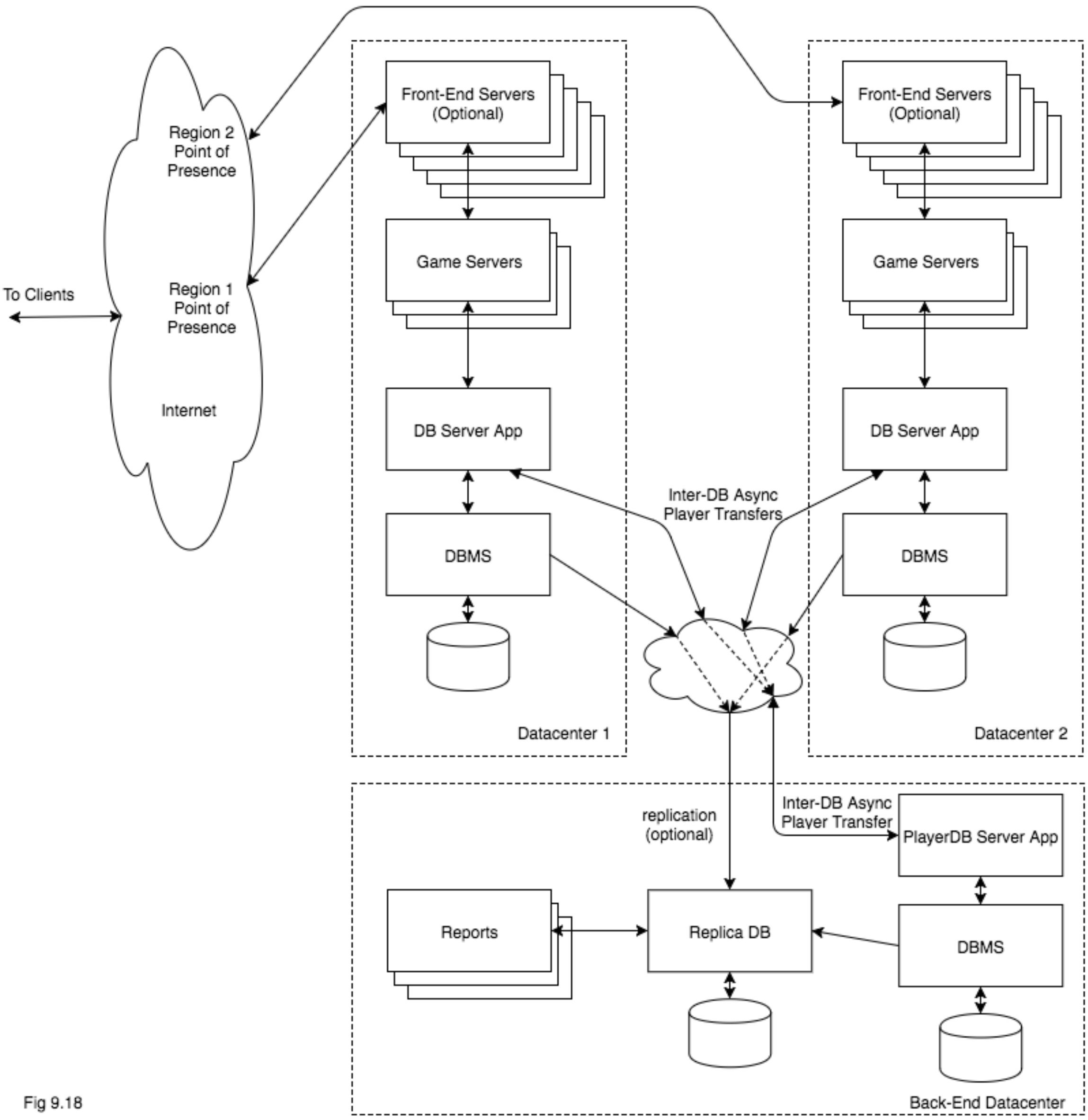


Fig 9.18

While on the first glance, it visually looks very similar to the Fig 9.17, there are several important differences:

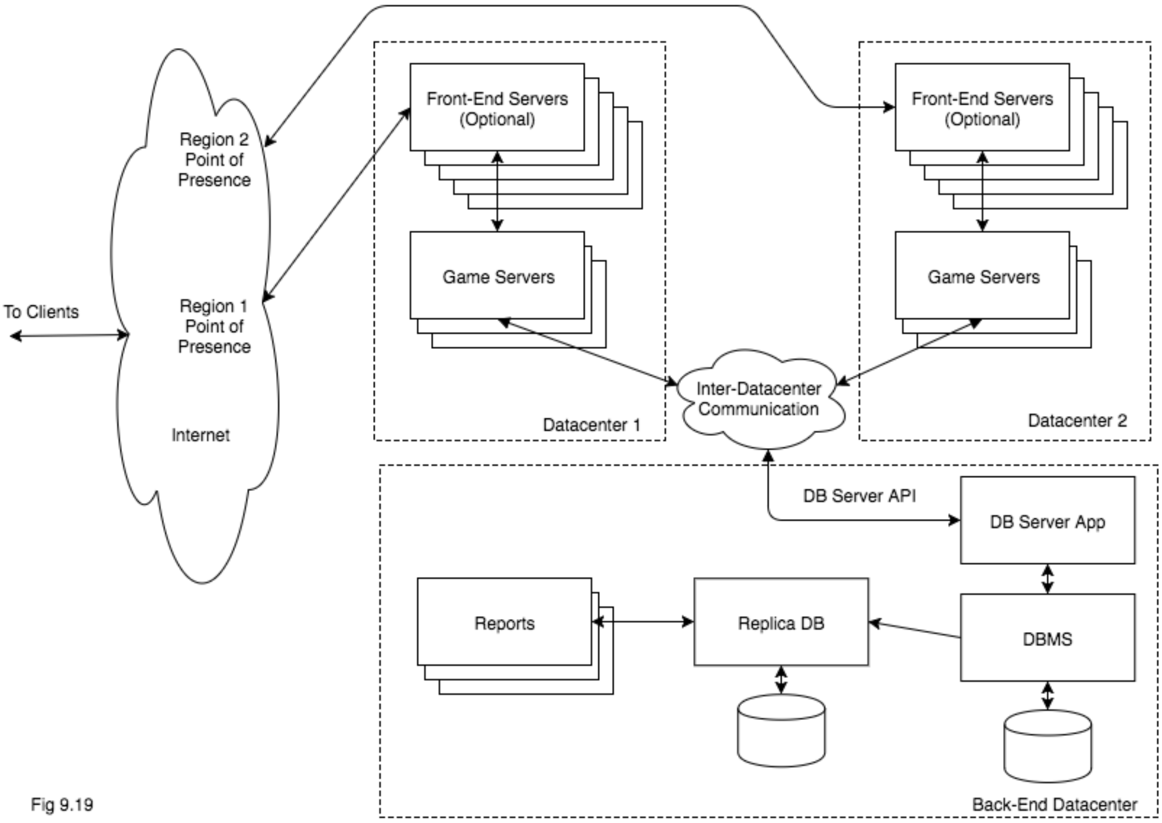
- PlayerDB is no longer merely a supplementary thing to avoid ID collisions, it is a real DB with real (and critical) data in it.
    - As a result – we DO need to include this data into Replica DB.
    - As an interesting side effect – as we're moving more and more data from Datacenter DBs to centralized Player DB, replication from Datacenter DBs will become optional
- Interactions with PlayerDB are no longer mere request-response, they're full-scale Inter-DB Asynchronous Transfers.
    - As a side effect – usually, ALL the Inter-DB Async Transfers will be between central PlayerDB and Datacenter DB (and not between Datacenter DBs directly)

As far as I know, this model is not too popular in the wild – but it does have significant (IMO *very* significant) merits. In particular, architecture on Fig 9.18 is very-well-suited for scenarios when players can easily switch between different Datacenters. Also – it can be made better resilient to Datacenter DB failures; in case of such a failure – a rollback to last-known-player-state to the state-within-central-DB can be made (and while such a rollback is certainly not a picnic, it is still orders of magnitude better than losing the whole Datacenter DB completely). On the negative side – this model has a bit more tightly coupling that previously-discused PlayerID Database; on the other hand (and most importantly) - we can still be sure that even while inter-Datacenter connectivity is down, players can still play <phew />.

Implementation-wise, this model is very similar to the Shared-Nothing model I'm arguing for to ensure database scalability (which will be discussed in Vol. VI's chapter on Databases) – so you may need most of the relevant machinery anyway (!).

Only Game Servers in Datacenters (No Datacenter DBs)

Going even further towards centralization of our database (a move which is going to be appreciated by your DBAs for sure[114]) – we can use a model shown on Fig 9.19:



To Clients
Region 2 Point of Presence
Region 1 Point of Presence
Internet
Front-End Servers (Optional)
Game Servers
Datacenter 1
Front-End Servers (Optional)
Game Servers
Datacenter 2
Inter-Datacenter Communication
DB Server API
DB Server App
DBMS
Replica DB
Reports
Back-End Datacenter

Fig 9.19

Here, we're keeping the most time-critical stuff (which is usually your Game World Servers) close to the player – while keeping all the databases completely centralized.

On the positive side – we can easily see that Fig 9.19 is *much* simpler than Fig 9.18 (the one with both Datacenter DBs and centralized DB).

[114] Although as for the rest of your teams – it is not so clear, see further discussion

On the negative side, such infrastructure is IMO *too*-tightly-coupled for its own good. In particular – unless we're *extremely* careful, this model may exhibit problems (such as slowing gameplay down or even completely stopping it) whenever the inter-Datacenter connectivity is broken – and in practice, such failures have been seen to happen on rather regular basis (once per several months) for inter-Datacenter links <sad-face />.

Front-End Servers as a kinda-CDN

In our last model (the one on Fig 9.19, without per-Datacenter DBs) we moved pretty far towards centralization; however, there is still room to make the system even more centralized. Specifically – we can use our Front-End Servers as a kinda-CDN, while keeping all the processing (and Game Servers) to one central location, as shown in Fig 9.20:

CDN

https://en.wikipedia.org/wiki/Content_delivery_network

A content delivery network or content distribution network (CDN) is a globally distributed network of proxy servers deployed in multiple data centers

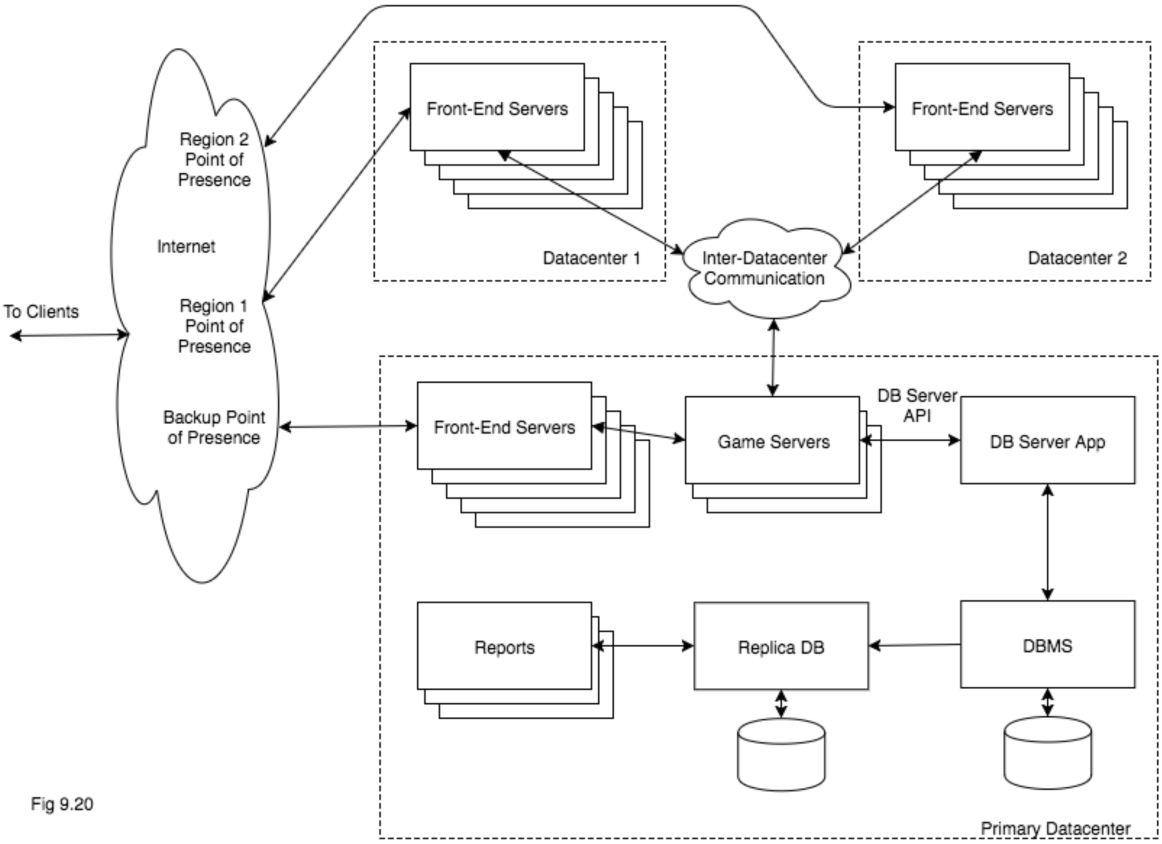


Fig 9.20

The idea here is pretty much like the one behind classical CDN: to use Primary Datacenter to process all the data – and then to add Secondary Datacenters (Datacenter 1 and Datacenter 2 on Fig 9.20) to reduce latencies for end-users. While this approach has *some* merit, 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

CDN-like arrangements of Front-End Servers MAY save some of your players a few milliseconds in latency. From my experience, it was hardly worth the trouble

When comparing our game-kinda-CDN to a classical web-oriented CDN, we'll see that as the data within classical web-oriented CDN is (almost-)static, web-oriented CDN can avoid round-trip to an authoritative server holding the (almost-)static data (by serving the data from its own cache instead). In contrast, as the game traffic is as non-static as it gets – whenever we're doing something, round-trip to the authoritative Game Server becomes pretty much inevitable[115].

Still, CDN-like arrangements of Front-End Servers similar to that of on the Fig 9.20, *may* save some of your players a few milliseconds in latency (that is, if you have a really good connection between datacenters – but you usually will get it without much additional efforts); this, in turn, may allow to level the field a bit with regards to latency. From my admittedly limited experience with such deployments, using such CDN-like Front-End-Server-based deployments was hardly worth the trouble latency-wise (as discussed above - unlike with real CDNs, 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 it still *might* make sense even given rather limited latency gains – especially for eSports, where fairness is of paramount importance.

In addition, there are other (admittedly minor) reasons to use such deployment architectures. First, they help to survive Internet connectivity loss in your primary Datacenter (provided that "Inter-Datacenter Connectivity" on Fig 9.20 survives). In practice, however, if you run your Servers from a decent multi-homed datacenter, long connectivity losses are rather rare (of the order of once-per-month or so); on the other hand - your datacenter WILL experience transient connectivity faults of around 1.5-2 minutes long (typical BGP convergence time) on regular basis, so if you're looking for excuses to use this nice diagram on Fig 9.20 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 (that is, provided that your inter-Datacenter connectivity is completely independent from your Internet connectivity so they won't fail at the same time). Second, such an approach MAY allow to survive *quite a few* of the DDoS attacks (though relying on it as the *only* DDoS protection is probably still too risky, and currently I tend to prefer BGP-based DDoS protection with a 3rd-party provider, though YMMV).

[115] well, it is theoretically possible to run Client-Side Prediction on Front-End Servers, saving round-trip at the cost of providing *approximate non-authoritative* data, but I didn't hear of any such experiments (and don't even realize why they might be necessary – especially as running the same Client-Side Prediction on the Client will provide to improve latency even further – and much cheaper for us too).

Last but not least, architectures similar to Fig 9.20, may come handy in some quite 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 – or your connectivity options are limited and traffic crosses certain Datacenters anyway, so it might make sense to put your Front-End Servers (working as concentrators) there.

Implementation-wise, the biggest problem with such CDN-like multi-datacenter Front-End Server configurations, is that as usual for multi-Datacenter deployments, we MUST account for a possibility that our secondary Datacenter goes down (in particular, in case of inter-datacenter connectivity going down) – but for the case of such CDN-like deployments, it becomes a Big Headache™ more often than not. If we cannot stop playing upon a failure of the secondary Datacenter – we have to reconnect our Clients elsewhere; 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 non-critical activities in case of such a failure might be possible depending on specifics of your game.

Regional Datacenters Summary

After going through all those options (from completely separated Datacenter DBs, via Mergeable-DBs, Centralized PlayerID-DB, and Centralized PlayerDB, to completely-centralized DBs and completely-centralized Game Servers) we can see that all these approaches represent more or less continuous spectrum going from “completely isolated Datacenters” via “Datacenters which are more-or-less-loosely-coupled to Central DB” towards “completely centralized DB and even centralized processing”.

Which of these approaches to choose – depends on specifics of your game, but in general – I would stay away from extremes on this spectrum. Specifically, completely unrelated DBs tend to be too difficult to deal with on the DB side, and too-tightly-coupled processing tends to be too sensitive to an occasional connectivity loss between your Datacenters. As a result – I feel that most of the time, the optimum solution lies somewhere between Centralized PlayerID-DB, and Centralized DB (the one without Datacenter DBs). And for quite a few games, the middle of this range (Datacenter DBs+Centralized Player DB) often becomes the best choice (and as a side bonus – the same mechanics will allow you to scale your DB to hundreds of thousands of simultaneous players, more on it in Vol. VI’s chapter on Databases).

DB Server App

For pretty much all the games out there, there is a need to store some data in the database; as a result – there should be a way to access the database. In practice, database access can be implemented using one of several very different approaches.

The most obvious way to access your database is also the worst one. While it is certainly possible to use your usual ODBC-style calls (with SQL statements right within them) to your database right from your Game Servers (i.e. *without* a separate DB Server App), do yourself a favor and skip this option. It will have several significant drawbacks which will become obvious in the medium-run, and extremely annoying in the long run. These problems range from too tight coupling between your Game Servers and your DB, to significantly worse contention due to longer locks; even worse – this approach will effectively enforce a concurrent-request processing model (without leaving a chance to go no-concurrent, which has significant benefits – more on it in the *Single-Writing-DB-Connection* section below). In short – I don't know *any* game-with-more-than-100K lines of code, where going directly to the database from your Game Servers is appropriate.

While in theory, it is possible to use your usual ODBC-style blocking calls to your database right from your Game Server (Re)Actors, do yourself a favor and skip this option.

DB Server API

A much better alternative (which I'm arguing for, and which was mentioned several times above) is to have a separate DB Server App running on your DB server box, and to have your very own message-based DB Server API (expressed in terms of messages or non-blocking RPC calls) for your Game Servers to communicate with this DB Server App.

This will keep all DB-related work where it belongs – on DB Server box, within appropriate DB Server App(s). An additional benefit of such a separation is that you shouldn't be a DB guru to write your Game Worlds, but you can easily have a DB guru (who's not a Game Logic guru) writing your DB Server App(s); the interface between the two development spaces (Game World Team and DB Team) becomes very straightforward, with a very clear separation of concerns (in particular, *all* the worries about DB integrity in all possible senses, go to the DB Team).

[[TODO: wikiquote Conway's Law]]In a sense – we can say that this separation is pretty much dictated by Conway's Law (already mentioned in Vol. II's chapter on (Re)Actors): to separate Game Logic Team and Database Team (which is certainly a Good Thing™ pretty much universally) – we have our architecture to reflect this separation-which-exists-in-organizational structure.[116]

[116] While usually, Conway's Law is used to argue to change organizational structure to follow design – it actually works both ways; what really matters is that "You may see tensions in your own organizations where your structure and software are not in alignment" [[https://www.thoughtworks.com/insights/blog/demystifying-conways-law]]; as in our case changing organizational structure (="having people who know very well *both* database *and* Game Logic") is not realistic – we should change design to avoid the tensions.

To achieve this clean separation, DB Server API exposed by DB Server's App(s), SHOULD NOT contain *any* SQL (this would defeat all the decoupling we're after). Instead, your DB Server API SHOULD be specific to your game, and (just like with Logic-to-Graphics Layer discussed in Vol. II's chapter on Client-Side Architecture) SHOULD be expressed in game terms such as "take PC Z and place her (with all her gear) into Game World #NN". All the logic to implement this request (including pre-checking that PC doesn't currently 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 Server App(s).

In addition, *all* the requests in DB Server API MUST be atomic – i.e. should be executed within one single ACID transaction; no things such as "open cursor and return it back, so I can iterate on it later" or "start transaction and do something but don't commit yet" are ever allowed in your DB Server API (neither you will really need such things, this stands in spite of whatever-your-DB-guru-may-tell-you). This doesn't mean that *implementation* of your DB Server is not allowed to use multi-statement transactions (in fact, it *will* need them pretty much for sure); however, all the requests processed by your DB Server, MUST start an ACID transaction with your DBMS, process everything within, and commit/rollback the transaction – all while processing one single incoming event/request; in other words, transactions MUST NOT be allowed to span different DB Server API events.

Having non-atomic interactions between with your Game World Servers and your DB Server tends to cause way too much trouble in real world to be acceptable. In practice, three things are the most annoying with such non-atomic interactions (and I'm not sure which one is worse, but IMO any of them is bad enough to give up non-atomics completely). First, non-atomic interactions tend to expose internals of the database to the Game Server (which is not good from clean separation point of view, and will cause you lots of trouble as your game grows). Second, non-atomic interactions are inevitably concurrent (and worse – they're *arbitrarily long* concurrent), which can easily cause lots of trouble with consistency – and dealing with these concurrency-issues-causing-bad-inconsistency-once-in-a-blue-moon requires very deep understanding of such obscure and DB-dependent things as transaction isolation levels (see also the *Concurrency Issues. Transaction Isolation, Deadlocks, etc.* section below). Last but not least, such non-atomic requests usually leave locks in DB (see the same discussion about implementation of transaction levels) – and these locks are held for long times, which tends to cause all kinds of trouble (from significantly reduced performance to *highly* increased chances of deadlocks).

With this in mind, you're going to have quite a few different types of requests coming from your Game World Servers to your DB Server App. In particular, instead of *two* requests "add money to player's account" and "remove money from player's account", there MUST be several higher-level interactions defined in your DB Server API such as "move money from one player's account to another player's account", "move money from player's account to our account", or "move money from player's account to 'outgoing payments' table". On the other hand, as soon as you have this nice and clean DB API tailored for Game World Server (or whatever-other-Game-Server) needs, you can proceed with writing your Game World Servers without worrying about exact implementation of your DB Server(s), *and* proceed with writing your DB Server(s) – without worrying about exact implementation of your

Game World Servers. Of course, there will be new requests added to the DB Server API as your game grows, but level of decoupling offered by this kind of API is the best we can hope for – and has been seen working very well in real-world games processing billions real-world DB transactions per year.

One last word of caution with regards to DB Server API. While it *is* possible to maintain transactional integrity by creating “compound” messages (with these “compound” messages containing a list of things to be done, each of these things processed by DB Server App separately but within the same ACID transaction) – I am generally *against* this approach. If we allow to have a “compound” message with separate instructions such as “add money to player’s account” and “remove money from player’s account” – then, while transactional integrity will be ensured, we will still have a problem that Game World developer will be able to break certain very-high-level integrity constraints within the database (such as “sum of all the money in the database doesn’t change”). And as soon as there are multiple-people-from-different-teams who’re responsible for enforcing such a constraint – it will be enforced by *nobody* (and will be broken *much* more frequently than we’d like it to). In contrast, with messages in DB Server API being of the kind of “move money from player’s account to our account” – *all* the responsibility for enforcing this constraint lies with DB Team (exactly where it belongs), and I’ve seen such constraints to be enforced very successfully in real-world systems – by a DB Team, that is.

Meanwhile, at the King’s Castle… Implementing DB Server App, Take 1

As soon as we have this really nice separation between Game Servers and DB Server App via our very own message-based DB Server API, in a sense, the implementation of DB Server App becomes an implementation detail. Still, let’s discuss how this not-so-small and extremely-important detail can be implemented. Within this Chapter, we’ll discuss implementation of the DB Server App very very briefly; for any further details of DB Server App implementation (as well as for all the further improvements of the DB Server App) – see Vol. VI’s chapter on Databases.

Concurrent Transactions (=”Multiple Connections”)

Historically, the most popular way of handling database access, is to have multiple connections to the database, with incoming requests processed concurrently by the database. While (as we’ll see below) this approach has significant drawbacks – let’s discuss it first.

Let’s take a look at the following Fig 9.21:

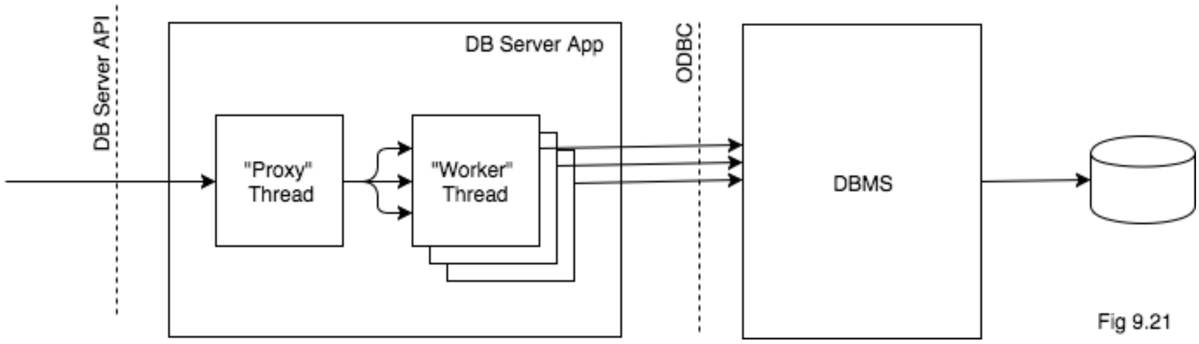


Fig 9.21

Here, we have a rather traditional multi-threaded DB Server App; it has a "Proxy" thread – which takes incoming requests, and merely passes them to one of several "Worker" threads; "worker" threads simply take these requests and process them one by one using traditional blocking ODBC/JDBC/libmysqlclient/whatever-else calls to our DBMS.

It is important that the number of "Worker" threads should be limited (i.e. independent from number of requests, and defined in relation to capabilities of the system, such as number of cores on our database box); this is necessary because having too many requests processed in parallel, will thrash the system, significantly reducing overall throughput. From this perspective – our system on Fig 9.21 works pretty much as a DIY "transaction monitor", limiting number of concurrent requests to the DB to keep the system alive. As for the number of the "Worker" threads - usually 2*number-of-cores-on-DBMS-box is not bad starting number.

If you wonder whether our Proxy Thread can become a bottleneck – well, it is extremely unlikely (except maybe for some very strange edge cases). It is perfectly feasible (and actually very far from being a rocket science) for such a Proxy Thread to process up to 100K requests/second – and having even 100K DB transactions/second qualifies as an Extremely High Number for real-world transactional databases.[117]

Overall, this concurrent multi-connection processing model is very similar to classical massively multi-connection DB processing. Sure, we effectively implement transaction monitor ourselves (instead of somebody else doing it for us), but other than that – the whole thing is pretty much the same as usual. And in return for this additional effort of implementing DB Proxy logic, we manage to keep our own DB (Re)Actor API as an extremely clean separation layer between DBMS and our Game Servers, which tends to help a Damn Lot™ even in the medium run.

Implementing Multiple-DB-Connections using (Re)Actors

[117] Some years ago, when asking about a DB with 1K transactions/second, I was told by guys from Toronto DB/2 team that it is the most loaded DB/2 instance on Windows they know about; while things have changed since – still, 100K of DB transactions per second is an extremely decent number even in 2017 (to put it into perspective – the whole Twitter has only 6K tweets/second on average).

If we want to follow (Re)Actor-fest model while implementing multi-connection DB Server App – then implementing the same multi-connection approach on top of (Re)Actors, is also very straightforward:

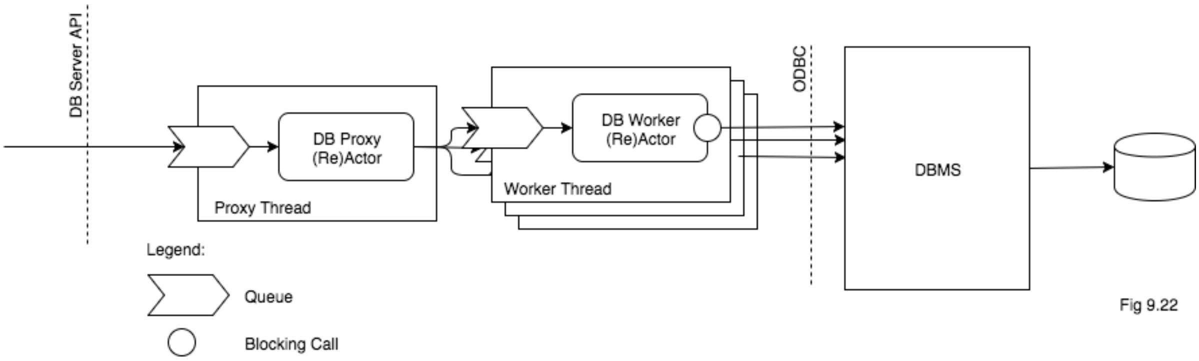


Fig 9.22

Here, compared to previous diagram on Fig 9.21, we're merely specifying how to implement those "Proxy" and "Worker" threads discussed above, nothing else. In this model (and as it stands usually with (Re)Actors)– it doesn't really matter whether our (Re)Actors run within the same process, or within different processes; the only thing which is important – is that each of these (Re)Actors gets its own thread (this is necessary because our ODBC/etc. calls to the database are usually blocking).

Multiple Connections: Pros and Cons

Now, let's discuss pros and cons of this concurrent/multi-connection approach.
The main (and extremely highly touted) upside of such concurrent processing schemas is that it is inherently somewhat-scalable; however – that's about it for the list of pros.

Downsides, however, are numerous.

Concurrency Issues. Transaction Isolation, Deadlocks, etc.

First, let's observe that as soon as we're into concurrent transactions, we MUST understand things such as locks, transaction isolation levels, and deadlocks. Way too often these issues are ignored – with really devastating results as soon as load on the system grows. In this Chapter, we won't discuss these issues in detail (for *some* discussion of these – please refer to Vol. VI's chapter on Databases) – but will rather will provide an overview of related complexities.

IMO the worst thing about using multiple DB connections, is that whatever-we're-doing,[118] those concurrent DB transactions going over these multiple DB connections cannot possibly be made deterministic, period.[119]

[118] This includes having using perfectly deterministic (Re)Actors everywhere

[119] Yes, it stands even for SERIALIZABLE isolation level

It means that our system may work perfectly under test, but will fail in production while processing exactly the same sequence of requests.[120]

Worse than that,

there is a strong tendency for concurrency-related bugs to manifest themselves only under heavy load.

As a result, you can easily live with a concurrency-related bug (for example, using *SELECT* instead of *SELECT FOR UPDATE*) quietly sitting in, and without manifesting itself. And then, when your Big Day (such as Tournament of The Year) comes - this bug crashes your site.[121] Believe me, you really don't find yourself in such a situation, it can be really (and I mean *really*) unpleasant.

In general, there *are* approaches which provide reasonably good assurances for ensuring concurrent transactions to work; one such approach goes along the lines of (a) using SELECT FOR UPDATE for *all* the data you ever read in a transaction – and (b) always following *exactly the same* pre-defined order (the one written in blood on the wall of DB department) for *all* the UPDATEs and SELECT FOR UPDATEs (this is necessary to avoid deadlocks). These approaches have their own drawbacks (in particular, deadlocks can still happen – for example, due to index-related implicit locks, so they still MUST be accounted for), but in general – they can be made work.[122]

The real problem here, however, is related to code maintenance. With dozens of changes per week typical for a gamedev environment – it is *extremely* easy to violate one of those rules; moreover – no realistic testing will be able to detect such a bug, so each such mistake will effectively create a ticking time bomb, waiting for that worst-possible-moment to manifest itself.

I will stop short of saying that it is *outright impossible* to make sure that real-world frequently-changed concurrency-based systems are bug-free; however,

The costs to ensure that concurrency-based system is bug-free while making lots of changes – are extremely high.

To start with – as noted above, such bugs are inherently untestable <multiple-ouch! />, which makes all the regression testing pretty much useless against them. In turn – it leads to tons of mundane work on code reviewing (and double-code-reviewing) – which it is not a

[120] Just because our production system got an interrupt at a bit different moment, ouch!

[121] [[TODO: wikiquote finagle's law]]To make things even worse, it goes beyond generic Finagle's Law of "Anything that can go wrong, will—at the worst possible moment." When probability of the problem depends on site load in a non-linear manner (which *is* the case for most of the concurrency bugs), chances of it manifesting itself for the first time exactly during your heavily advertised Event of the Year become huge.

[122] Another (theoretical) approach is to say that "let's use SERIALIZABLE transaction isolation level, so smart database will do everything for us" – but it doesn't really work in practice, as performance goes out of the window for SERIALIZABLE level (and you still need to follow strict order of obtaining locks to avoid deadlocks).

picnic at all. BTW, if you could write a code analyzer which can check whether your-concurrency-rules-are-complied-with automatically – it would change things drastically in this regard; however – I never heard of such a tool, and writing it would be a highly non-trivial exercise for sure.

[[wiki MVCC]]On the other hand, there are concurrent systems with millions transactions/day out there which are working, and there are DB guys&gals out there who made them work.[123] One thing which I'm not so sure about, is whether you'll have such a person on your game DB team (such people are very few and far between, *this* we can count on). BTW, as a side note: IF[124] going for multi-connection system (which I normally wouldn't) – I'd prefer MVCC-based system to a lock-based one (as MVCC tends to be more forgiving for concurrency bugs, and tends to introduce less concurrency issues[125]).

Performance Issues

From performance perspective, it has been observed that concurrent-transaction approach tends to lose *badly* to the single-writing-connection one.

Most importantly – whenever we're using multiple connections, we SHOULD NOT use app-level caching (i.e. even DB-Proxy SHOULD NOT be allowed to cache). While correct app-level caching at DB Proxy is theoretically possible even for multiple connections, re-ordering of replies on the way from DB Worker to DB-Proxy makes the whole thing way too complicated to be practical. While I've implemented such a thing myself once, and it worked without any consistency problems (that is, after several months of heavy replay-based testing and fixing subtle bugs found in the process), it was the most convoluted thing I've ever written, and I clearly don't want to repeat this exercise – especially as significantly more straightforward alternatives exist.

In addition – multiple-connection approaches tend to cause quite a bit of contention within DBMS; this includes not only about observable-from-app-level row-level locks, but also internal mutexes etc. And this contention (exactly as *any* kind of contention) can eat quite a bit of performance.

Scalability Issues

[[wiki: Amdahl's Law]]Last but not least, the scalability of the multiple connections, while apparent, is never perfect (and no, those TPC-C numbers don't prove that linear scalability is achievable for real-world transactions). The first problem here is related to the same contention issue mentioned above: as soon as we have contention over certain resource – we lose scalability, plain and simple (at least according to Amdahl's Law, but in practice scalability loss is *much* larger than that, due to context switch costs being significant).

[123] usually – starting along the lines briefly outlined above, *plus* a lot of DB-specific trickery <ouch! /> to deal with DB peculiarities and/or to speed things up

[124] and that's indeed a big fat "if"

[125] Note that MVCC DBs still do use locking for writing, and can deadlock as a result quite easily

Things become even worse if we try to scale beyond one single server box; configurations including heavy-weight solutions such federated DBs,[126] while claimed to be scalable – are observed to scale pretty badly in the real world <sad-face />.

In contrast, single-connection-made-scalable Shared-Nothing approach which we'll discuss in Vol. VI's chapter on Databases, can be extended to achieve near-perfect linear scalability; this is predicted theoretically, and also has been observed in practice at least for systems with up to several billion transactions per year.

Single-Writing-DB-Connection

In contrast to the traditional multiple-connection schema above, let's consider an alternative one – based on a *single* DB connection (at least – a single *writing* DB connection; other connections are ok as long as they're read-only *and* have Uncommitted-Read isolation level).

Sure, most of the people with a DB background[127] will throw away even the thought of having single DB connection outright. However – if asking them about their arguments against single DB connection, they won't be able to produce any, except for a non-argument "nobody is doing it this way"[128], another non-argument about "not being able to utilize all the CPU cores"[129], and a (semi-)argument about lack of scalability.

Out of these, only the (semi-)argument about scalability is worth considering. However, as it was mentioned in the *Scalability Issues* section above – scalability of the single-DB-connection-made-scalable Shared-Nothing approach is actually *better* than that of the traditional "throw everything at DB hoping it will scale" multi-connection model. In turn, it means that we're out of the killer arguments against the single-writing-DB-connection – and at least can take a closer look at it.

Apparently, this approach looks *very* simple:

[126] more generally - *any* system which is based on blocking synchronization algorithms, such as two-phased commit. BTW, as long as we do want ACID transactions over a generic federated DB – there is no way to avoid blocking <sad-face />.
[127] well, except for those who participated in my projects <wink />
[128] Not only it is a non-argument, but it is actually wrong. What stands is that "no book on databases describes single-connection-DBs", but quite a few real-world systems *are* using single-write-connection DBs – and very successfully too.
[129] We don't really care about "utilizing" cores, what we care about is ability to do the job as load increases (which is covered under Scalability)

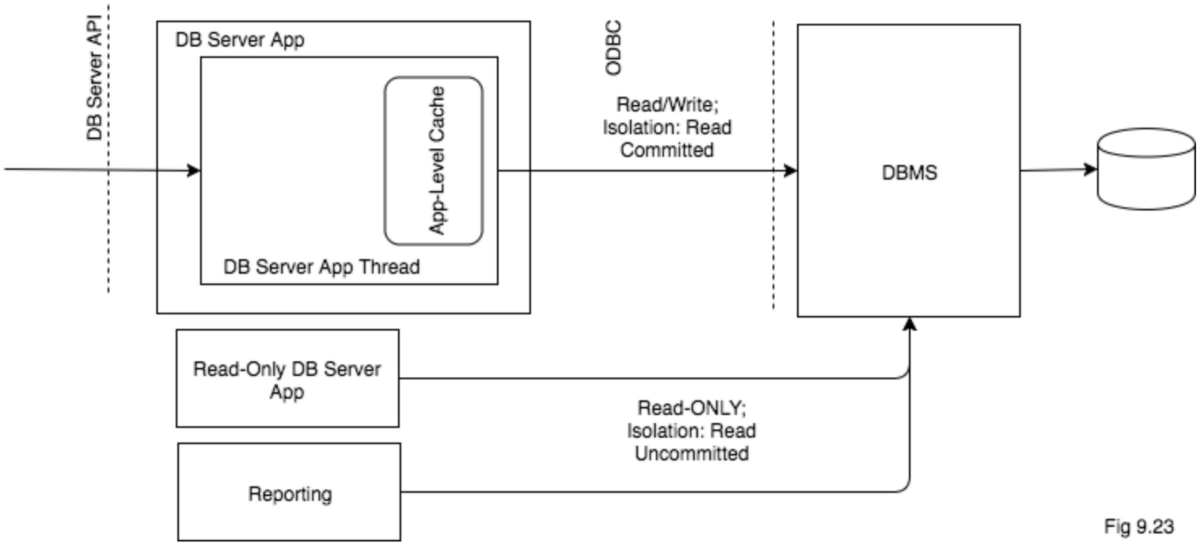


Fig 9.23

The whole point here is that the only meaningful thread of the DB Server App merely receives all the requests one by one – and processes them in a serial manner, that's it. It means that our single-connection DB Server App can *safely assume that there are no other transactions happening in parallel to its own ones*; as a result – no things such as SELECT FOR UPDATE are ever needed, there are zero worries about deadlocks (and about contention in general), and so on.

Note that, as shown on the Fig 9.23, in addition to that single-writing-DB-connection coming from DB Server App, we can also have several read-only connections. These connections MUST be read-only (to avoid interfering with the main writing connection); in addition - to reduce their impact on the writing connection (and to prevent deadlocks) – we SHOULD restrict these read-only connections to Read-Uncommitted transaction isolation level (at this level – most of DBMSs out there issue only very-short-term locks, with contention being extremely limited, and zero chance of deadlocks). Note that in spite of these precautions – at some point we'll still start to feel negative effects from the existence of the read-only connections (mostly – because of cache poisoning caused by them), so in the long run we'll need to move most of these read-only data consumers to the read-only replicas (more on replicas – in Vol. VI's chapter on Databases).

App-Level Cache

An important practical item on Fig 9.23 is App-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 after* all the necessary performance-related optimizations (such as prepared statements or even stored procedures, indexes, etc. etc.) are made on the DBMS side. Just think what is faster: simple hash-based in-memory search within your DB Server App (where you already have all the data, so we're speaking about 100 CPU clock cycles or so even if the data is not available in L3 cache), or going to DB lib -> marshalling -> going to kernel ring -> going to DBMS process over some IPC -> going from kernel ring to user ring -> unmarshaling -> finding execution plan by prepared statement handle -> executing execution plan via reading pages mostly from memory but sometimes from disk <yikes! /> -> marshaling results -> going to kernel ring -> going back to DB (Re)Actor process over RPC -> going from kernel ring to user ring -> unmarshaling results –> going from DB lib to DB App. In the latter case, we're speaking at least about a hundred microseconds, or over 1e5 CPU clock cycles, it is three orders of magnitude difference from a simple search in an app-level cache.[130]

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

Implementation-wise, the only not-so-trivial issue about this cache is ensuring its coherency but as our writing-DB-connection *is the only one* - keeping this app-level cache coherent is very straightforward.

In practice, 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 – and caching these few tables already helps a Damn Lot™ performance-wise (over 10x improvement has been observed in practice).

Implementing Single-writing-DB-Connection DB Server App as a (Re)Actor

Implementing the same thing under (Re)Actor-fest model is also obvious and extremely simple:

[130] 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

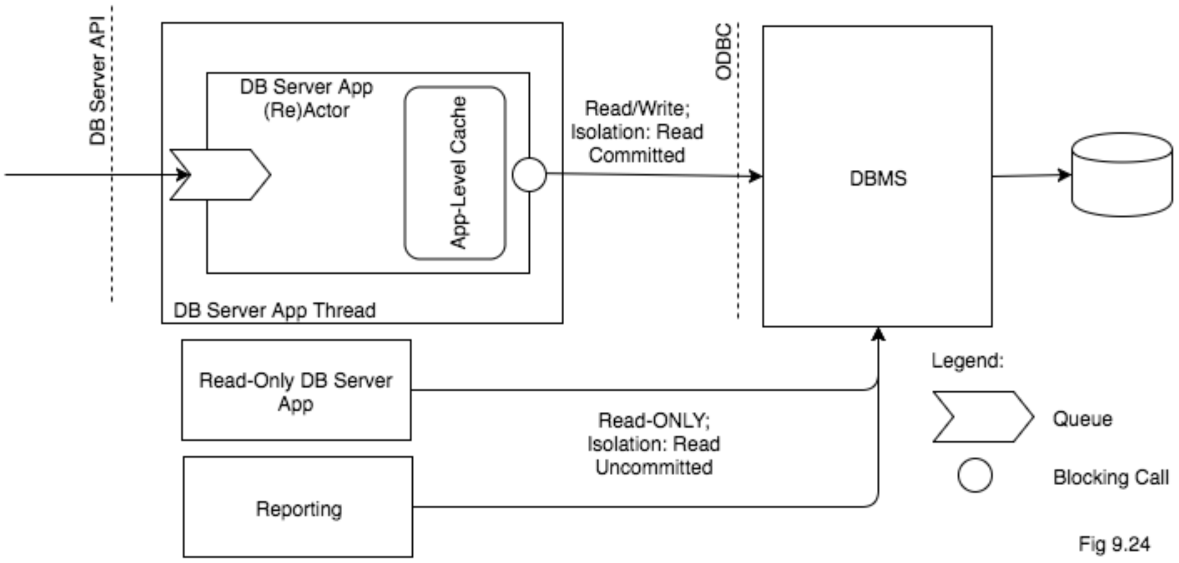


Fig 9.24

Here, there is a single DB Server App (Re)Actor which has a single connection to DBMS (such as an ODBC/JDBC/…) – and processes all the requests using blocking calls to the DBMS. That's it – it is indeed *this* easy.

Single-writing-DB-Connection: Benefits

Actually, at least at the first stages of your development and deployment, I'm usually *strongly* advocating to go with this single-connection approach[131]. It is very nice from many different points of view.

First, it is damn simple. This makes development extremely straightforward: receive request, check its validity against cache, bind appropriate prepared statement(s), execute BEGIN TRAN[132], execute bound prepared statement(s), execute COMMIT – that's it, we're done with incoming request, and are ready to process the next one.

[131] NB: while it will evolve with time – it won't become a multi-connection one in the long run; instead – it will become a set of single-write-connection-DBs.

[132] BEGIN TRANSACTION, START TRANSACTION, whatever-else-your-DB-vendor-prefers

Second, there is no need to worry about concurrency issues such as transaction isolation levels, locks and deadlocks (we'll discuss DB concurrency issues in Vol. VI's chapter on Databases).

There is no need to worry about transaction isolation levels, locks and deadlocks

Third, if following (Re)Actor-fest model - it can be written as a perfectly deterministic (Re)Actor (with all the associated goodies discussed ad nauseum in Vol. II's chapter on (Re)Actors); moreover, this determinism stands both (a) if we use "call wrapping" for the calls coming from DB (Re)Actor to DBMS (more on "call wrapping" in Vol. II), or (b) if we consider DB itself as a part of the (Re)Actor state, in the latter case no call wrapping is required to achieve determinism.

Fourth, the performance is very good. There are no locks whatsoever, and the light for the transaction is always green, so everything goes unbelievably smoothly (and without those thread context switches which tend to cause lots of waste). Add application-level caching, and we have a clear winner performance-wise![133] I've seen a real-world single-connection system which had an average transaction processing time in hundreds-of-microseconds range (that's with real-world transactions modifying multiple tables, real commit to disk after every transaction, etc. etc.).

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, we'll discuss ways to modify this single-connection approach to provide virtually unlimited scalability, [134] in Vol. VI's chapter on Databases; moreover – opposed to inherently-flawed-due-to-inevitable-contention scalability provided by traditional DBMSs, in our case it will be a near-perfect Shared-Nothing scalability.

Still, in spite of all the benefits provided by single-writing-DB-connections, this schema clearly sounds as an heresy from any-DBMS-person-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 DBMS guy to implement it <wink />). I've seen such single-connection architecture[135] handling 50M+ DB transactions per day for a real-world game, and it were real transactions,

[133] I know I sound like a commercial

[134] while in practice I've never seen systems processing above 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 this kind of numbers is even more challenging with traditional multiple-connection approach (when going beyond one single DB server box, for most types of transactional loads DB scalability isn't linear even if your DB salesperson is ready to sign it with his blood just to sell it to you).

[135] with a cache of PLAYERS table

with many creating/modifying several dozen rows, with all the strict ACID guarantees, audit tables and so on and so forth.

One thing to keep in mind for this single-connection approach, is that it is very sensitive to latencies between DB Server App and DB; we'll speak about it in more detail in Vol. VI's chapter on Databases, 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,[136] or something like NVMe, for the disk which stores DB log files (other disks don't usually matter too much). And 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, though I have no idea about real latency values they provide).

BTW, let's keep in mind that for the time being, we did NOT discuss implementation of DB Server App in detail; rather – we're making a *very high-level outline* of such an implementation, with a real discussion on the implementation postponed until Vol. VI's chapter on Databases.

DB Server App Summary

Let's summarize my feelings expressed above:

- You MUST have a very clean DB Server API
    - DB Server API MUST be expressed in game-level terms, and requests in DB Server API MUST NOT include SQL statements. While having SQL statements right within your game code may *seem* to add flexibility – in practice it just adds unnecessary coupling, and will hurt badly in the medium- and long-run <sad-face />
    - All DB requests in DB Server API MUST be atomic (i.e. DB transaction MUST NOT span different DB requests)
    - Having clean DB Server API is MUCH more important even than the question of "whether to go for single-connection or multiple-connection approach". After all – with clean DB Server API in place, it is possible to switch from single-connection model to multi-connection one (and vice versa) *without any changes to your Game Servers(!)*.
- 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 DBMS under at least million-per-day DB write-transaction load – I suggest to go for a single-connection approach
    - Whether to use (Re)Actors to implement it – is not *that* important (though personally I still prefer (Re)Actors <wink />)
    - If you're not doing insane things[137] *and* are following advice from Vol. VI's chapter on Databases – with some work (including near-optimal indexes, near-optimal physical DB layout, app-level caching, and read-only replicas for

[136] don't worry, write-back is perfectly fine as long as you have BBWC, and you still have all the integrity guarantees in the case of power loss

[137] Which admittedly can be difficult, especially if this is your first experience with DBs

reporting etc.) you should be able to process up to 10M transactions/day over one single DB writing connection.

        - At that point, you'll need to work on implementation of your DB Server App (effectively splitting it into share-nothing architecture) – more details on it in Vol. VI.

- If you do happen to have such a DB guru, try to convince her to go single-connection, but if she vehemently opposes – you MAY try multi-connection DB Server
- Note that we did NOT discuss whether your DB needs to be SQL or NoSQL – this is a subject of a separate discussion in Vol. VI's chapter on Databases. For now, it is more or less clear that you'll need to have an ACID-compliant DB, but the rest is not that obvious so we'll postpone this discussion.

MOGs and Cloud

These days, whenever we speak about the Server-Side of MOGs, everybody and their dog are crazy about clouds. However, "cloud" being a very broad umbrella term – we need to realize what is *really* meant in every specific case. Let's take a look at several *very* different things which are named "cloud" in the MOG context.

"Cloud Gaming"

Of course, with "cloud" being THE buzzword for last 10 years or so – it has inevitably lead to the emergence of the term "Cloud Gaming". Moreover, as Wikipedia says [TODO: Wikipedia.CloudGaming] – there are two different types of "Cloud Gaming": the one which is based on "video streaming", and another – on "progressive downloading".

Video/Pixel Streaming

Out of all the cloud-related stuff which happened in the recent years – one of the strangest things was a concept of the game being fully played – and rendered – on the Server-Side, where the game output was compressed, and sent back to a "thin" Client as a video stream (a.k.a. "pixel stream"). The most known real-world deployment of such a system was the one by *OnLive* (later re-launched as *CloudLift)*; the whole thing didn't really fly, at least commercially, and experienced significant technical issues too (in particular, it had serious problems working over all-but-the-very-best-connections with no-packet-loss-whatsoever).

Technical problems of video/pixel streaming games are numerous:

- Latencies are bad (ruling out faster games completely). In particular:
    - Client-Side Prediction (discussed at length in Vol. I's chapter on Communications) is completely impossible with video/pixel streaming. This means that input lag cannot be less than an RTT.

  - Video streaming is just, well, streaming – which means that Head-Of-Line blocking (discussed in Vol. IV's chapter on Network Programming) is pretty much inevitable, and in case of single packet loss – we're speaking about delays of at least 2*RTT (which need to be buffered on receiving side to avoid constant "jerking", so actually we're speaking about at least 2*RTT delays all the time – which BTW was consistent with real-world observed delays for *OnLive*).
    - To make things worse – even in presence of such buffers, multiple packet losses in a row (which, as it was noted in Vol. IV, are becoming more and more frequent over the Internet in recent years), cause considerable delays and degradations in player experience.
  - Video quality suffers significantly (to get quality comparable to Blu-ray – we'd need around 10Mbit/s of video stream – and even more CPU power to compress it in real-time)
  - CPU power required on servers to render 3D – and to compress stream with acceptable quality in real-time(!), is huge. Comparing it to traditional architectures (such as those discussed in Vol. III's chapter on Server-Side Architectures) – we're speaking about 100x increase in Server-Side CPU power(!).

Video streaming is just, well, streaming – which means that Head-Of-Line blocking (discussed in Vol. IV's chapter on Network Programming) is pretty much inevitable

As a result –

I do *not* think that games based on video streaming,[138] are viable at least in the near future. Within this book, we will *not* discuss them any further.

NB: video streaming which stays completely within LAN (such as "Steam In-Home Streaming") – with rendering happening on a Client PC and then streamed to other devices within the same LAN – does not suffer from the problems above, and can be made viable with current technologies

Progressive Downloading (a.k.a. File Streaming)

Progressive downloading (a.k.a. file streaming) is radically different from video/pixel streaming. The basic idea is to write your game in a usual Client-Server way – while avoiding to force player to do heavy downloads at once; instead – we would be downloading stuff (such as additional levels, characters, quests, etc.) as-player-plays. The key point here – is to enable "instantly playable" games. Probably, the most well-known representative of this variety of "Cloud Games" is *Kalydo/Utomik*.

[138] That is – from Client to Server, see below

Progressive downloading doesn't suffer from the problems of video streaming – and can be actually seen as an evolution of DLCs (though an automated one, and having *much* smaller pieces than traditional DLCs). As we'll see it in Vol. V's chapter on Client Updates, progressive downloading can be implemented without departing from the traditional communication and architectural patterns (well – at least without departing from them *too much)*.

Progressive downloading *can* be implemented without departing from the traditional communication and architectural patterns

IaaS vs PaaS vs SaaS

Game-specific "cloud gaming" aside – cloud is traditionally separated into different "service models"; out of these - the most popular ones are "Infrastructure as a Service" (IaaS), "Platform as a Service" (PaaS), and "Software as a Service" (SaaS).

At this point we'll be speaking mostly about so-called Infrastructure-as-a-Service (IaaS). Under IaaS – what you'll get from your cloud provider, will *look* almost-exactly[139] as a remotely-accessed hardware server box; while most of the time it will be a *virtual* server rather than a physical one (though see below on "bare-metal cloud") – it can run *exactly* the same software as can be run on physical server boxes.

Other cloud models (PaaS and SaaS) are significantly trickier. First of all, we need to realize that it is very common to create *proprietary* APIs for services which are marketed as[140] PaaS/SaaS; and *proprietary* APIs inevitably mean that (a) choosing cloud is not a deployment-time decision (i.e. you need to decide on it looong before), and (b) you're having an *Absolute Vendor Lock-In* (which, as discussed in Vol. II's chapter on DIY-vs-Reuse, is a Bad Thing™). Examples of such PaaS-with-proprietary-APIs include *Google App Engine*, and significant parts of *Microsoft Azure*; in the game engines world – we *might* want to consider services such as Photon Cloud, as a kind of PaaS (and with the Absolute Vendor Lock-In too <sad-face />).

On the other hand – sometimes vendors market services-with-*standard*-APIs (one example would be using MySQL-as-a-service) as PaaS or SaaS. Such PaaS/SaaS services with *standard* APIs can be used without introducing additional Vendor Lock-Ins <phew /> – though you still need to be very careful with two things: (a) costs,[141] and (b) latencies. Still,

[139] Usually, the only substantial difference being performance and latencies

[140] at this point, we don't care what "really" qualifies as PaaS/SaaS according to Wikipedia or any other source; what is important is to decipher wording used by various CSPs

[141] "cloud" doesn't mean it is cheap (moreover, as we'll see in Vol. VII's chapter on Preparing for Launch, more often than not, it is *exactly* the opposite) – so make sure to compare costs before jumping the cloud wagon. For example, [TODO: Frost, http://www.admin-magazine.com/CloudAge/Articles/MySQL-as-a-Service] mentions

even before going to cost and latency analysis (which we actually should postpone until later), we have to remember that

Having "standard" APIs is *very* important for services marketed as PaaS/SaaS

If PaaS/SaaS has "standard" API – we should care only about costs and latencies (and even more importantly – we can switch to in-house instance of the same service if 3rd-party service doesn't work for us). In short – we do NOT really need to consider such cloud-services-with-standard-APIs before Vol. VII (chapter on Preparing for Launch).

On the other hand – if it has *proprietary* API – the whole game becomes very different. With *proprietary* APIs, the choice of PaaS or SaaS is not a deployment-time decision anymore; as a result – if you'd like to use this kind PaaS or SaaS – you should consider them not at a deployment stage, but *right now*, *while architecting your game*. It doesn't mean that you cannot use, say, Photon Cloud – but it means that

Choosing a cloud service with a proprietary API is a decision which MUST NOT be taken lightly.

Pros and Cons of the IaaS Cloud

Now, let's briefly discuss merits of by-far-the-most-popular cloud service model – Infrastructure-As-A-Service a.k.a. IaaS.

First, about IaaS pros (when comparing IaaS to traditional rented servers):

- IaaS cloud *does* provide elasticity at cheap prices. In other words – if your load varies greatly with time – you will be able to save quite a bit by using per-hour (or even per-second) billing.
- Fast hardware replacement in case of hardware failure. Typically, for virtualized cloud server the only problem is to *detect* the failure – and then your provider will re-launch your instance elsewhere within seconds (note that you still lose all the in-RAM data of the instance – and any hard disk data too). For rented servers – fixes usually come in a few hours, so to ensure the continuity you may[142] need to have (and pay for) a stand-by server of each type.
    - Note that normally, high availability and fault tolerance are *not* included into IaaS offerings. In other words – an IaaS server is almost-exactly like your physical server, and can crash at any moment; if you want to have high

if your load varies greatly with time – you will be able to save quite a bit by using per-hour (or even per-second) billing

$11/day for a 10G MySQL-as-a-service database – which is outright exorbitant (these days, you can rent the whole server with 1T disk for about $100)

[142] (or may not – as failures beyond fans and HDDs are really *really* rare – more on it in chapter 10)

availability and/or fault tolerance – you're still on your own (but you can still use any of the methods discussed in chapter 10 – except, most likely, for those which rely on VMs).

Now, an (apparently significantly longer) list of IaaS cloud cons, as they apply to games:

- **Costs.** If you are using your system 100% of the time – cloud prices are usually *significantly higher* than renting the same computing power.[143] As of mid-2017, it was possible to rent a "workhorse" 1U/2-socket server with 2×8=16 cores, 64G RAM, and 8x2T HDDs – and residing in a very decent datacenter, for about $150/month. To rent comparable computing power (though without HDDs) from a leading-but-not-overly-expensive cloud provider, pricing for per-hour billing was $0.862/hour – or $630/month, *it is wallet-blowing four times more expensive(!)*.[144]
    - In some scenarios, however, higher per-hour pricing can be compensated by elasticity. For quite a few games out there, a price-optimal deployment will include *both* per-month rented servers (to handle fixed load), *and* per-minute rented cloud servers (to handle load spikes). For a detailed analysis of economics of the cloud and of such "hybrid" deployments – see Vol. VII's chapter on Preparing for Launch.
- **Traffic pricing.** Quite a few games out there are rather heavy traffic consumers. For example, for a "typical" simulation server capable of running 1000 players with 100kBit/s going to each of the players – for a non-cloud hosting ISP we're speaking about "unmetered 100Mbit/s connection", which (as of mid-2017) can be obtained (for a $150/month server mentioned above) for as little as $20/month.[145] However, for the cloud, a similar amount of traffic (~13 petabytes[146]) will cost you several hundreds of dollars; it is of the order of *10x price difference(!)*.
- **Higher and unpredictable latencies.** Due to the very nature of virtualized clouds, which need to move instances around, there are occasional "latency spikes" of the order of hundreds of milliseconds – and sometimes going into seconds.
    - There is a way to mitigate it – by using so-called "bare-metal cloud servers" (which are essentially merely ultra-fast-provisioned servers with per-hour billing); more on them below
- **Significantly less control over exact location of your server.** As discussed above – for fast-paced games there *are* reasons to know where your servers are – and clouds are usually pretty bad with it; the reason for this issue, once again, is that clouds are moving instances around (and can easily move the instance from one datacenter to

[143] NB: through this section, I am speaking only about *published* pricing; any kind of *special deals* which might be available from service providers (especially for high-profile games), are not included

[144] Sure, you can buy the same thing with per-month billing, but (a) you will lose all the elasticity, and (b) it will still be about 2x more expensive than dedicated servers. As for ultra-cheap cloud providers such as *Linode* or *Digital Ocean* – they bill per month and provide per-core pricing which is comparable to the cost of renting servers; however – as with any per-month billing, there are no elasticity benefits, and other considerations – such as latencies and their DDoS policies – are pretty bad for game servers.

[145] YMMV, no warranties of any kind, batteries not included.

[146] Assuming 50% average utilization

another one if they feel like it). Once again, “bare-metal cloud servers” *could* mitigate this issue.

- **Inability to customize hardware.** By design, clouds are built from commodity server boxes. While to certain extent this stands for all the hosting ISPs (i.e. for *any* hosting ISP you won’t be able to use exotic hardware – at least unless you’re doing co-location), cloud providers offer even less options to choose specific hardware, than a good hosting ISP renting out “dedicated servers”. In particular, there are at least two options which are important for certain subsystems, and obtainable on most of traditional hosting ISPs, but are not available in clouds:
    - Larger 4S/4U boxes. Historically, there are two “standard” sizes for server boxes: (a) smaller “workhorse” 2S/1U boxes, and (b) larger 4S/4U server boxes.[147] The latter ones are more expensive per-GHz – but on a positive side, they tend to have significantly longer MTBFs. This, in turn, comes handy if we want to avoid dealing with fault tolerance for a few critical server boxes – such as database server(s).[148] In short – with 4S/4U server boxes (coming from Big Three server manufacturers), they tend to have MTBFs in the range of 5-10 years, and quite often it is the most practically reliable thing to use; for more discussion – see Chapter 10.
    - For OLTP databases such as those used in games,[149] latency of disk writes is important. For hosting ISP, we can easily get a BBWC RAID card, bringing disk write latencies pretty close to the latency of PCIe transfer forward and back – and usually the number is in dozens-of-microseconds. For cloud – we’re often speaking about distributed cloud storage (with latency spikes reaching *hundreds of milliseconds* [TODO: https://www.datadoghq.com/blog/aws-ebs-latency-and-iops-the-surprising-truth/] – which is about 10’000x higher(!)). Even for low-latency on-cloud-box shared SSDs (which are going to cost you even more than those already-high numbers above) – latency-spikes due to somebody-else-doing-backup-to-the-same-SSD can easily reach milliseconds, which is already pretty bad.[150]

By design, clouds are built from commodity server boxes

MTBF

https://en.wikipedia.org/wiki/Mean_time_between_failures

Mean time between failures (MTBF) is the predicted elapsed time between inherent failures of a system during operation

[147] there is a third common size – 2S/2U, but its uses are quite specific and relatively limited

[148] While it may sound as a fallacy - as we’ll see in Chapter 10, fault tolerance itself is also often error-prone and tends to cause quite a bit of trouble – and this can include MTBF being *decreased* because of faulty fault tolerance mechanisms.

[149] especially for those which use single-write-connections which I recommend, see [[TODO]] section above and Vol. VI’s chapter on Databases for discussion.

[150] Once again, if we’re speaking about *bare-metal cloud server* with an on-server SSD – *then* it *might* become competitive.

- **Need to handle resource allocation failures.** To be profitable, cloud services need to keep the balance of the hardware-they-have – and the hardware-they-really-use-to-run-their-services. If too many cloud users request CPU at the same time – well, the cloud provider will need to decide who of the customers gets priority, and who gets offline while the load spike persists.[151] Sure, developments such as EC2's "spot instances" do help to establish clear priority rules[152] - and help to mitigate this problem, but IMO it is still imprudent to build a cloud-based system with no handling of the "CSP refused to allocate new-server-we-requested" scenario.
    - To handle such a "cloud resource allocation failure" scenario, to the best of my knowledge, two ways are possible: (a) to have a backup CSP, where your system will *automatically* go in case of primary CSP failing to allocate a new server for you; (b) to have an ability to reduce system load (like stopping lower-priority games, etc.). However, implementing (a) may be tricky for faster games (where "bare-metal cloud servers" are often necessary), and (b) strongly depends on the specifics of your game.
- **Inability to use VM-based techniques.** If your servers are virtualized – you won't be able to use virtualization techniques, including such a potentially useful thing as VM-based fault tolerance (such as *VMWare Fault Tolerance* or *Xen Remus*, more on them in Chapter 10).
    - In theory, cloud providers may include them into their offerings – but I don't know of any provider doing it yet.
    - Bare-metal clouds are exempt from this restriction.
    - Side note: if considering VM-based fault tolerance – beware of additional latencies it creates (for more discussion – see chapter 10)

Bare-Metal Clouds

One relatively recent and certainly welcome addition to the cloud scene, is so-called "bare-metal cloud" servers.

One relatively recent and certainly welcome addition to the cloud scene, is so-called "bare-metal cloud" servers. Essentially – they are just good old leased/rented servers offered by traditional hosting ISPs, but ideally – with two significant improvements:

- Ultra-fast deployment (within 2-3 minutes)
- Ability to prepare your own disk image on one of these "bare-metal servers", to store this image with the provider, and then to request new instance from this stored image
    - This way, first you prepare your own disk image of your own configuration – together with your preferred OS with its settings, with your own apps etc. etc. – and then you are able to deploy the whole thing within 2-3 minutes after you

[151] and no CSP I know provides sufficient remedies from such failures in their respective SLAs

[152] of course, for operational servers of our game, we should *not* use "spot instances"; they're important because *other* users of the cloud may use them, generating profit for CSP – while we're still enjoying prioritized access to cloud resources.

realize that you need this new server box. Very neat – and if not for exorbitant cloud pricing, I would say that this is exactly the way to go.
    - I am not sure that *all* the providers who are speaking about "bare-metal" clouds, are allowing to store your own image with them. However, as for those ISPs which don't – I have very serious doubts that they are usable for games. Bottom line – *make sure to double-check it with your CSP of choice.*

Accounting for IaaS Clouds and Bare-Metal Clouds in Architecture

As it was mentioned above – for quite a few games, having a "hybrid" deployment, with monthly-rented-servers to handle fixed load, *plus* cloud servers to handle load spikes, is an optimal schema price-wise. On the other hand, at our current stage of architectural design, designing for a specific cloud vendor is almost-universally premature; in general – all decisions about "which cloud vendor to use" should be deployment-time decisions. The reasons for it are numerous, including such things as "who knows which new cheap-and-good cloud provider will emerge by that time", "who knows what kind of deal we'll be able to strike with a specific cloud vendor", and "we might need to support TWO cloud vendors – at least to account for potential resource allocation failures".

To enable dealing with cloud server allocation (including "hybrid" deployments), while not tying ourselves to a specific cloud vendor prematurely – the following approach *might* work:

- Our Matchmaking Server could assume that we have N_{fixed} server instances which are intended for "fixed" load
- In addition, we could have a very generic non-vendor-specific *AllocateServer()/DeallocateServer()* API used by our MatchMaking Server to create new server instances above the N_{fixed} (and to destroy them when they're no longer necessary)
    - In some cases - our MatchMaking Server should also try to consolidate the load on as-few-allocated-servers-as-possible (so it can deallocate as much servers as possible, and to save us some money <smile />).
- We can implement a *simulation* implementation of *AllocateServer()/DeallocateServer()* API (with an adjustable delay between request and "creation" of such a server[153]). One way to implement this simulation is based on our own VMs (though exact implementation details are not too important).
- With all this in place - we can test the whole thing at least in a very basic way, making MatchMaking Server to "allocate" and "deallocate" new servers – while making sure that the whole system still works as intended.

Of course, when we come to deployment and will start using specific cloud vendors – both a different-implementation-of-AllocateServer()-API and subtle-adjustments-to-MatchMaking-logic will be necessary, but at least we'll have a "working skeleton" in advance, and – most importantly - will be able to support *pretty much any* IaaS cloud provider very quickly.

[153] For allocation of bare-metal servers – it can be as much as 3-5 minutes

TL;DR on Clouds

To summarize my rants about clouds (only that part which is relevant to architecture):

- At least for now - don't bother about video/pixel streaming.
- File streaming (a.k.a. progressive downloading) is not *that* different from usual game architecture; we'll discuss its basics in Vol. V's chapter on Client Updates.
- From our current perspective, PaaS/SaaS services can be divided into two wide categories:
    - Those using *proprietary* APIs. Decisions to use these services MUST NOT be taken lightly – and only if you're perfectly ok with having an Absolute Vendor Lock-In.
    - Those using *industry-standard* APIs (such as MySQL API etc.). Using such services is a deployment-time decision, so we don't need to consider them right now.
- As for IaaS services (which are the most popular cloud services out there) – they tend to cost *much* more for fixed loads, and to cost *much* less for spiked loads. As a result – for many games, price-wise the optimal way is to use "hybrid" deployments, more on them in Vol. VII's chapter on Preparing for Launch.
    - Architecture-wise, anticipating such a deployment option is usually a Good Thing™. One way to anticipate such scenarios without committing to a specific cloud vendor is briefly described above.

Server-Side. Eternal Linux-vs-Windows Debate

As we're pretty much done with the Server-Side Architecture in general <phew />, we can start discussing more specific details, such as operating system and programming language you're going to use on the Server-Side. Let's start with discussing operating systems for the Server-Side.

NB: 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 MOG 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[154]. While in theory you can run an OS X server, or can dream about trying that 16-sockets-with-32-cores-each SPARC M7 box 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 a x64 box running 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 take such a look has invariably lead to almost-religious wars.

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-platform programs. On Client-Side, being cross-platform is often a

[154] For the purposes of our discussion, we'll consider *BSD as a flavour of Linux. While this is admittedly a sacrilege - 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 for a long while.

requirement; on the Server-Side, going cross-platform admittedly can be avoided; however, even for the Server-side, I still usually prefer cross-platform code for several reasons:

- Better code quality. Programs directly depending on platform-specific stuff tend to be significantly more obscure than necessary. In addition, they tend to have LOTS of poorly understood implicit dependencies, making code maintenance MUCH more difficult than it should be.
- With cross-platform code, we don't need to go into Linux-vs-Windows debate right here, making it a deployment-time decisions rather than development-time one. 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 three-months-down-the-road 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 too much hassle
    - Once upon a time, I've even seen a large chunk of code migrated to Power CPU to get better per-core performance (it was damn latency-critical piece of code). As the code was cross-platform to start with, porting was completed in a few days (mostly testing)
    - moreover, you can have some servers on Windows and some on Linux at the same time (optimizing different audiences according to different parameters).
    - If at some point in time, you'll need to work with a 3rd-party library which happens to run only on one platform (and it does happen; in particular Windows-only DLLs for communicating with payment processors tend to be a common problem for Linux boxes) – well, you won't have much trouble running just that one server under different OS.
- 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 there is a definite correlation between the two)
- cross-platform programming helps to test your code better. If running your code on two platforms – you will get more bugs fixed; moreover, very often you see that the bug which doesn't want to manifests itself on one platform – manifests itself on another platform. One of my friends names this process of testing in different environments as "shaking the program" (Ligoum n.d.); from what I've seen, such "shaking" tends to cause quite a few bugs to fall out.
    - With deterministic (Re)Actors, in theory it SHOULD NOT be the case (they SHOULD behave exactly the same under any kind of OS). However, the very question of "whether your (Re)Actor is really deterministic" may need to be tested too, and two platforms tend to help fixing non-deterministic issues too.

How to achieve a Holy Grail of cross-platform code, is a separate story, which we'll discuss in "Going Cross-Platform" 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 (Re)Actors (and for pretty much any programming language), the best we can possibly hope for, is "run once - test everywhere", and "testing everywhere" takes time <sad-face />. Which, in turn, makes convincing-managers-going-cross-platform-route quite difficult (that is, unless you're using Java/Python/..., which are still "test everywhere" but are not *perceived* as such), so you may be forced to choose your OS even if you would like to avoid choosing it in the first place <sad-face />.

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 from proponents of BSD/Solaris).

Now, we can forget about the boring cross-platform stuff, and to concentrate on the classical-and-juicy 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

Of course, Linux is open source, and Windows is not. However, if we throw away reasons revolving around "The Greater Good of Mankind" and concentrate on our job at hand, we'll need to think about practicalities.

The (semi-)practical pro-open-source argument goes along the lines of "if you ever have a problem with Linux, you'll be able to fix it". However, you 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 pro-Linux argument, but otherwise – don't hold your breath over it; fixing kernels and drivers is *damn* complicated, and writing a driver that occasionally causes kernel panic is MUCH easier than writing a driver which doesn't <sad-face />.

I have no choice other than to brace myself and be prepared to all the punches from both Windows and Linux fans

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 – see, for example, (London Stock Exchange gets the facts and dumps Windows for Linux n.d.). My personal experience, however, doesn't support this observation (well, if we're not speaking about 9x which was indeed a disastrous disaster). In short – from what I've seen, if all you're using from Windows, is NT-derived kernel (and without any fancy COM components or .NET) – Windows has been observed work perfectly fine (more on disabling unnecessary software in Vol. VII's chapter on Preparing to Launch).

However, if you add any large Windows subsystem (such as .NET) 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 (*maybe* with a rather slim advantage for Linux).

Security

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 than Windows (that is, if you're exercising at least basic caution and are not running your web server under root account).

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 of the 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).[155] In short: it wasn't a problem of Linux as such, but a problem of Linux being misconfigured. While this specific example is not that important, it leads us to an all-important generalization:

Each server is only as secure as its admin

[155] if you don't understand why running your services under root account is a problem – wait until Vol. VII's chapter on Preparing to Launch, we'll briefly discuss it there

Which means that

It is more secure to run Windows with good admins, than running Linux with bad ones.
Also, it is more secure to run Windows if your admins are Windows ones.

On the other hand, if you have highly qualified admins for *both* Windows and Linux, then I'd certainly prefer Linux from security perspective. In particular, even as of 2017,[156] I would still say that I'd rather not rusk running a Windows box without a firewall between the server and the Internet (regardless of the games) – and for certain games, running Linux server wide-open to the Internet, is fine (with quite a few things hardened, this usually includes *SE Linux*). This (given roughly the same quality of admins) indicates quite an advantage for Linux at least on one of the very important attack vectors.

Network Processing

As we're speaking about multi-player games, everything network-related is quite of importance to us. So, the question "how efficiently the OS processes network packets", is firmly within our scope.

In particular, if your game is latency-sensitive, all chances are that you'll need to use UDP (see Vol. I's chapter on Communications, as well as Vol. IV's chapter on Network Programming for further discussion). And when you're using UDP, under heavy load you may easily run into your *recvmsg()/recvmmsg()* thread becoming a bottleneck.

In such cases, if port-per-thread scaling described in "UDP-related (Re)Actors" section above as well as multiple threads reading from the same socket, don't help - you're pretty much out of cross-platform options. On the other hand, there are a few platform-specific tricks which may help you. I tend to separate these tricks into two very broad categories: "light-weight" ones and "heavy-weight" ones. From my perspective, "light-weight" ones are those which do not require rewriting our network layer, and are merely minor adjustments (either purely configuration ones, or within the network code – but still minor ones).

Light-weight platform-specific network trickery I know about:

- Linux/*BSD: Receive Side Scaling (RSS)/Receive Packet Steering(RPS)/Receive Flow Steering(RFS). These are all about processing network IRQs on specific CPU cores (where you're ready to process the packets, so there are no cache misses etc. due to switching to a different core). They need to be configured in kernel (without the need to change your code), and have been observed to provide some performance benefits, but rather mild ones. See (Scaling in the Linux Networking Stack n.d.) for a detailed description.
- Windows: also supports Receive Side Scaling (which is actually an option provided by NIC), see (Introduction to Receive Side Scaling n.d.) for details, but doesn't seem to support RPS or RFS.

[156] and 10 to 5 years ago, the answer was even more obvious (causing "Are you guys really Crazy?" reaction when somebody was running an open-to-the-Internet Windows box)

Heavy-weight platform-specific stuff usually means using some little-known platform-specific API:

- Linux/*BSD: *netmap* (netmap - the fast packet I/O framework n.d.). When using *netmap*, performance becomes outright crazy ((netmap - the fast packet I/O framework n.d.) gives a number of ~15 million packets per second per core), *but netmap* requires *netmap*-aware drivers for NICs which are rather scarce <sad-face /> (worse than that, finding information on supported cards is difficult for *netmap*).
- Linux/*BSD: DPDK (DPDK n.d.) has ideas similar to *netmap* and provides performance comparable to *netmap* – but once again, requires explicit support for a specific NIC <sad-face />. I've made a micro-research of the most popular servers (those which dominate rentable servers in datacenters - Dell R530/R630 and HP DL120/180, more on it in Vol. VII's chapter on Preparing to Launch); as of the mid-2017, it *seems* that while both HP and Dell are officially committed to support DPDK, they do NOT seem to support DPDK for those NICs which come in these servers *by default* (and you're not likely to get anything other than default when you're renting servers – at least not without a LARGE hit in costs <sad-face />). As a result – while IMO DPDK is a *much* better bet for the Server-Side than *netmap* (because of support by manufacturers), before going into DPDK, ask your datacenter guys what exactly are the NICs they have in their rentable servers, then check that these NICs do support DPDK, and DON'T be surprised if they don't <sad-face />.
- Windows: Registered IO (RIO). RIO is quite different from the netmap/DPDK described above, and is more an interface between kernel and userland (unlike *netmap*/DPDK which are interfaces between hardware NIC and userland); as a result, it does NOT seem to require special NIC drivers. However, this abstraction doesn't come for free, and RIO seems to have significantly lower performance than *netmap*/DPDK: Microsoft claims they've got up to 4 million packets per second processed using RIO (New techniques to develop low-latency network apps n.d.) (this is for the *whole server* with an *unspecified* number of cores) – which compares poorly to *netmap*'s 15 million packets *per core*. For more information on RIO, see (New techniques to develop low-latency network apps n.d.).
- Quick summary: as always, you need to pick your own poison yourself. DPDK provides better performance – but it may require upgrading your Server-Side NICs (and installing *anything* besides standard-config-available-in-your-datacenter, is next-to-impossible unless you're co-locating, see Vol. VII's chapter on Preparing to Launch). On the other hand, RIO seems to work for *all* the NICs, but – it provides less performance improvement.
- Last but not least – before even to start considering these things, think whether you really need this much performance from your interface with NIC. With a typical game (if such a thing exists), for a network-heavy Front-End Server we're likely to be able to support around 10K simultaneous players (corresponding to processing around ~200-300K packets per second) without any of the trickery above. Using DPDK (and taking into account high single-digit microsecond app-level processing), we MIGHT be able to get into multi-million packet/second range, but unfortunately questions about using NICs with DPDK support, mentioned above, will remain unclear until the very moment of deployment <sad-face />.

    - As a result, for early stages of development, I do NOT advise to spend time on heavy-weight techniques such as DPDK or RIO. Instead, I'd rather suggest to concentrate on other things (including removal of unnecessary delays within app-level processing) for the time being, and delay any discussions about *netmap*/DPDK/RIO until post-deployment; there you will know MUCH more about your system to see whether this thing you have is a bottleneck (and if you're following my advice on having your Game Logic confined to (Re)Actors – moving from Berkeley Sockets to DPDK won't affect your Game Logic at all(!)).

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 (especially - about using different TCP congestion algorithms), 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-serving services, usually have little effect on games, where average packet size is below 100 bytes (that's including 20 bytes of IP header), and TCP throughput is not something we really care about.

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- or double-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. On the third hand <wink />, experience shows that most of the development with regards to TCP congestion is concentrated on bandwidth with much less attention to latencies, so don't hold your breath over it. Also note that as we cannot control OS on the Client-Side 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 candidate Server-Side TCP stacks in a real-world environment (the one with thousands of Clients) and see whether there are any differences. Which BTW makes yet another reason to have your code cross-platform.

If touching a subject of "Linux network stack vs *BSD", there is a long-standing perception that for network-related things, BSD stack is better; from my experience, however, the answer is that "it depends", so you'll need to try both in your specific environment (which qualifies one more reason to have cross-Linux-BSD code <wink />). That being said, chances are that for games you won't see much difference. Sure, BSD's *kqueue()* is more convenient to deal with than Linux's *epoll()* (it allows to handle more input events directly, including signals and user events without creating an artificial pipe just for this purpose – which is often necessary with *epoll()*), but for the purposes of game Server-Side it is not likely that this difference will be *too* important.

Other Technical Differences (scheduler, IPC, file access, etc)

There are quite a few debates out there related to comparisons between Linux and Windows schedulers, IPC mechanisms, etc. However, looking at these differences from a gamedev point of view - these differences are pretty much 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.[157] It means that the scheduler (almost) always doesn't really have much choice on which-thread-to-schedule – simply because *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, at least for games the difference was non-observable in practice even in the days of Linux round-robin and O(n) schedulers.[158]

One topic which is closely related to schedulers, is related to so-called NUMA scheduling. The thing here is the following. In production, you're very likely to use 2-socket (or 4-socket) x64 servers, which are NUMA for the last 10 years or so. And for NUMA,[159] 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).

NUMA

https://en.wikipedia.org/wiki/Non-uniform_memory_access

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

The topic of keeping NUMA locality when scheduling, is still very much in active development (see, for example, (Corbet)), 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)

[157] in practice, it is more complicated, as depending on the hardware, interrupt coming from NIC can be processed only on some 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 <sad-face />; see, however, discussion about RSS etc. in “Network Processing” section above

[158] 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. Please let me know if you try it though <smile />

[159] I'm speaking about classical NUMA, with a node per socket

comparison, I wasn't able to notice the difference between Windows and Linux in this regard (which admittedly might or might not be because of (Re)Actor-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, and 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 (Lameter). A bit more on practical suggestions related to manipulating NUMA-related things from application level will be mentioned in Vol. IX's chapter on Optimizations and Scaling.

When it comes to IPC (Inter-Process Communication), both systems are very similar. We'll discuss it in more detail in Vol. IX's chapter on Optimizations and Scaling, but the rule of thumb is always the same regardless of the platform: if you want it to be Really Fast™ - use shared memory for IPC, all the other mechanisms are inherently slower. And fortunately, shared memory is available on both Windows and Linux. On the other hand, if you don't care too much about achieving absolutely best speed for your IPC - all common other methods (such as pipes and sockets) are readily available on both these platforms; and while using them will cause a performance penalty compared to shared memory – it is usually within 20%, which is small enough to ignore it until you're Really Big. Fancy stuff such as completion ports and APC, may in theory provide some difference, but in practice it wasn't observed to provide any advantage at least for (Re)Actor-based architectures (on the other hand, it is obviously possible to construct an architecture which would run MUCH better using completion ports, as well <insert-any-other-specific-IPC-technology-here>). In short - IPC-wise (and if not designing for one single pre-selected IPC mechanism), you will have quite a difficulty to find significant differences between Linux and Windows.[160]

As for file systems – first, let's note that 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 (and writing logs); under these conditions all the differences between JFS, ZFS, ext4, and NTFS, won't play any significant role. And for your database servers – you need to rely on your DBMS for I/O, and to choose whatever-your-DBMS-prefers (more on it in [[TODO]] section below).

To summarize - technically (and looking *only* from games perspective) both Windows and Linux kernel are doing really good job and (drivers aside) you're quite unlikely to observe significant differences performance-wise (that is, *if* you're using methods-which-are-optimal-for-respective-OS).

C++ Compilers

If speaking about C++ (or, Kernighan forbid, C), a question of compiler becomes quite important. If you're going Windows route, your obvious choice would be MSVC[161], and for

[160] ok, local sockets tend to be a tad slower on Windows than on Linux, but if you're really after speed, you still should use shared memory, so it becomes pretty much a moot issue

[161] Starting from Visual Studio 2017, it *seems* to support Clang as one of the options to compile Windows executables – both as Clang/LLVM and Clang/C2 (C2 being a Microsoft 2nd-stage compiler); however – at the moment of this writing, I don't know of any

Linux it is probably LLVM/Clang (recently, Clang tends to outperform GCC). When comparing MSVC from Visual Studio 2015 to GCC/Clang, Linux-based compilers (especially GCC 4.8 and up, or Clang) tend to produce better-quality code, which may amount (in practice) to as much as 5-10% overall performance difference between Clang-or-GCC-on-Linux vs MSVC-on-Windows.[162]

If comparing LLVM/CLang to GCC, in practice the difference (as of mid-2017) is pretty much negligible most of the time.

Is it Enough to Decide?

All the technical arguments we discussed above are repeated ad infinitum all over the Internet, and as you probably see, I personally tend to favor Linux, but honestly, I don't really feel that these arguments are sufficient to make a decision for our game servers. In practice, the real deal is usually about the following two reasons – it is (a) license costs and (b) time to market.

Free as in "Free Beer"

Most of the time, you will need quite a few servers to run your game. Ok, let's make it "a LOT of servers". From what I know, industry standard number for simple enough simulations revolves around 100 players/core or 1000 players/2-Socket "workhorse" server (and these days, 100 players/core and 1000 players/server, give or take, is pretty much the same thing). So, if you want to run your game with 100K players simultaneously – you'll need of the order of 100 those 1U/2S "workhorse" servers. Among other things which we discuss in other places, this number means that the price of the Windows license can start to hurt in a Pretty 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 – more on it below). 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

performance benchmarks of Clang-for-Windows (and didn't do them myself either) <sad-face />.

[162] performance difference for individual functions can be much larger, but on average and taking into account such things as context switches and associated very severe cache misses, it is not that much as it may seem from "pure calculation" benchmarks

TCO wars

Of course, Microsoft is well-aware of this argument, so at some point around 10 years ago, Microsoft has pushed a point of view that despite license costs, a long-term cost of ownership (known as TCO) is lower for Windows than for Linux (mostly due to higher salaries of Linux guys). This point of view 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.

TCO

https://en.wikipedia.org/wiki/Total_cost_of_ownership

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.

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 in office environments). 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 from 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.” (Noyes)

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 your server 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 [[TODO]] subsection below.

On ISPs and Windows-vs-Linux Cost

If you by any chance think "hey, we will rent servers from ISP anyway, so license costs won't matter", you're deadly wrong. Sure, you will most likely rent servers from ISPs (see Vol. VII's chapter on Preparing to Launch for details), but ISPs (no real surprise here) need to factor in the license price into their server rental price. As of the mid-2017, kind of typical price difference between CentOS two-socket "workhorse" server and the-same-hardware server with Windows Standard,[163] 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 cost 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 (!)).

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%).

Time To Market: Familiarity to your Developers

Going besides server costs, we still have that ubiquitous question of Time-To-Market. In general, 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.

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[164]). It is not that to exploit lower cost of Linux boxes you need all of your developers to be Linux gurus (as it was mentioned above, you'll fare much better when you can keep your (Re)Actors "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 code – which will likely be server-specific anyway, plus Client-Side Prediction if applicable, which is relatively small). Second, your Game Logic code needs to be "pure" and cross-platform

[163] No "Windows Essentials" edition was observed as a rental option, probably because of license restrictions

[164] and that somebody on your team is familiar with developing Server-Side on Windows

anyway (see above). Third, 95+% of the outside of Game Logic can be made cross-platform without using Vendor-Lock-In stuff, with a relative ease. 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.

Still, having (almost-)nobody on the team with Linux experience, usually hits Linux development time badly enough to consider Windows. Still, I am arguing for writing this supposedly-Windows code as cross-platform as possible (with Game Logic being perfectly cross-platform) – then, a bit later (and when you're successful enough for Windows license costs starting to hurt), you will be able to port your infrastructure code to Linux (keeping Game Logic stuff intact). Once upon a time I've went this route myself – and it did work like a charm (though it did require extreme vigilance to remove Windows-specific stuff as soon as it accidentally appeared in the code).

It is All about Numbers <sad-face />

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.

Pretty often, under such circumstances time-to-market considerations will override lower server costs

Pretty often, under such circumstances Time-to-Market considerations will override lower server costs, so often the Linux-to-Windows question is down to 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 renting 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.

On the other hand, if your estimates show that you can handle 10K players per server (which does happen, for example, for casino multiplayer games) – 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.

DB Server Considerations

It should be noted that considerations for choosing operating system for DB Servers tend to be rather different from those for Game Servers (though these different considerations can still lead you to choosing the same OS <wink />). Here goes my personal list of special considerations for DB Servers:

- First of all, number of your DB Servers is usually relatively low, which makes all the reasoning above about license pricing, rather irrelevant for DB Servers. In other words – well, you might be able to run your DB Servers on Windows. It means that you happen to have your DB guys to be fans of MS SQL – well, you can try doing it too.[165]
- While most of DBMS do support both Windows and Linux – make sure to take a look at their preferred operating systems and distros. For example, if you're going to use Oracle DBMS, it makes perfect sense to run it on Oracle Linux (and tends to reduce fingerpointing between DBMS-vendor and OS-vendor too).
- In any case, I strongly insist on having your OS/distribution *officially supported* by your RDBMS running; while it is not a strict requirement – having officially supported RDBMS can easily reduce unplanned downtime from 0.1% to 0.01% for your DB Server (and believe me, that's a Damn Lot™ of difference).
    - This doesn't give advantage to Linux or Windows – but gives an edge to mainstream server Linux distros over not-so-mainstream ones. Usually, large commercial RDBMS tend to support distros such as RedHat and SUSE)and to less extent Ubuntu and Debian); in any case - make sure to check the list of supported distros for your DBMS of choice.
        - Even a question "whether CentOS is good enough" is not that obvious one when speaking about DB Servers. While technically CentOS == RedHat, from fingerpointing point of view it is not so: running CentOS gives your DBMS support a damn good excuse of "hey, you're running an unsupported distro" – which tends to cause more trouble than it is worth when dealing with those guys (when you're trying to make them to fix that bug in their DBMS which haunts you). As a result, I'd still prefer to have RedHat rather than CentOS – for DB Server, that is (and this reasoning does *not* apply to Game Servers)
- When it comes to file systems – interaction of your DBMS with file system depends on the way how your DBMS handles disk I/O:
    - Some of serious commercial RDBMSs prefer to work with their own containers, so any impact of file system will be pretty much eliminated; even if those DB containers are lying on top of file system (as huge fixed-size files) – they're not going to use much of file system magic anyway, making file system choice a rather moot issue.
        - A related but slightly different issue is "Concurrent I/O vs Direct I/O" (which is in turn related to a specific implementation of file system). However, even here the jury is out which of these two performs better under real-world loads. In short – if you're into this kind of stuff, you'll need to test it yourself.
    - On the other hand, quite a few DBs out there which tend to rely on file systems heavily at least by default. Still, even for them observation it is not that obvious which file system is better; as of 2017, ext4, XFS, and BtrFS are

[165] NB: personally, I happen to dislike MS SQL, but this is a different story – and I cannot be sure that this dislike (originated 15-20 years ago) is still valid now

the most common contenders – but make sure to perform your own testing on exactly that hardware you're going to use,[166] and under close-to-real-life loads before making a decision. And if you're too lazy to do it – at least use whatever-file-system your DBMS vendor recommends (while they can be wrong for your specific load, but at least you'll have a reasonably good starting point).

Mixed Bags

In some cases, you may need to run a mixed bag of Operating Systems on your deployment site. A few examples of such things include:

- DB Servers running on different OS than Game Servers (see above for a list of considerations which are different between Game Servers and DB Servers). Of course, it is better to keep both Game Servers and DB Servers on the same OS (even if the distro is different), but if it happens to be better to run them on different platforms – well, it is not the end of the world either.
- Even if you're running most of your Game Servers from Linux – there is always a chance that some weird payment provider will ask you to use their Windows-only DLL (and they may provide so good rates that your management may ignore your pleads).
    - Or it can be a 3rd-party library (say, 2FA library handling all those 2FA devices).
    - Or it can be some kind of strange stuff regulators in some European country asked from you – and believe me, this stuff can easily be weird enough to require calling Windows .DLL.
    - Or pretty much any kind of 3rd-party stuff which you need to integrate with for almighty Business Reasons <sad-face />
- On the other hand, even if you're running a mostly-Windows shop – it is still usually better to run your public e-mail/web servers under Linux
- As your game becomes more and more successful - another team (such as "team writing CSR reports") may be created – and they may prefer to use ASP.NET running on Windows
- Etc. etc. etc.

Moral of the story. While it is usually possible to start your MOG running all its servers under one OS, keep in mind that as soon as you're successful – you will probably need to go heterogeneous, at least on some fringe (as we prefer to think <wink />) servers.

Linux-vs-Windows: Time to Decide

To summarize my arguments above:

[166] this is DAMN important; in particular, testing on hardware without BBWC RAID card, and with it (or on SSD and HDD) – can provide *drastically* different results

- if you want to use Linux because you're familiar with it – you're fine regardless of number of servers you 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 cross-platform code, which will run on more familiar Windows first (to speed time-to-market), and to migrate to Linux later
    - however, to achieve it – the code needs to be cross-platform from the very beginning; porting Windows-specific stuff (the one nobody intended for porting in the first place) to Linux is a recipe for a mortgage-crisis-size disaster.
    - extreme vigilance to avoid being inadvertedly locked-in is required (see Vol. II's chapter on DIY vs Re-use for details). On the other hand, (Re)Actors tend to make dependency fighting simpler.
- if you're in doubt – use Linux, it is safer that way[167]

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 Vol. V's chapter on C++):

- DO fight 3rd-party dependencies. Unnecessary dependencies tend to make Windows less stable, less secure, the code less manageable, etc. Refer to Vol. II's chapter on DIY vs Re-use for discussion 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 for Windows. 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.

DO fight 3rd-party dependencies. In spades for Windows.

[167] "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".

- 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. However, if most of your game in not .NET-based, running .NET 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).
- As a rule of thumb, stay away from web services (that is, unless you're into Web-Based Architecture), at the very least for time-critical pieces of your system. In general, *any* technology that has a blocking RPC-like interface for inter-process (and even worse, inter-server) communication, should be avoided for several reasons. Problems with blocking calls are numerous, from causing excessive context switches (which in turn makes the whole thing poorly scalable) – to difficulties with detecting and handling timeouts and problems staying non-revealed during in-lab testing (this is very typical for blocking RPCs, as blocking RPCs are very latency- and load-sensitive, and latencies and loads for intra-lab testing are usually very small to cause any trouble <sad-face />). For more details – please see Vol. II’s chapter on (Re)Actors.
- Stay away from COM.[168] 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, which you should fight as stated above. Consider an offense of using DCOM as just a seriously aggravated form of the offense of using COM.

Things to Keep in Mind: Linux

Linux also has its 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 easily; within our game-related code, there is no reason to depend on package manager or exact directory structure. If you need these badly, move this kind of dependencies 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).
    - If considering *BSD family, they are somewhat different from Linux, but as long as you're

DON'T program for one single distribution.

[168] yes, I know lots of people consider COM long-dead; unfortunately, it is not

using POSIX APIs (and that covers 99% of what you'll really want in practice,[169] 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. Write your own daemon doing these things.

Things to Keep in Mind: (Re)Actors

In addition, there are a few things to remember about, which apply if you're using (Re)Actors (which you should <wink />), but regardless of the platform you're developing for:

- DON'T use platform-specific APIs within your (Re)Actors (see below about using them outside of (Re)Actors). Leaving aside a few narrow exceptions, your (Re)Actors need to stay "pure" (see Vol. II's chapter on (Re)Actors 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 (Re)Actors. The whole (Re)Actor-fest system can be written in a fully cross-platform manner.[170] Even if your code can use platform-specific optimizations,[171] it is still better to have a purely cross-platform version (at the very least, to have a baseline to compare your optimizations against).[172]

DO consider cross-platform code even outside (Re)Actors.

MOG Server-Side. Programming Languages

Going Cross-Platform

In the previous section we discussed choosing an Operating System for your MOG servers. And 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

[169] the remaining 1% includes things such as *epoll/kqueue*

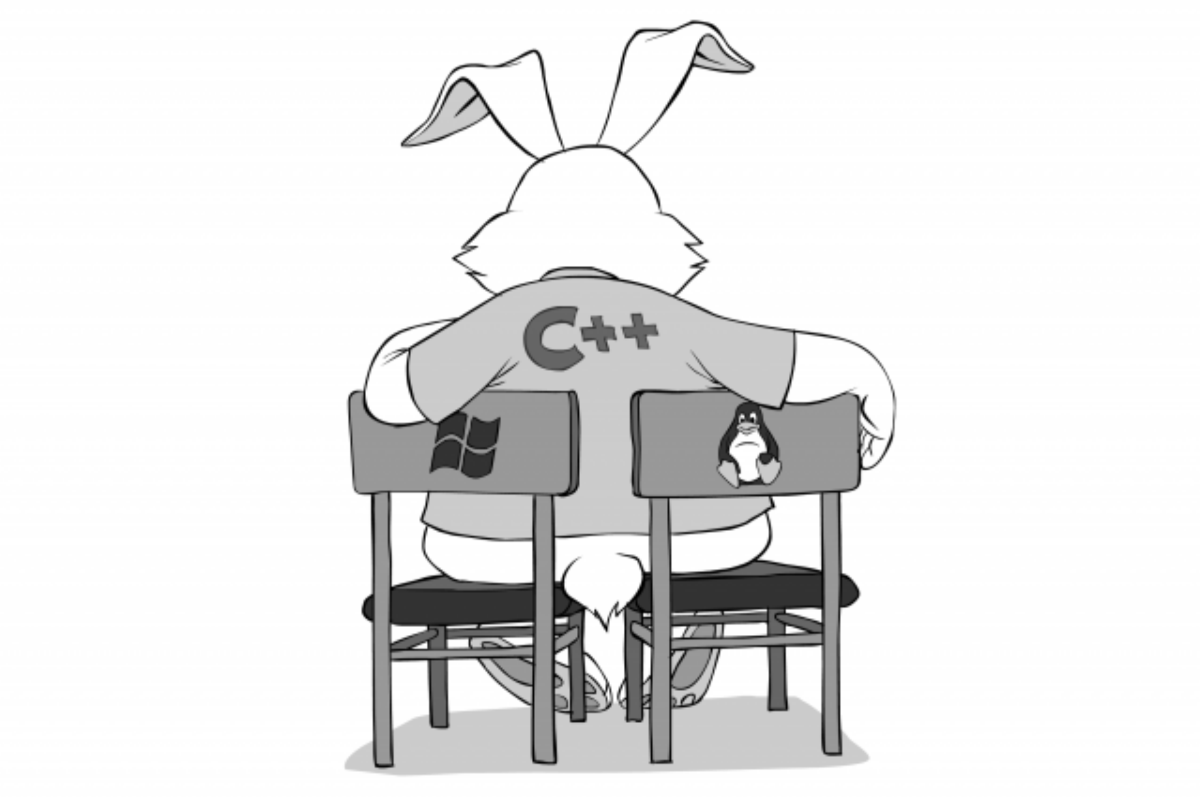
[170] been there, done that

[171] most likely – *epoll()/kqueue()*

[172] 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

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++. BTW, I am certainly *not* trying to argue that C++ is *the only* programming language for the Server-Side (in fact, for the Server-Side advantages of C++ are admittedly *significantly smaller* than for the Client-Side[173]) – but I can have a personal preference, can't I? <wink />. Also let's note that *if* you're using C++ on Client-Side (which is quite likely – see Vol. II's chapter on Client-Side Architecture for relevant discussion), *and* are going to share some code (such as simulation code used both for Server-Side simulation and Client-Side prediction) – than C++ will get an objective advantage in addition to any subjective ones <smile />.

Now, let's see whether we can make C++ cross-platform. To those having any doubts in this regard:

yes, C++ can be made cross-platform, I've done it myself on numerous occasions.

It tends to work even better when you have your code restricted to event-driven side-effect-free processing (a.k.a. deterministic (Re)Actors, see Vol. II's chapter on (Re)Actors for details); on the other hand, (Re)Actors are certainly NOT necessary to make C++ cross-platform. For our current discussion, one thing is important about (Re)Actors: as soon as

[173] down to the point that "unless you already have an unusually strong C++ team – it is usually better to use some-other-programming language on the Server-Side"

your (Re)Actor 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 very similar to “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 Vol. II’s chapter on (Re)Actors). This is especially true for C++.

Note that for some pieces of code,[174] 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 programmers out there who don't know it, especially among hardcore zealots of Windows- or Linux-specific development):

I know it is Really Old News™ for all the seasoned C++ cross-platform developers, but believe me or not, there are lots of programmers out there who don't know it

- 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 uses *only* the stuff which is supported across the board. In practice, there are several big sets of APIs which we can safely consider cross-platform:
    - C++14 standard, including *std::* library
    - Most of C Standard Library
    - *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 different semantics from POSIX *exec*()*.
- You launch it, iron out all the bugs, and then 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 Vol. II’s chapter on Client-Side Architecture 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.

[174] if you’re using (Re)Actors – usually it should be *outside* of (Re)Actors, i.e. in Infrastructure Code

- 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 – and can have platform-specific optimizations at the same time. For further discussion on C++, see Vol. V's chapter on C++.

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.*
> *– G. Orwell–*

Another way to achieve cross-platform code is to use one of the cross-platform languages, such as *Java*, *Python*, *C#*, *Go*, 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 many 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.

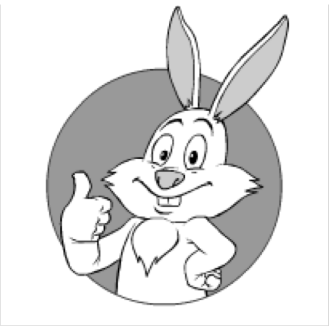
This "going platform-specific being more difficult" is actually IMO the main advantage of cross-platform languages when going cross-platform

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[175] 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 a (Re)Actor 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).
    - 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).



Almost all cross-platform programming languages I know are garbage-collected

- 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 (Re)Actor, but reading a book on C++ can easily be overwhelming (especially those books which start with discussing interesting-but-not-really-important-and-rarely-used-things such as operator overloading and multiple inheritance).
- Good C++ developers are really few and far between, not to mention they're very expensive. For most of the languages mentioned 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 some of these cons are not as well-known as pros, so I'm spending more time elaborating on them):

[175] Rust being the only exception

Almost all cross-platform programming languages I know are garbage-collected

- Almost all cross-platform programming languages I know 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 potentially be used, but won't be used, ever) ('No Bugs' Hare). 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 theoretically reachable). Of course, memory bloat for garbage-collecting languages can be managed (there is nothing difficult in explicitly assigning *null* to a reference); however, then – just as with C++ memory management - it once again becomes all about self-discipline, and whether after doing it garbage-collected languages will still provide that much development speedup over manually memory-managed C++ - is not obvious to me.
        - On the other hand, it should be noted that for (Re)Actor-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 memory management is usually not that much of a problem for (Re)Actor-based development), and fixing them usually isn't *too* difficult.

Semantic Garbage

https://en.wikipedia.org/wiki/Garbage_(computer_science)

Semantic garbage cannot be automatically collected in general, and thus cause memory leaks even in garbage-collected languages.

-
    - 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 MOBA match, lasts longer-than-usual, 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# code may exhibit very different behaviour with regards to GC 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* parameters for CLR (for detailed discussion on it, see (Bray n.d.)). Such “concurrent” garbage collectors tend to 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 relatively minor overall performance penalty) bring pauses down to single-ms range even for large heaps, which makes it "good enough" for-all-games-except-maybe-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 (=”if your game is fast, you’ll need to measure it yourself”).
    - As an 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[176]) 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 (Re)Actors (as (Re)Actors are Shared-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 (Re)Actors. However, there is a price for doing it, and the price is related to the per-VM overheads; 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 stopping being cross-platform), this is a complication one needs to be aware about in highly-loaded network-oriented environments. Also I have no idea whether the workaround would work as intended under Mono.

[176] 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

- 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 their respective JIT compilers for both Windows and Linux, 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 <sad-face />.[177]

JIT

https://en.wikipedia.org/wiki/Just-in-time_compilation

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

- Even when compared with JIT-enabled cross-platform language, C++ performance can be made at least somewhat better 99% of the time.[178] On the other hand, 95% of the time you won't bother with such optimizations even in C++, so it doesn't really matter much. Possible exceptions include heavy AI and/or heavy physics simulations (especially if they go well with SSE). Oh, and for "glue" code it doesn't matter at all (neither you should try to optimize it in the first place).

SSE

https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Streaming SIMD Extensions (SSE) is an SIMD (Single Instruction Multiple Data) instruction set extension to the x86 architecture

Personal Preferences and (Re)Actors

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 or .NET Core *might* work for you too (with some caveats: Mono is not exactly 100% compatible, and .NET Core, while officially supported by MS under Linux, has limited feature set, so you need to know from the very beginning what exactly you're targeting – full .NET or .NET Core[179]).

When it comes to Go, I tend to dislike its goroutines; I explained more on this dislike in Vol. II's chapter on (Re)Actors, but very briefly goroutines (unless configured to run in one thread) encourage programming style which requires synchronization; while it is possible to

[177] 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 (though make sure to start testing your game under that-implementation-you're-going-to-use-in-production, ASAP).

[178] 100% of the time if we allow to use *inline asm*, but believe me – you do NOT want to go there

[179] BTW, personally, being a minimalist, for any new project I'd use Core without a shade of doubt

use it in a manner similar to (Re)Actors (and avoiding all the problems) – this manner is not enforced by the go language, which makes it easy to step on a slippery slope of thread sync which will quickly lead to a mutex-ridden demise </sad-face >.

Overall, I can say that that with some self-discipline, (Re)Actors described in Vol. II's chapter on (Re)Actors (and which are pretty much the same as Erlang's actors/processes or Akka actors), can be easily implemented in any of the cross-platform programming languages (and in C/C++ too).

Scripting Languages

We went through C++ and cross-platform languages, but we're not done yet.

As it was mentioned in Vol. II's chapter on Client-Side Architecture, for game development, there is a common practice to use scripting languages to write Game Logic (with people writing in scripting languages including, but not limited to, Game Designers). Moreover, on the Server-Side (unlike Client-Side) obscurity-based protection from bot writers is not an issue (as Server-Side code is not supposed to be exposed to players); this in turn means that scripting languages become significantly more feasible for the Server-Side.

On the Server-Side (unlike Client-Side) obscurity-based protection from bot writers is not an issue (as Server-Side code is never exposed to players)

Therefore, it seems to make perfect sense to allow using some kind of scripting language on the Server-Side (regardless of you using it on the Client-Side).

Two most common scripting programming languages used in games, are *Lua* and *JavaScript*. I won't go into comparison of these two languages (they're actually *very* different), but will just note that both will do their job when it comes to game scripting. However, one thing needs to be mentioned in this regard, and it is that future of the Lua hinges on further development of *LuaJIT* – and development of *LuaJIT* doesn't look too good as of mid-2017 (development is not too active, and with previous history of conflict about support for Lua 5.3, and original author Mike Pall leaving the team – it is unclear whether it will continue well or not). This, in turn, IMNSHO counts as a *strong* argument against Lua (and in favor of JavaScript); moreover, some of former Lua fans have already switched to JS (see, for example, [TODO: https://realmensch.org/2016/05/28/goodbye-lua/]).

Other than that, the most common concern about allowing scripting on the Server-Side is related to performance. However, with *LuaJIT* (though see my concerns about *LuaJIT* above) and *V8 JavaScript* (which also has its own JIT), this is much less of a concern than for non-JIT-ted script engines.

When speaking about Server-Side JavaScript – I actually mean *Node*.js. And I have to confess that I like *Node.js* a LOT (more than any other programming language barring my first love C++ <wink />). If they throw in built-in determinism – I may even consider divorcing C++ for the Server-Side after all these years of happy marriage <smile />. On the somewhat-negative side – I feel that *Node.js* is *too much* about being non-blocking; as it was discussed in Vol. II's chapter on (Re)Actors – I am advocating *mostly*-non-blocking processing (and *may* allow to block on local disk/DB operations), opposed to 100%-non-blocking. Still, out of the ready-to-use systems for Server-Side – *Node.js* would be probably my 2nd choice (after DIY-framework-in-C++).

On Programming Languages as Such

There is one more thing which I didn't touch yet – it is a question of the differences between programming languages themselves. Here, I am going to be hard once again– this time for not going into a lengthy discussion about pros and cons of syntactic sugar used by different programming languages. 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.[180] This is also confirmed by Line-to-Line conversion exercise which will be discussed in Vol. IV's chapter on Things to Keep in Mind.

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

In other words, we have good news and bad news. Good news is that

Whatever mainstream programming language we're choosing – we can't be TOO wrong <smile />

A flip side of it is that

Whatever programming language you happen to know/love – it doesn't automatically make you "better" than the rest of developers

BTW, I know quite a few people out there who will say I'm deadly wrong about it, and that <insert-their-favorite-programming-language> is obviously so much better. Well, it is not. The whole discussion about advantages programming languages reminds me of the discussion 15 years ago or so, about "RISC vs CISC vs VLIW vs EPIC" – with LOTS of arguments flying around about performance advantages of EPIC/Itanium; however, at the end of the day, it so happened that all these things are irrelevant for performance, and it is "NUMA vs FSB" question which really matters performance-wise (with Itanium buried for good after all the efforts and billions spent there – exactly because of concentrating on EPIC but not on more down-to-earth things such as NUMA). IMO pretty much the same thing is

[180] it doesn't really stand for *Erlang*, and I am not 100% convinced that it stands for *Lua*, but *C++/C#/Java/Python/Javascript* as-you-SHOULD-use-them-for-application-level-programming are all pretty much the same, saving for relatively limited amount of oddities and peculiarities

happening with programming languages now: it is not specific syntax which matters for developers, it is more subtle things such as threading models etc. which really make a difference.

Another observation which helps in this regard, is that there is a tendency for modern mainstream programming languages to *converge* with time. For example, *C++11* code is much closer to *Python* code than C++03,[181] *Java 5+* (with generics) is much closer to C++ than Java 4- (the one without generics), and so on. In general, programming languages borrow certain constructs and practices (usually best ones, but it is not guaranteed) from each other, bringing them closer. There are lots of examples of such convergence, with the most obvious ones being RAII-like behavior[182] and lambdas[183] (where languages already converged), and coroutines (where converging is still in progress).

In practice, from my experience, the only thing which tends to be fundamentally different between the programming languages, is the difference between manual and automated memory management. Still, with more-or-less modern style of C++ code (with widespread use of containers and *std::unique_ptr<>*), the difference is actually not *that* drastic. Sure, creating reference loops from is not advisable with *std::unique_ptr<>*, but to be honest, I don't remember last time when I really needed such a loop anyway.[184]

Which Language is the Best? Or On Horses for Courses

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?

Horses for courses

https://en.wiktionary.org/wiki/horses_for_courses

An allusion to the fact that a racehorse performs best on a racecourse to which it is specifically suited

My two cents points in this regard are the following. First, there is no such thing as "the best language for everything". Rather, 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* or *C#* for the third one. Doing everything-we-have-in-such-a-huge-project-as-MOG in one single language, while possible, in many cases will be suboptimal.

[181] ok, it is probably better to say "SHOULD be written in a manner which is much closer to…"

[182] I mean try-with-resources in Java 7, *with* statement in *Python*, and *using* statement in C#

[183] In spite of all the peculiarities such as lambdas in Python (StackOverflow.PythonLambdaLoop) and C# (StackOverflow.C#LambdaLoop) having rather strange behavior with regards to lambdas within loop (or maybe it's C++ peculiarity that it behaves exactly as intuitively expected?).

[184] that is, for loops consisting of "strong" references such as *std::unique_ptr<>*.

On Programming Languages and (Re)Actors

My second cent in this regard is that at least with (Re)Actors, it is easy to combine (Re)Actors 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-(Re)Actors infrastructure code and intra-(Re)Actor code) was written in C++. Great performance, full control, no problems with GC, everybody was really happy, etc. etc. But finding good C++ developers for app-level job isn't easy <sad-face />
- 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.
- At this point, there was a line-to-line translation project of outside-(Re)Actor C++ infrastructure code into Java (to facilitate writing (Re)Actors in Java) – along the lines which will be discussed in Vol. IV's chapter on Things to Keep in Mind[185]
    - This outside-(Re)Actor Infrastructure Code in Java was compatible at message format level with C++ code, which means that from C++ (Re)Actor standpoint, Java-based (Re)Actor was indistinguishable from a C++-based one, and vice versa.
    - So, C++ and Java (Re)Actors could interact easily (after agreeing on interfaces, of course), without no problems whatsoever. In particular, Java (Re)Actors were able to "subscribe" to the data "published" by C++ (Re)Actors, and get all the updates in real-time (most of the data necessary was already published by C++ (Re)Actors, so Java (Re)Actor subscribing to the data they needed, was mostly possible without changing C++ code).

At this point, there was a line-to-line translation project of outside-(Re)Actor infrastructure code into Java (to facilitate writing (Re)Actors in Java)

In a different project (and similar situation), a JavaScript (Re)Actor was produced to allow Server-Side scripting (in addition to existing C++ (Re)Actors). In this case, C++ outside-of-(Re)Actor code was re-used, which called *react()* (written in JavaScript) from within. Pretty much the same approach can be (more or less easily) extended to all the other programming languages of interest.

[185] we could try to go JNI route instead, but we preferred pure Java and didn't regret this decision

In both cases, all the paradigms of our (Re)Actors were transparently maintained for all the (Re)Actors across all the supported languages. This included more or less the following things:

- *react()* was a single access point to our (Re)Actor, see Vol. II's chapter on (Re)Actors
- timer actions were expressed in terms of timer messages (though now I'd probably prefer same-thread futures or coroutines instead, see Vol. II for discussion)
- extensive support for communication was provided, including:
    - support for non-blocking RPCs (it was OO-style same-thread callbacks, but now I'd probably prefer same-thread futures or coroutines instead, see Vol. II)
    - support for state synchronization interfaces (see Vol. I's Chapter on Communications) with same-thread callbacks
- support for some of the recording/replay goodies described in Vol. II's chapter on (Re)Actors

Supporting Multiple languages/compilers/JITs: Is It Worth the Trouble?

The next obvious question on the way to multiple programming languages is the following:

Are such cross-language approaches worth the trouble of implementing them?

Well, as always, YMMV, but from my experience the answer is

Absolutely!

In such a multi-language development paradigm (whether (Re)Actor-based or not, though I prefer (Re)Actors for other reasons) 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 cross-language approach (whether (Re)Actor-based or not), we're no longer restricted to one single VM, or to the availability of specific compilers which compile into that single VM. With cross-language paradigm 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 approach 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

Another very practical reason to keep an ability to go cross-language – is that in larger real-world projects, it happens to be silly to restrict your developers to one single programming language (tool, …). Just one real-world example. There was a game which used C++ and only C++ for both Client and Server; it was fine for a while, until some point later a need arose for some additional integration with 3rd-parties. It didn't need to be ultra-fast (i.e. "stop the world" didn't cause any trouble), and in fact was mostly "glue" code. At the same time (as it often happens) supply of C++ developers in the city was pretty much exhausted (i.e. finding C++ developers became difficult). On the other hand, finding Java developers for this Server-Side subproject was easy – and Java did the job perfectly well too.[186]

Another example is when you need to have web-based SQL-based reporting; and with all due respect to C++, writing reports in C++ is a crazy idea <sad-face />. Having a separate team working in {ASP.NET|PHP|Python+Django|…} will work MUCH better.

These two examples are just a tip of a huge iceberg of real-world cases where you SHOULD want to separate your teams – and to give each team a choice of whatever-language-they-want-to-use (while providing a way for them to communicate with each other over well-defined interfaces). Doing otherwise, while possible, will severely hit you as soon as you're past, say, 30-50 developers or so.

Of course, there will be LOTS of developers who'll say "hey, <insert-their-favorite-language-here> is THE BEST one for everything out there"; however, for each and every programming language there is a field where it is, well, sub-optimal (to put it mildly). Have you ever tried to convince your Game Designers to write in C++, kernel developers to write in JavaScript, write a memory-constrained program in Java or C#, or a readable one in Perl? Sure, all these things are possible – given enough time that is, but *efforts* necessary to do different things, vary significantly from one language to another one. In other words, yes – there are different horses for different courses.

[186] Of course, as the languages are not that different (see "On Programming Languages as Such" section above) - it is not that difficult for Java developers to learn C++ - but it is still easier not to do it <smile />

Oh, BTW, to make it perfectly clear – I am NOT arguing to have all of your code in all the languages (I am not *that* crazy <wink />); what I am arguing for, is to have a common framework which allows for interoperability between large chunks of code written in different languages. Chunks being "large" is important here: I am not speaking about having ten-line piece of C within JavaScript; rather it is about having the whole teams working in different languages on their separate projects separated by well-defined APIs.

Supporting Different Programming Languages within the Same Project

Assuming that I've managed to sell you the idea of using cross-language development, our next question is "how to implement it?"

From my experience, there are two possible approaches. The first one is related to line-to-line translations (of the Infrastructure Code, that is); line-to-line translations can be implemented in a pretty much the same manner as was discussed in Vol. IV's chapter on Things to Keep in Mind. I done it myself translating 50K-LoC Infrastructure Code from C++ into Java line-by-line – and it worked like a charm (well, at least compared to trying to do a completely separate implementation perfectly compatible over the wire).

The second approach is to have some common denominator (usually C/C++)you're your Infrastructure Code, and then to integrate this common denominator into each of the programming languages you need. Usually, it is not that difficult. For example, you can have C++ communication code running under JNI and calling your Java *MyReActor.react()* from there. Or under CLI, it is possible to have unmanaged code doing pretty much the same thing. Or with LuaJIT/V8, it is possible to have C++ app calling an appropriate script engine.

Which way to use – is up to you; usually I prefer line-to-line translations (IMO they tend to cause less trouble in the long run), but YMMV.

Inter-Language Equivalence Testing: Yet Another (Re)Actor Replay Benefit

Right above we discussed inter-language porting of Infrastructure Code. However, in some cases, you may also need to port a part of your (Re)Actor code from one language to another one. It may happen, for example, to optimize the time-critical piece of code, or more generally - to rewrite the whole thing into language-which-is-better-suitable-for-the-job. 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, if you're using (Re)Actors, there is an easy way to test the code equivalence. The procedure goes as follows:

- "record" a big chunk of inputs and outputs for (Re)Actor-being-ported (and running old code); "recording" can be done along the lines described in Vol. II's chapter on (Re)Actors, and may be done even in production.
- "replay" it in lab on the new code. (as described in Vol. II)

    - 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 game, if there is no bug which has manifested itself after the first four hours after new code deployment, there won't be any bugs in Game 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, and comparing corresponding variables.

Bingo! After this kind of equivalence testing – we can be reasonably confident that new version (even if it is written in a completely different programming language(!)) is indeed *exactly* equivalent to the old one.

Chapter 9 Summary

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors
-- Unknown author, XXI century

Trying to summarize our enormously-long Chapter 9 on just a few pages:

- Two major deployment architectures for games are “Web-Based” and “Classical”
    - With “Web-Based” Deployment Architecture, the most non-trivial thing is caching (and usually it should be *write-back* caching to reduce DB load)
        - Surprisingly, (Re)Actors can be used for Web-Based Architectures too.
    - With “Classical” Deployment Architecture, it is all about stateful In-Memory processing (and all kinds of simulations tend to be mapped very naturally there)
        - IMNSHO, (Re)Actors really shine here – but TBH, you still can get away without them
    - “Hybrid” approach (a.k.a. Mixed Stack) is also possible
- Regardless of specific architecture you use – you SHOULD have a separate DB Server App, separated by DB Server API
        - DB Server API MUST be expressed in terms of Game Logic (without any SQL in sight)
        - All operations within your DB Server API MUST be inherently atomic
- If you can have LOTS-of-players+observers for a single Game World – chances are that you may need Front-End Servers
- If your game is fast-paced – Regional Datacenters are likely to be necessary to reduce latencies
    - When implementing them – make sure to avoid naïve architectures (those with databases being completely separated).
- My strongly-preferred approach to DB Server App is based on heretical-for-any-DB-person (but working very well in more-than-one-major-real-world-game) single-writing-DB-connection approach

    - In chapter 9, it is only cursory mentioned. Much more detailed discussion will follow in Vol. VI's chapter on Databases
- When it comes to MOGs, term "cloud" is badly overloaded.
    - "Video streaming" a.k.a. "pixel streaming" doesn't fly
    - "File streaming" is just dynamic loading of parts of the Client – and will be discussed in Vol. V's chapter on Client Updates
    - PaaS/SaaS have two very different flavours:
        - Those with proprietary APIs will lock you in, and decision to use them MUST NOT be taken lightly
        - Those PaaS/SaaS which have de-facto-standard APIs (including MySQL API etc.) are merely deployment-time decisions which we can ignore for the time being
    - IaaS is by far the most popular cloud service model out there.
        - We will likely need to use *both* rented-per-month servers, *and* cloud (rented-per-minute) servers
        - Developing for *one single* cloud provider is rarely a good idea
        - Instead – we should develop for a generic *AllocateServer()/DeallocateServer()* API (which will be implemented for a specific cloud provider closer to deployment time).
- For Server-Side OS – Linux has a slight technical advantage over Windows, though both can be made to work technically
    - Real advantage of Linux comes in when we're take into account licensing costs for hundreds of servers
    - Requirements for Game Server boxes and for DB Server box are quite different
- For Server-Side programming – unlike with Client-Side, C++ is no longer default programming language, but is rather "one of several viable options"
    - Other contenders mentioned include *Java*, *Python*, *C#*, and *Node.js*
        - *Lua* future is unclear because of *LuaJIT* future being unclear.
- It can *easily* happen that different parts of your system are better suited for different programming languages
    - To address it - I am advocating for writing different parts in different languages (however – for early development stages it is usually not a strong requirement)
        - This requires enabling cross-language coarse-grain interactions (as one example - between different (Re)Actors written in different programming languages).

Bibliography

Baryshnikov, Maksim. n.d. "Engineering Decisions Behind World of Tanks Server."

Beardsley, Jason. n.d. "Seamless Servers: The Case For and Against." In *Massively Multiplayer Game Development*.

Bray, Brandon. n.d. *The .NET Framework 4.5 includes new garbage collector enhancements for client and server apps.* https://blogs.msdn.microsoft.com/dotnet/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps/.

Corbet, Jonathan. n.d. *"NUMA scheduling progress".* https://lwn.net/Articles/568870/.

Cybersource. n.d. *"Linux vs Windows. Total Cost of Ownership Comparison".* https://static.lwn.net/images/pdf/cybersource-tco-study.pdf.

n.d. *DPDK.* http://dpdk.org.

Duquette, Patrick. n.d. "6.2 Implementing a Seamless World Server." In *Game Programming Gems 5*.

IDC. n.d. *"Windows 2000 Versus Linux in Enterprise Computing".* https://www.cetic.be/IMG/pdf/TCO.pdf.

n.d. *Introduction to Receive Side Scaling.* https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/introduction-to-receive-side-scaling.

Lameter, Christoph. n.d. *"NUMA (Non-Uniform Memory Access): An Overview".* https://queue.acm.org/detail.cfm?id=2513149.

Lightstreamer. n.d. http://www.lightstreamer.com/.

Ligoum, Dmitry. n.d. "private communications with."

n.d. *London Stock Exchange gets the facts and dumps Windows for Linux.* http://www.itwire.com/opinion-and-analysis/the-linux-distillery/28359-london-stock-exchange-gets-the-facts-and-dumps-windows-for-linux.

n.d. *netmap - the fast packet I/O framework.* http://info.iet.unipi.it/~luigi/netmap/.

n.d. *New techniques to develop low-latency network apps.* https://channel9.msdn.com/Events/Build/BUILD2011/SAC-593T.

'No Bugs' Hare. n.d. *"Memory Leaks and Memory Leaks".* http://ithare.com/memory-leaks-and-memory-leaks/.

Noyes, Katherine. n.d. *"Five Reasons Linux Beats Windows for Servers".* http://www.pcworld.com/article/204423/why_linux_beats_windows_for_servers.html.

n.d. *Predicting the Performance of Virtual Machine Migration.* https://www.cl.cam.ac.uk/~sa497/akoush-mascots10.pdf.

Redis.CAS. n.d. http://redis.io/topics/transactions#cas.

RFG. n.d. *"TCO for Application Servers: Comparing Linux with Windows and Solaris".* http://www-03.ibm.com/linux/whitepapers/robertFrancesGroupLinuxTCOAnalysis05.pdf.

n.d. *Scaling in the Linux Networking Stack.* https://www.kernel.org/doc/Documentation/networking/scaling.txt.

StackOverflow.C#LambdaLoop. n.d. *"Captured variable in a loop in C#" where="StackOverflow".* http://stackoverflow.com/questions/271440/captured-variable-in-a-loop-in-c-sharp.

StackOverflow.PythonLambdaLoop. n.d. *"What do (lambda) function closures capture in Python?".* http://stackoverflow.com/questions/2295290/what-do-lambda-function-closures-capture-in-python.

Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli. n.d. "Architectural Breakdown of End-to-End Latency in a TCP/IP Network."

Zubek, Robert. n.d. *"Engineering Scalable Social Games".* http://gdcvault.com/play/1012230/Engineering-Scalable-Social.

—. n.d. *"Private communications with".*

Chapter 10. Fault Tolerance

Anything that can go wrong will go wrong.
-- Murphy's Law

When speaking about business-critical systems (and our game certainly qualifies as such), one all-important question which you'd better have an answer to, is the following: "What will happen if some piece of hardware (or software) fails badly?" Of course, within the scope of this book we won't be able to do a formal full-scale FMEA for an underspecified architecture, but at least we'll be able to give some hints with regards to "how to build architecture which is able to withstand failures?"

FMEA

https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis

Failure mode and effects analysis (FMEA) was one of the first systematic techniques for failure analysis.

On SPOFs vs MTBFs

Most of the time, whenever speaking about Fault Tolerance, we can hear analysis in terms of SPOFs (="Single Points of Failure"). While SPOF-based analysis is all fine and dandy, way too often it devolves into something like [TODO: dunce]"hey, this system has a SPOF, so it MUST be worse than this system without a SPOF". This statement is not only horribly wrong,[187] but relying on it has probably caused more failures than any-other-misconception-about-Fault-Tolerance.

MTBF is the ONLY thing which matters

Instead, when speaking about Fault Tolerance, the only thing we actually care about, is "how often our system fails"; in other words –

it doesn't really matter whether we really have a SPOF – as long as overall system reliability (measured as MTBF) is higher than alternatives.

Overall, having a SPOF is known to be not *that* bad, as long as SPOF is very simple and its chances to fail are extremely low. Just as one example – Really Fault-Tolerant hardware-based boxes such as Stratus and NonStop, when dealing with potential CPU failures, simply run several[188] CPUs in parallel and compare their outputs (to determine whether one of CPUs has failed); on the other hand – with such

MTBF

https://en.wikipedia.org/wiki/Mean_time_between_failures

Mean time between failures (MTBF) is the predicted elapsed time between inherent failures of a system during operation

[187] Actually – it is even wrong in theory, especially if we take *all* the things into account, more on it in [[TODO]] section below

[188] usually – four, but this discussion goes beyond the scope of this book

an approach, the comparator itself is actually a SPOF; it is just that it is so simple and fails so rarely (at least compared to failure rates of CPUs), that a question of comparator's failure can be ignored for all the practical purposes.[189]

As a very practical result –

Whenever somebody[190] tells you "this system is better because it has fewer SPOFS" – make sure to ask "but what about *real-world* MTBFs?"

Note that "real-world" is very important here, as theoretical analysis of redundant systems tends to provide MTBFs which are *way-too-good-to-be-true*. An especially common mistake in this regard is assuming certain events to be *independent* while they're actually not; the most devastating result of such a mistake I know, wasn't from computers, but from aviation: for a DC-10 plane, they theoretically calculated that the odds against all three hydraulic systems failing simultaneously are as high as a billion to one [https://en.wikipedia.org/wiki/United_Airlines_Flight_232]; however – as it has been seen on an unfortunate UA flight 232, all the hydraulic systems were located close enough to be hit by debris coming from one single uncontained failure of the engine. As for computer-related MTBFs of redundant systems being estimated too high – we'll see it below.

Adding Fault Tolerance Can Make Your MTBF Worse

Now, armed with this all-important observation, let's take a closer look at these MTBF numbers (and apply them to the real-world server-based system). Let's consider a system which can be run using one 4S/4U Server Box for its OLTP DB; as we'll see in [[TODO]] section below – 4S/4U box is often enough to run up to *several hundred thousand* of the players, so such as scenario is highly practical.

From what I've seen, your usual 4U/4S box has an MTBF of about "one failure in 10 years" (that is, if we monitor and repair partial failures such as disk failures, power source failures, fan failures, etc.); this is more or less consistent with MTBF of 45000 hours (~5 years) for a comparable server discussed in (Determining the Availability and Reliability of Storage Configurations n.d.).[191]

It means that if we'll run our database from such a *single* 4U/4S box, it will fail once per 5-10 years even without any Fault Tolerance. And for certain of your servers (such as DB Server) this is exactly what I'm usually suggesting to do (unless you *really* need Fault Tolerance, such as for stock exchanges). The reason for this suggestion is three-fold:

- First, for most of the games out there – MTBF of 5-10 years qualifies as "good enough"
  - NB: to account for this potential failure, we still need to run an "async replica" (or at least to ship DB logs in real-time to a different box) so we have

[189] at some point, Stratus was boasting that 80% of all the nuclear attack warnings go via Stratus boxes; IMO, it does qualify as "good enough" for pretty-much-any-other-use-too.

[190] Especially a salesman

[191] Note that for 2S/1U servers MTBF is usually significantly lower, but still in the range of 3 years or so

a copy which is usually only a few seconds / minutes behind. While recovery from such a scenario will still be a pretty big headache – if happening once per 10 years[192], it won't be *too* bad.

- Second – trying to eliminate single points of failure in such a system will bring in additional systems with additional complexity, and MTBF of failure of such additional systems (especially software-based ones, and Gates forbid, driver-based ones) is usually *significantly lower* than these 5-10 years. As a result, instead of decreasing chances of failure, adding fault tolerance will INCREASE them.
    - I cannot even count how many times I've heard the dreaded phrase of "our system has failed because of misconfigured failover scripts". Moreover, at least in half of such cases more detailed investigation reveals that failover wasn't caused by a real hardware failure – but rather by misusing failover for other purposes (testing/maintenance), or by a failure of fault detection mechanisms (see also "Fault Tolerance – on Failure Detection" section below).
- Third – to have a really Fault Tolerant DB Server, we'll inevitably need to have some synchronous exchanges between two server boxes (for example, some kind of synchronous replication). And any kind of synchronous stuff can easily lead to very unexpected unnecessary system failures. Just one real-world example:
    - Once upon a time, I've seen a RAID HDD failing in a strange way – it didn't really fail (and didn't even report predicted failure), but slowed down 10x or so. As such, neither RAID nor any other fault-tolerant subsystem was able to say that HDD has failed; on the other hand – the whole system was slowed to the point when it was utterly unable to handle the load (so it effectively was a system downtime). Worse – in a synchronous Fault Tolerant configuration, chance of downtimes due to such failures is DOUBLED compared to a non-Fault-Tolerant ones <sad-face />. This, in turn, leads to lower MTBFs of the whole system – exactly due to introducing supposed Fault Tolerance (!).

instead of decreasing chances of failure, adding fault tolerance will INCREASE them ☹

BTW, to be perfectly clear: I am NOT arguing that Fault Tolerance is always bad; instead – I am saying that depending on specifics (and especially if applied in a careless way under the premise of "hey, smart guys will do it for us"), adding Fault Tolerance *might* lead to a significant *decrease* in overall system reliability (such as significantly lowered MTBFs).

On MTBFs of redundant systems

My observations/calculations above look counter-intuitive, and before relying on them, we have to be 100% sure they're solid – so let's do a bit of math. After all, intuitively, when speaking about adding redundancy to a system, we expect the end-result to be better

[192] actually, less frequently, as most of the real-world failures will leave your data intact

(actually – MUCH better) than that of non-redundant system. Let’s see whether such an expectation is *that* universal as it seems to be.

Let’s consider a redundant system Z consisting of 2 redundant components X and Y. Now we need to introduce MDT (=“Mean Down Time”), which is the mean time between the component going down and it being brought back to operation; MDT is usually measured in hours and usually ranges from 8 hours to several days.[193]

Let’s note that the maths below, while perfectly common and accepted (see, for example, Chapter 8 in [TODO: Smith]) is using quite a few implicit assumptions. In particular, it assumes that (a) MDTs are negligible compared to MTBFs, and (b) that failure probabilities (inverse of MTBFs) can be added together (i.e. that failure probabilities are small enough to say that non-linear effects when adding probabilities, are negligible).

Note that both these assumptions, while potentially confusing, do stand in most real-world situations. What we’ll be concentrating on is a different implicit assumption – the one which doesn’t usually stand <sad-face />.

//WARNING: INVALID IMPLICIT ASSUMPTION AHEAD

At this point, it is common to assume (erroneously! See below) that redundant system Z will fail if and only if one of the following scenarios happen: (a) after component X fails, component Y will fail within MDTx (i.e. while component X is still being repaired); or (b) after component Y fails, component X will fail within MDTy (i.e. while component Y is still being repaired). The probability of such a failure of component Y within the MDTx, assuming that MTBFs are large, and MDTs are relatively small compared to MTBFs, is

Pyx = 1/MTBFy * MDTx

NB: relying on assumption (a) above

It (still erroneously) means that MTBFz can be calculated as

$\mathbf{MTBFz_{incorrect}}$
= 1 / (1 / MTBFx * Pyx + 1 / MTBFy * Pyx)
= 1 / (1 / MTBFx * 1/MTBFy * MDTx + 1 / MTBFy * 1/MTBFx * MDTy)
= MTBFx * MTBFy / (MDTx + MDTy)

NB: relying on assumption (b) above

//END OF INVALID IMPLICIT ASSUMPTION

[193] Note that MTTR (=”Mean Time To Repair”), while closely related to MDT, doesn’t normally include such things as ‘How to get the replacement part from supplier to your datacenter’ – and so is not directly usable for our MTBF analysis.

It looks all grand and dandy (and with typical MTBFs of 3–5 years and MDTs being maximum 3–5 days, we'd have an $MTBFz_{incorrect}$ of thousands of years – wow!) –

until we notice that there is one thing which is utterly unaccounted for in the formula above: it is the MTBF of the redundancy system itself.

Let's name it MTBFr.

Practically, MTBFr needs to cover all the components which form the redundancy system itself. Just one example: if our system uses a 'heartbeat' Ethernet cable between two nodes to detect failure of the other node, then failure of this cable is likely to lead to all kinds of trouble (including extremely disastrous 'split-brain' failures), and so it needs to be accounted for in MTBFr. In a similar manner, network cards (and their respective drivers(!)) serving this 'heartbeat' cable, also need to be included into MTBFr. Moreover, if this cable and NICs are made redundant (which would be quite unusual, but is certainly doable), they will still have their respective MTBFr, and moreover there will be some kind of software (or, Linus forbid, drivers) handling this redundancy, which will also have its own MTBFr. And so on, and so forth.

With MTBFr in mind (and realizing that whenever redundancy system itself fails – the whole thing will fail too) – $MTBFz_{correct}$ can be written as

$$\mathbf{MTBFz_{correct} = 1 / (1/ MTBFz_{incorrect} + 1/ MTBFr).\ (*)}$$

How large your MTBFr is depends, but I can assure you that for the vast majority of real-world cases, it will be much smaller than those hyper-optimistic 'thousands of years'.

And in practice (and especially whenever redundancy is implemented in drivers – more on it below), MTBFr can be easily *much* smaller than MTBFx. For example, if MTBFr is 1 month (BTW, it is not a joke, I've seen quite a few redundancy systems which exhibited less-than-a-week MTBFs under serious loads) while having MTBFx at 3–5 years – the formula (*) will show that $MTBFz_{correct}$ is 30x-50x smaller than original non-redundant MTBFx.

I rest my case.

A Real-World Story

To complement theory with practice, a real-world story. Once upon a time, there was a game with hundreds of thousands of players. And a pre-IPO technical auditor has come to analyse it. At some point, he asked about the downtime numbers, and was surprised how low they were ("how you guys can possibly have downtimes which are 5x-10x rarer than the rest of the industry?"); after all the necessary logs were presented – he had to accept it. A half an hour later, looking at the deployment diagrams, he asked "hey guys, why don't you use clusters?" – and the answer was "that's because we want to have those downtimes which are 5x-10x lower than the rest of the industry".[194]

I tend to attribute these real-world effects exactly to the balance of (a) very high single-server MTBF to start with; and (b) low MTBFs of software- and especially driver-based fault-tolerant solutions.

Hardware Fault Tolerance is-much-better-than Software Fault Tolerance is-much-better-than Driver-Based Fault Tolerance

I have already bashed software- and driver-based fault tolerance quite a few times above, so now let's make it very official:

IMNSHO, as a Big Fat Rule of Thumb™, Driver-Based Fault Tolerance is a DUD.

DUD

http://dictionary.cambridge.org/dictionary/english/dud

something that has no value or that does not work

Let's discuss different implementations of Fault Tolerance in more detail – and see how they usually fare in the real world:

- **Hardware-Based Fault Tolerance** usually works. This includes things such as HP NonStop or Stratus; also fault-tolerant hardware boxes (such as reduntant switch or router) *MAY* work too.
  - NB: even if such solutions are using their own drivers – they still MIGHT work.[195]
- **Software-Based Fault Tolerance** *MAY* work. This includes both app-level solutions (such as HAProxy, though see some reservations about it below) *and* OS-level solutions including OS-level hardware-independent drivers – though see below re. hardware-specific drivers.
  - Still, Software-Based Fault Tolerance (especially the one which is *NOT* integrated with your app and does things "automagically" behind the scenes) carries a *huge* risk of dreaded "misconfigured failover scripts" (which, from my experience, are responsible for over 50% of all the system failures).
- **Fault Tolerance based on Hardware-Specific Drivers.** By far the worst thing which can happen with your Supposedly-Fault-Tolerant system – is if it uses some hardware-specific drivers allowing your stock OS to interact with their custom hardware. From what I've seen – such things tend to cause *much more trouble than working without them.*
  - A few real-world examples:

[194] of course, lack of clusters wasn't *the only* reason for having such low downtimes; in particular, I'm sure that using (Re)Actors (as discussed in Vol. II's chapter on (Re)Actors) were also a very significant contributing factor

[195] as we'll see below for Drivers, it is interaction between two very different parts (OS and hardware) which tends to kill their reliability; when you have your-own-hardware with your-own-OS – it *MIGHT NOT* apply

        - Once upon a time, one of the best-known (and best-working) manufacturers, Stratus, has tried to create a much-cheaper alternative making two Windows boxes redundant (it was known as Stratus RADIO); I've seen one of the first such boxes as a part of their "beta" program. When we tried to load it - after merely half an hour of running it under a load, it went to infamous Blue Screen of Death, with a failure in one of those custom drivers used to make things redundant. Apparently, the problems were *that* bad, that Stratus has completely abandoned this effort before even releasing RADIO to production.
        - At some point, we were playing with a RAID card where most of the RAID was actually implemented as a part of their custom driver. Guess what – after a few months, we found that some of the data on that RAID is corrupted <ouch and double-ouch! />.[196]
        - Just recently, I've seen The Ultimate Nightmare™ of all the communication failures – massive data corruption observed at TCP level; ironically, it was caused by redundancy drivers for NIC <sigh />.
    - The reason behind this all-important observation is simple and well-known: *it is just that 99% of hardware guys are usually not too skilled in writing drivers*. Or, without making is seem that I blame them[197] – *writing a hardware driver requires knowing BOTH intimate knowledge of the hardware, AND intimate knowledge of the dark art of writing-drivers-for-specific OS*; not only that people with knowing *BOTH* these things are very scarce, but also even for such people thinking of BOTH these things at the same time is very likely to push them over 7+-2 cognitive limit, usually with devastating results.

Actually, the situation with driver-based Fault Tolerance is *that* bad, that by default (i.e. without any credible 3rd-party data showing that this-particular-driver-based-Fault-Tolerance system DOES work for several years in a real-world without causing too much trouble) I'd advice to

Stay Clear from Driver-Based Fault Tolerance.

From my experience:

- More often than not, it is better to stay *without* Fault Tolerance at all (relying on MTBFs being high enough), rather than to use Driver-Based Fault Tolerance.
- If/when the hardware failures become a problem – it *is* almost-universally better to use (or implement-your-own) Software-Based Fault Tolerance rather than relying on a 3rd-party Driver-Based one.

Communication Failures

[196] Later, it was confirmed that it was a bug in the driver, and a fix was released. Too little, too late…

[197] Actually, I deeply sympathise with their predicament

Now, let's see what can possibly go wrong within our deployment architecture (such as one of the architectures discussed in Chapter 9)? First of all, there are switches (or even routers/firewalls) residing between our servers; while they're not shown on our diagrams in Chapter 9, but they still exist (see Vol. VII's chapter on Preparing for Launch for further discussion on them).

While these network-level boxes can (and often SHOULD) be made redundant, even in this case their failures (as well as transient software failures of the network stack on hosts) may easily cause occasional packet loss; worse than that – some of these failures also may cause TCP disconnects on inter-server connections. Even worse – we have to keep in mind that TCP does *not* provide any reliable guarantees of the stream being non-corrupted;[198] what we have to do to ensure stream integrity – is to use crypto-level checksums (at least 128-bit ones), and if the checksum is broken, we won't have any other options than to drop-and-reestablish offending TCP stream ourselves.

Recovery from Channel Failures

While saving for such hardware-or-driver failures, TCP disconnects for Server-to-Server communications will be almost-non-existent (that is, as long as we're staying within one single datacenter) - but hardware and especially driver failures are generally bad enough to account for them (BTW, as soon as we do it – according to Murphy's Law the chances of such failures go down <wink />; still – it is a good insurance-like tradeoff).

To deal with communication failures, our Server-to-Server protocols need to account for a potential channel loss (such as a TCP channel loss) and allow for guaranteed recovery after the channel is restored. In Vol. I's chapter on Communications, we've already discussed some Server-to-Server protocols which allow to recover from transient failures – and these can (and SHOULD) be used for Server-2-Server communications. Very briefly, implementation-wise we have to:

- Whenever the channel is lost – re-establish it
- ensure re-sending on the sending side whenever-channel-is-lost-and-re-established.
- eliminate duplicates on the receiving side.

As a whole – it will guarantee us *exactly-once* delivery for each of the messages, and if our mechanics also guarantee preservation of the message ordering - it will become indistinguishable from the normal work (i.e. where the channel has never been lost).

Of course, Server-2-Server channels loss being undetectable is the Holy Grail™ of the Fault Tolerance – so if we do care about fault tolerance, we certainly should go this way. One word of caution though – we need to keep in mind that with an active channel loss detection, such a loss will be "indistinguishable"/"undetectable" only *as long as we don't speak about delays* (in other words, channel loss *will* lead to delays, we just need to make sure they're not *too* bad); still, such infrastructure-level recovery of transient failures tends to be a *huge* improvement compared to dealing with it ad-hoc at app level: with

[198] see also above re. my recent experience with a redundancy-driver-induced failure of the TCP stream

infrastructure-level recovery, at least we shouldn't care about "what to do if the message got lost/TCP channel got broken" at app-level <phew />.

Speaking of delays (and keeping them at bay): one nasty thing which can happen to connectivity – is "hanged" TCP connections; these may be detected (by both sides of communication) by some of keep-alive techniques (application-level and/or TCP-level) which will be discussed in Vol. IV's chapter on Network Programming; what is important for us now, is that regardless of the specific mechanism used to detect "hanged" connection, the only thing we can meaningfully do whenever we detect that the connection has "hanged" – is to abort it and re-establish, so all the mechanics mentioned above is still necessary.

Implementing Redundant Channels

The next question we'll have, is "how to make sure that if Ethernet switch fails, we still have our system working?" For a long time, in this regard, I was tacitly agreeing with a common approach of "hey, it is not a developer's problem, let admins handle it for us"; unfortunately – while redundancy at the switch level does work good enough, at host level it is universally implemented by hardware-specific drivers – which has been seen to cause all kinds of trouble, from kernel panics to corrupted data <sad-face />.

As a result – I am currently arguing for a DIY-style connectivity redundancy, along the following lines:

- there are two separate networks to connect your Server Boxes (in addition to any out-of-band management network). Each of these networks has its own switch (completely isolated from another one), and its own range of IP addresses (such as one having 10.0.x.x and another one having 10.1.x.x).
- each server has two separate NICs – one for each network (and each with its own IP address)
- every Server-2-Server connection is actually implemented as *two* TCP connections,[199] one going via 10.0.x.x, and another one – via 10.1.x.x.
- each message is sent over both these TCP connections. On the receiving side – it the first message arriving which wins (with duplicates eliminated in a manner similar to those Server-2-Server transient failure protocols discussed in Vol. I's chapter on Communications).

That's it. With this approach, we do have perfect redundancy (including – and without those-ever-failing-drivers too <phew />.

[199] +encryption-level checksum at least to provide acceptable level of integrity for in-transit data

Game Server Failures

Of course, in addition to communication failures, any of the servers can go badly wrong. However, ways of handing Game Server failures and DB Server failures tend to be rather different. Let's start our analysis with catastrophic failures of our Game Servers; as we are likely to have dozens (if not hundreds) of Game Servers – chances are that they will fail MUCH more frequently than just a few of our DB Servers.

There are tons of solutions out there claiming to address this kind of failures; however, we should keep in mind that as a rule of thumb, the stuff marketed as "High Availability", doesn't help to preserve in-memory state: what you need if you want to avoid losing in-memory state, is "Fault-Tolerant" techniques (see also "Server Fault Tolerance: King is Dead, Long Live the King!" section below).[200]

Note that the stuff marked as 'High Availability', usually doesn't help to preserve in-memory state: what we need to avoid losing in-memory state, is 'Fault-Tolerant' techniques.

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 Vol. VII's chapter on Preparing to Deployment 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 your game is a stock exchange), you'd be pretty much able to ignore this problem entirely.

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 (rather on bulletproof prevention of *all* the failures).

Containment of Game World Server failures

[200] on the other hand, IF your architecture has your Game Servers are stateless (or at least their state can be recovered; this includes Stateless-App-with-in-Memory-Write-Back-Cache and Disposable-Stateful-Apps discussed in Chapter 8) – we CAN use more conventional (and easier-to-achieve) High Availability for our Game Servers (though NOT for Write-Back Caches if there are any).

The very first (and rather obvious) technique to minimize (though not to eliminate) the impact of your Game World server failure on your whole game site, is to make sure that your Game World Server reports relevant changes (without sending the whole state) to DB Server as soon as they occur. As a result, 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.

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.

This much is more or less obvious, but now there is a less obvious part: 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, achievements, and artifacts preserved.

In general – you should save to DB *at least* end of each of the Game Events (more on Game Events in discussion on 'No Bugs' Rule of Thumb in Chapter 8); however – in certain cases (especially if there are significant changes to the stats of your player when she's essentially in a single-player mode) you *MAY* need to save your state more frequently than that.

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

Except for some rather narrow special cases (such as stock exchanges), saving all the state of your game world into DB won't work due to performance/scalability reasons (more on it in Chapter 8). Also, we need to keep in mind that even if we 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-currently-in-progress for more than 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 (more on it in discussion on ‘No Bugs’ Rule of Thumb in Chapter 8).

If you disrupt the game-event-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.

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.

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 Reactor)

The second pitfall on this way is related to DB consistency issues and DB Reactor API:

Your DB Reactor API MUST enforce logical consistency

For example, if (as a part of your very own DB Reactor API) you have two DB-related 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” occurrence, you should have a special DB Reactor request “PC X took over artifact Y from PC XX” (and it should be implemented as a single DB transaction by DB Reactor); 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 money, with any other goodies in between.

Actually, even better is to have an EndOfGameEvent request – which would do lots of things, *including* artifacts changing hands if necessary.

Server Fault Tolerance: King is Dead, Long Live the King!

While mitigating effects of the Game World Server failures is one of the easiest thing to do – it is not the most reliable one. Moreover – for quite a few subsystems (such as any kind of Write-Back Cache) we may easily need to have real Fault Tolerance.

When speaking about real Server Fault Tolerance, we need to distinguish between "high availability" and "fault tolerance". Usually, "high availability" in this context merely means that you have a hot-swap server, which is ready to be launched as soon as necessary; however, "high availability" usually does NOT include saving in-memory state – and this can be fatal for most of our Game Servers. On the other hand, if we want "fault tolerance" – which usually includes preserving up-to-date in-memory state too (!) - there are some ways to implement it, allowing us to have our cake and to eat it too.

However, let's keep in mind, that all fault-tolerant solutions are complicated and costly, and in the games realm they often qualify as an over-engineering (even by my standards <wink />). On the other hand, there are two notable exceptions:

- If your game is a stock exchange or a reasonable facsimile, you're likely to want full-scale fault tolerance
- Quite often, there are a few Really Critical Servers within your game, which MAY warrant fault tolerance. Examples of such Really Critical Servers include:
    - Servers which are critical for the operation of your game as a whole – these routinely include such things as directory servers, DB servers (see also "DB Server Failures" section below), Write-Back Caches, etc.
    - Servers which run that highly publicized Tournament of the Year which you cannot afford to fail.

Still, I'd say that implementing fault tolerance for ALL your Game Servers is an overkill for the vast majority of games out there, so think twice before going that route.

Clusters: HA, not FT

When speaking about handling Server faults within your team, pretty much inevitably somebody will say[201] "Hey! It's easy! We'll just use cluster by <insert-cluster-implementation-here>!".

Unfortunately, clusters (at least all those cluster which I've seen) are NOT providing "Fault Tolerance", but rather merely a "High Availability" <sad-face />. In other words, with clusters in case of failure current in-memory state is NOT preserved. This alone makes them completely unusable for Fault Tolerance purposes as defined above.[202]

[201] this tends to happen especially often after reading a enthusiastict-to-the-point-of-being-outright-misleading promotional material by hardware or OS manufacturer

[202] as we'll discuss in Vol. IX's chapter on Deployment Architecture Take 2, I also have a strong dislike at least of traditional clusters-with-shared-disk even for the purposes of HA;

Fault-Tolerant Servers: Damn Expensive

Historically, fault-tolerant systems were provided by damn-expensive hardware such as (FAULT TOLERANT AVAILABILITY FOR CRITICAL APPLICATIONS AND VIRTUALIZED WORKLOADS n.d.) and (NonStop (server computers) n.d.). These beasts have everything doubled (and CPUs often quadrupled(!)) to avoid all single points of failure, and 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[203] – you can pretty much forget about them <sad-face />.

Fault-Tolerant VMs

Modern Fault-Tolerant VMs are using one of two technologies: 'virtual lockstep' and 'fast checkpoints'. Unfortunately, each of them has its own limitations <sad-face />.

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, as there is no magic involved, there are some significant caveats, described below. BTW, make sure to 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 have to admit that I didn't have a chance to try them myself yet <sad-face />.

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 observe, modern Fault-Tolerant VMs are using one of two technologies: "virtual lockstep" and "fast checkpoints". As we'll see below, "virtual lockstep" is better suited for latency-oriented applications (games included), but unfortunately, it doesn't seem to be supported anymore by major VM hypervisors <sad-face />.

Virtual Lockstep: Not Available Anymore?

very briefly, my problems with such clusters include such things as shared HDD being a SPOF, Really Bad behavior in case of failure of the heartbeat link (see also discussion in "Fault Tolerance – on Failure Detection" in this Chapter), and ease of misconfiguring a failover script, which tends to make clusters having *smaller* MTBF than MTBF of good single servers <sad-face />.

[203] That's where I was able to lay my clavicles on one of them – and it did work like a charm too

The idea behind virtual lockstep (which was used in VMWare vSphere 4-5) is based on treating VM pretty much as a… deterministic finite state machine (which is fundamentally similar to our deterministic (Re)Actors as discussed in Vol. II's chapter on (Re)Actors). Virtual lockstep takes one single-core VM, logs all its inputs, passes all these inputs to the secondary server, and runs a secondary (backup) VM there, feeding all the inputs to that backup VM all the time (see (How Fault Tolerance Works n.d.) for description of virtual lockstep in VMWare, and also see discussion on our own implementation of DIY virtual lockstep in "Low-Latency Fault Tolerance: DIY Virtual Lockstep" section below).

Let's see how the failure of the primary VM is handled in this case. Whenever computer hosting primary VM suffers from a sudden death, we can simply switch all the communicating parties to the secondary one (how to detect the failure and how to switch communications is a different story, but it is more-or-less doable). However, we MUST beware of a scenario when primary VM already got some packet (message/whatever) and replied to it (right before it sudden death), and secondary VM didn't get this logged packet (because host of primary VM has died); this scenario can easily lead to a fatal discrepancy between VM states, with all the hell going loose because of it. Avoiding this problem is surprisingly simple: while primary VM can and should process all the packets/inputs at full speed, the host of primary VM SHOULD NOT release *replies* of the primary VMs to these inputs, until secondary VM confirms that it got the inputs-which-caused-these-replies (NB: it is not necessary to wait until secondary VM has *processed* the input, it is simply necessary to *receive* the input and have the presence and order of inputs fixed for the secondary VM before allowing primary VM to reply). This creates a "delayed replies window" of processed-but-unsent replies on the primary host, which means that

Fault Tolerance introduces an additional latency while the system is working normally and without any failover.[204]

This additional latency is pretty much inevitable for *any* implementation of Fault Tolerance; however, if your both Server boxes are sitting next to each other in the datacenter (and everything runs smoothly as it should) – for virtual lockstep we're speaking about the additional latencies of the order of hundreds microseconds (i.e. within 1ms). For most of the games out there, while potentially undesirable, this kind of delay is not fatal; also, as we'll see below, this is a very minor delay compared to alternative fault-tolerant implementations.

One additional thing to be mentioned is that amount of traffic between virtual-lockstepped server boxes is roughly equal to their I/O – and for boxes which do most of their work with the network (such as Game Servers), it is roughly equal to their network traffic – in other word, it is extremely unlikely to overload a usual 1G Ethernet link (which is a kind of minimal-standard these days).

Overall, virtual lockstep seems to be very-well-suited for our purposes; however – it also has one drawback: virtual lockstep cannot possibly handle more-than-single-core VMs. This is

[204] in case of failover, latencies will be much worse, but as long as failover happens once in a blue moon (and as the alternative is to crash completely) – it hopefully won't impact your game too badly.

related to an inherent non-determinism of multi-threaded programs and multi-core VMs (strictly speaking – to ensure determinism in multi-core environments, we'd need to log-and-send-to-secondary-box all the inter-core memory sharing occurrences, and this is one thing which is even a hypervisor cannot achieve with any reasonable performance).

While at least for a (Re)Actor-fest architecture we'd be able to live with this single-core restriction (hey, (Re)Actors are single-core anyway), being single-core-only was deemed unacceptable by vendors, and starting from vSphere 6, VMWare has switched to "fast checkpoints" fault tolerance – which allow multi-core VMs but at the cost of MUCH higher latencies (and MUCH more inter-server traffic too) <sad-face />.

In other words – virtual lockstep would be ideal for games (and other latency-oriented things), but – it does not seem to be supported anymore <sad-face />.

Checkpoint-Based Fault Tolerance: Latencies and Even More 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").

The very rough description of checkpoint-based fault tolerance goes as follows. First of all, we need to implement "checkpoints" – these are usually implemented over a list of CPU pages (4K in size for x86/x64) which were modified since previous "checkpoint". When we have these "checkpoints", and as long as we don't let any replies coming out of our primary server out (keeping them in "delayed replies window"), all the calculations which are made by primary VM between the "checkpoints", become inherently unobservable from the outside. On the other hand, as soon as we have committed the "checkpoint" to a secondary server – we can release all the replies which were made *before* this "checkpoint" was started. While decisions-which-happen-after-this-checkpoint, can be different when it is run for the second time on the secondary VM (as noted above, we can't have any determinism with multi-core VMs) – as long as nobody knows about the results of the first run, the whole system works exactly "as if" the inputs were never processed on primary VM (all the replies by primary VM after last checkpoint, never leave the "delayed replies window"), so the answer to the question "whether it was decided one way first, but then re-decided another way" becomes almost completely unobservable.[205] For more details on checkpoint-based VMs, see [TODO:Remus].

Note that with checkpoint-based fault tolerance (and unlike with virtual-lockstep-based Fault Tolerance) it is not necessary to feed all the inputs from the primary VM to the secondary VM. Or from a bit different perspective – with checkpoints, it is *state* which gets synchronized (and with virtual lockstep – it is inputs which are guaranteed to be the same,

[205] BTW in theory, using side channels or insider information, it may be possible to build an attack which would see the outcome on the first VM before it is released to the public, and then, if outcome is not favorable for the attacker, to bring the box with first VM down to retry the whole thing on the secondary box hoping for a better outcome this time(!)

and we're reconstructing the state from the same inputs based on determinism). This leads to two important observations (both apply only to checkpoint-based fault tolerance, *and* if we're not feeding the same inputs from primary VM to a secondary VM[206]):

- strictly speaking, primary and secondary VMs cannot be made identical, as some of the packets will be almost inevitably lost after failover switch to the secondary VM. However, as packet loss is a normal occurrence and OS/apps within VM are expected to handle packet loss anyway – this still works.
- it is an open question whether virtual lockstep or checkpoints have more traffic between the primary and secondary boxes: if most of the interactions are read-only – then checkpoint-based implementation may result in less traffic, but if there are frequent changes to the state (this includes disk writes(!)) – then virtual lockstep implementation will win traffic-wise. On the other hand, this is usually a moot issue unless we're speaking about inter-data-center fault tolerance (i.e. unless you want to withstand failure of the whole datacenter).

So far so good, but the problem with such "checkpoints" is that, as noted above, to achieve consistency guarantees, "delayed replies" in "delayed reply window" need to be kept until the moment when the next "checkpoint" is reached. Worse than that, in systems such as VMWare FT and Xen Remus,

these "checkpoints" are reached only every several dozens of milliseconds – and often only every 100-300ms – and it means that with checkpoint-based Fault Tolerance, we need to delay replies, introducing latencies for up to hundreds of milliseconds too <very sad-face />.

This observation essentially rules out "checkpoint-based" fault tolerance for Game Servers of quite a few games out there.[207] Which is a pity, because besides latencies, checkpoint-based fault tolerance has many desirable properties, such as support for multiple CPU cores and N+1 redundancy (though to be fair to virtual lockstep, we should also mention that in addition to worse latencies, checkpoint-based Fault Tolerance tends to have significantly higher CPU overhead).

DIY Fault-Tolerance for (Re)Actor-fest architectures

If third-party methods of achieving fault tolerance discussed above are not good enough (and they quite often won't) – then we have an option to implement fault tolerance ourselves. However, to do it – we'll need to have our Game Logic to be implemented as (Re)Actors (and not just any (Re)Actors, but *deterministic* (Re)Actors). On the other hand, as soon as we have our logic implemented in terms of deterministic (Re)Actors – we have at least two distinct ways to implement DIY fault tolerance – and (which is most important for us) in a latency-friendly manner.

[206] both VM hypervisors I know, seem to implement checkpoint-based fault tolerance without feeding inputs of one VM to another one
[207] though as a rule of thumb, it may still be used for DB Servers, Write-Back Caches, etc.

Low-Latency Fault Tolerance: DIY Virtual Lockstep

First of all, we can implement Virtual Lockstep ourselves – along the same lines described above for VMs. The idea stays pretty much the same:

- we have Game Logic implemented as a deterministic (Re)Actor
    - it is also desirable to have the (Re)Actor implementing serialization/deserialization of its state. While some level of protection from server failures benefits can be obtained without serialization – this protection is limited (see below)
- we have 2 server boxes instead of one (2 virtual servers MAY do too, though to achieve fault tolerance, these virtual servers they should reside on different physical boxes, and I'm not sure that your cloud service provider will provide this kind of guarantees)
- on the first box, we're running primary instance of our Game Logic (Re)Actor (of course, we can run several instances of different (Re)Actors on the same box)
    - Infrastructure code running on the first box, logs all inputs of the (Re)Actor (this includes all input events, *and* all "wrapped" calls etc. – see Vol. II's chapter on (Re)Actors for details), and sends them to the secondary box.
    - At the same time, primary (Re)Actor is processing these inputs and generates outputs
    - Each of the outputs of the primary (Re)Actor is put on hold until the moment when the secondary instance of the (Re)Actor (running on the second server box) confirms that it received all the inputs preceding this output (or more strictly – all the inputs which may have contributed to the computation of the output). As soon as confirmation is obtained – output can (and should) be sent to the recipients.
- on the second box, we're running secondary instance of our Game Logic (Re)Actor, accepting all the inputs coming from the first box, and applying them to the secondary instance of the (Re)Actor. Due to the determinism of our (Re)Actors, they have the same state after processing the same inputs.
- In case of catastrophic failure of the primary server box – we make secondary box a primary one (and former secondary instance – a primary one too). In case of catastrophic failure of the secondary box – we keep operating primary one (just not waiting for confirmations from the secondary one anymore)
    - At this point, we handled one single failure within the system without losing integrity. However, *redundancy* of the system is still compromised.
    - Note that in some cases, working without redundancy can be acceptable. For example, if you restart your system once a day anyway – then the chances of two boxes failing within the same day can be seen as negligible. A similar situation arises when it is not your whole system, but the (Re)Actors in question have a limited life span. If this is the case – it *may* allow you to implement serialization only for those (Re)Actors which have long life spans.
    - To restore redundancy – we'll need to build a secondary instance from the primary one. To do it, we'll need to (a) serialize primary instance, (b) move serialized state to the replacement box (which becomes secondary one), and (c) to create secondary instance from deserialized state.
    - Bingo! We restored redundancy, and can withstand another server failure.

This DIY Virtual Lockstep schema is pretty good for our game-related purposes; in particular, it exhibits pretty good latency (for the fault tolerant system, that is). In practice, if your server boxes are located next to each other, you should be able to limit the additional latency introduced by fault tolerance, to values below 1ms.

The only significant drawback of this schema I can think of, is that it has N*2 redundancy: in other words, it requires to keep two server boxes to have redundancy for one server box. Apparently, this can be improved without sacrificing latencies.

Another Low-Latency Fault Tolerance: DIY N+1 Reactor-based Redundancy: Logging Server

To improve over N*2 redundancy which can be achieved by "Virtual Lockstep" model - the following approach can be used to achieve N+1 redundancy:

- we still have Game Logic implemented as a deterministic (Re)Actor. And (Re)Actors implement state serialization/deserialization too.
- we still have 2 server boxes instead of one (and notes above about virtual servers still stand)
- on the first server box, we're running a Logging Server. Logging Server merely handles all the inputs for the (Re)Actor(s), logs them, and forwards them to the second box. Note that to keep deterministic nature of the (Re)Actors, this almost inevitably will include timestamping[208] events.
    - When "call wrapping" is necessary to achieve determinism (see Vol. II's chapter on (Re)Actors for relevant discussion), such requests should be transferred back to the logging server, and processed-and-logged there <sad-face />
        - On the other hand, if we're speaking about file I/O requests (or any other disk-based I/O such as DB requests), they can be processed right on the second box; however, there is a caveat – if doing so, all non-constant files need to be considered as a part of the Reactor state – and therefore MUST be serialized when storing the state into the log; for certain types of (Re)Actors, this can easily become "way too much" <sad-face />
    - To avoid keeping logs forever, logs on Logging Server are circular, and whenever a wraparound in logs is about to happen, Logging Server requests and gets current state of the (Re)Actor from the second server box. In other words, Logging Server always keeps "state X + all events after the state X" for all the Reactor(s) it serves.
- on the second server box, we're running the instance of our (Re)Actor (it is the primary instance, but also is the only one) – using *only* the inputs coming from the Logging Server.

[208] or some other type of ordering

- in case of catastrophic failure of the first server box (the one with the Logging Server) – we just replace it (with a spare), and restore redundancy by requesting the (Re)Actor state from the second box (and logging all the events after this state).
- in case of catastrophic failure of the second box (the one running the instance of our Reactor) – we (a) replace the box, (b) restore (Re)Actor from the last logged serialized state (the one stored on Logging Server), and (c) "rollforward" – i.e. apply all the events in the log to the restored instance of the (Re)Actor. Note that re-applying events doesn't need to take as much wall clock time as it took to process them originally; in other words, if your last state was stored 5 minutes ago, it can be perfectly feasible to rollforward input events in 5 seconds – while overall CPU usage will be the same during this process, all the waits can be skipped while rolling forward.

in case of catastrophic failure of the second box (the one running the instance of our (Re)Actor) – we (a) replace the box, (b) restore (Re)Actor from the last logged serialized state (stored on Logging Server), and (c) "rollforward" - apply all the events in the log to the restored instance of the Reactor

This Logging-Server-based schema is better than DIY Virtual Lockstep in a sense that it requires only N+1 servers (plus a spare) to run the whole thing. On the minus side, handling of "wrapped" calls is admittedly ugly – and while they can be avoided most of the time, in some cases it will lead to the files becoming a part of the (Re)Actor state – and this needs to be serialized on a regular basis. In other words – using this model for your DB Server isn't likely to fly.

DIY Fault Tolerance – Checkpoints

In addition to two options discussed above, at least in theory, there can be other ways to implement DIY fault tolerance. In particular, the same principle of the Checkpoint-Based Fault Tolerance as was discussed with regards to VMs above, can be applied to (Re)Actors too. On the other hand – exactly as with VMs, latencies are going to be very significant, so for most of the environments, I'd rather not use it.[209]

DIY Fault Tolerance – Connections and IPs

In all those DIY Fault Tolerance models which we discussed above, we ignored a question "how to replace the server at connection level?" (or more precisely, "how to re-establish all the connections to the Server instance with the connections to the replaced instance?")[210]

[209] also – if checkpoint-based stuff works for you, there is no real reason to write DIY fault tolerance in the first place, as checkpoint-based one is already available and working

[210] For VMs it is actually easier, as packets can be lost anyway and they can rely on in-VM OS's and apps to handle packet losses.

To solve this problem, we can use one of the two following approaches – IP replacement or re-connecting at source.

IP replacement can work as follows:

- on the replacement server, we simply change IP address
    - Note that changing IP over SSH[211]-which-goes-over-the-same-IP can easily cause weird chicken-and-egg problems. In practice, out-of-band management is almost-the-must for such trickery (more on out-of-band management in Vol. VII's chapter on Preparing for Launch).
    - Even more importantly – *before* changing IP address on the replacement server, we need to be 200% sure that the old server (*supposedly* failed one) does NOT respond to the-IP-we're-about-to-change. Here, we can easily run into all the kinds of partial failures – for example, if our heartbeat app on the supposedly-failed server is down, IP stack can be still up; in such a case, changing IP on a replacement server isn't likely to work.
        - To deal with it – the best thing is to turn off our supposedly-failed-server-box completely. In practice – turning it off using out-of-band-management such as IPMI/iLO/DRAC/…, while still not 100% reliable in theory, is usually good enough for our purposes.
    - [TODO/wiki ARP]after changing IP on our replacement server, so-called ARP caches for this IP on *other* server boxes will still point to the MAC address of the *old* server, causing packets these other server boxes send, to be lost for quite a while (up to 5-15 minutes <ouch />). To deal with it, we may want to do one of the following:
        - after changing IP address, we may drop ARP caches on all the *other* servers directly connected over the Ethernet to this one. See, for example, discussion on it in (Bergsma n.d.).
        - in addition to changing IP address, we may change MAC address of the new server to match MAC address of the old server on the corresponding interface (and also may want to bring Ethernet interface down and up). On the other hand – this further significantly complicates configuration (introducing risks of your MAC mappings being out of date), so I'd rather *not* use this approach. Also – *if* going this way, importance of out-of-band management grows even further.

Overall, I don't like relying on trickery such as changing IPs and MACs on the fly (though I admit that this *may* be because I am a developer and not an admin <wink />).

An alternative to changing IP address of the server box on the fly, is what I call "re-connect at source":

- After the failure is detected, each of the Clients of the affected server (these can be Clients or other Servers) is asked to use new IP address for communicating with the Server X from this point on
    - note that we have to make sure that this process of distributing new IP address (or more precisely – of notifying about the failure) is itself fault-

[211] Or whatever-other-mechanism

    - tolerant (answering the question "what if the server box which distributes these requests, goes down?")
- As an alternative, we can provide the secondary IP address to the Clients "well in advance", and make the Client to fails to use secondary IP as a fallback in case of problems communicating with the primary one.
    - IF we're relying on this approach, we MUST make sure that secondary server does NOT accept connections until it is told to take over by that-entity-which-detects-primary-box-failure (and turns primary server off); otherwise – we'll have a *huge* risk of "split-brain" failures (when half of the Clients are connected to the primary server, and half of the Clients are connected to the secondary one, and there is no consistent picture at all).
- note that handling of the connectivity issues discussed in "Communication Failures" section above, is still necessary.

If we're using some kind of an MQ product (such as RabbitMQ, or, IMO better, ZeroMQ) for our Server-2-Server communications[212] – handling reconnections can be done by MQ (MQs tend to have LOTS of mechanics to handle exactly this kind of stuff). Moreover, IMNSHO this is one of the *very few* scenarios where MQ products are *somewhat* useful in our game-like environments; still – *if* going this way, make sure that (a) your MQ system of handling addresses is itself fault-tolerant, and (b) that it detects "hanged" connections (such as "hanged" TCP connections) fast enough for your purposes.

Overall, for a connectivity-level fault tolerance – all three approaches will work, but personally I tend to give a preference to "re-connect at source" models, though this preference is not something I'd really insist on if facing *really fierce* resistance from admins <wink />.

DIY Fault Tolerance in case of Almost-Determinism

In some cases, our Server Apps may be not deterministic, but *almost-deterministic.* This happens in the case when we have some kind of non-deterministic behavior which still leads to *substantially similar* results. Examples of such *almost-deterministic* behavior include such things as multi-threading, and particularly GPGPU calculations (where it is usually very difficult to avoid multi-threading effects). In such cases, with respect to Fault Tolerance, at least two approaches are possible:

- to re-sync states completely on regular basis, delaying all replies until the re-sync happens – similar to what VM checkpoint-based fault tolerance is doing (see "Checkpoint-Based Fault Tolerance: Latencies and Even More Latencies" section above). In DIY (Re)Actor-fest environment, one way to achieve re-sync is via serializing (Re)Actor state (which has to be stored separately, for example as discussed in the *Another Low-Latency Fault Tolerance: DIY N+1 Reactor-based Redundancy: Logging* Server section above)
- to say that ALL such *almost-deterministic* results, while being released to the outside world, are always seen as *transient.* This means that while these results exist, they're NOT used for any decision-making, and are only treated as *advisory*.

[212] a bit more on it in Vol. I's chapter on Communications

- In game environment, one example of such *transient* and *advisory* stuff is publishable data sent to Clients; as discussed in Vol. I's chapter on Cheating, we SHOULD NOT make any decisions on the Clients, so any such data can be used *only* for visualizing. Moreover, as discussed in Vol I's chapter on Communications, often we have to implement Client-Side Interpolation and/or Client-Side Prediction, which are inherently approximate – so Client is always ready to *reconcile* with the updated version of the Game World when it gets the next update from the Server.
  - Therefore, if the ONLY thing which is not really deterministic, but just *almost-deterministic*, is Publishable State (and Client is using Publishable only for visual purposes – which it SHOULD as we want to have our server authoritative – see discussion in Vol. I's chapter on Cheating) – we still MIGHT be able to use Fault Tolerance along the lines outlined above. If our Server App fails and we restart it – on the second run results MAY be different; however, (a) the differences will be self-healed very soon (as soon as the next server update comes in); and (b) during self-healing, the differences will be only visual, and negligible too. Note that achieving "negligible" part of it is not something which we can easily quantify, but in MOG context, to make Client-Side Prediction work, keeping the differences between Server and Client "negligible" is a substantial part of our work anyway, so *usually* nothing special needs to be done in this regard to deal with *almost-determinism* in case of failover.

Game Server Fault Tolerance – What to Use

Actually, the very first question you should ask yourself in this regard is that "do we *really* need to have real Fault Tolerance for our Game Servers?" Sure, Game Servers will fail on a regular basis, but In fact – for most of the games out there, ensuring that we only mitigate the impact of such failures, is sufficient.

IF you happen to need Fault Tolerance for your Game Servers (which BTW should be decided on GDD level – see Vol. I's chapter on GDD) – then keep in mind that for latency-sensitive games currently-available VM-based Fault Tolerance is not likely to work well. In such cases – you *may* need to resort to (Re)Actor-based DIY Fault Tolerance (DIY Virtual Lockstep or DIY Logging Server ones). BTW – *if* doing it, make sure to think *hard* about "how to avoid split-brain conditions"; as these conditions are known to be *much-worse-than-simple-failures <ouch and double-ouch! />* - being 100% sure that your system is not prone to them even under most-unlikely-but-still-possible conditions, is absolutely critical.

DB Server Failures

It's All about Numbers – and Nothing Else

When we're speaking about DB Server failures – the first thing I need to tell is that, counter-intuitively (and unless you're running a stock exchange or a bank), failures of DB Server are not *that* important to deal with; not because they have less impact than Game World Servers (actually, they do have *much* more impact), but because they're much less likely to happen that a failure of one-of-Game-World-servers.

Let's take a closer look at this (admittedly rather controversial) statement. To do it, we'll need to jump a bit ahead and briefly mention a typical DB architecture (it will be discussed in detail in Vol. VI's chapter on Databases). With a typical game (from stock exchange to an MMO), it is likely to have a central "OLTP DB" (the one processing all the transactions) – and asynchronous replicas to run reporting (as it will be discussed in Vol. VI, there are a few other DBs too, such as archive one and analytical one, but they're not too interesting for us at the moment).

For read-only replicas, Fault Tolerance is trivial – just run two replicas and that's it. So, the only real thing we really need to discuss – is Fault Tolerance for the OLTP DB Server – the one processing all the transactions in the system.

As we'll see in Vol. VI (the same chapter on Databases) – most of the time, properly designed OLTP DB can run up to 100K simultaneous players (and 10M+ of daily writing transactions) off one single DB connection (effectively utilizing only 1.5-2 CPU cores). It means that even if we need more than 100K simultaneous players (we'll discuss how to scale your DB to achieve it, in the same chapter on Databases within Vol. VI) – one 4U 4-socket workhorse server (such as HP DL580) with 4x 24-core CPUs (and assuming that our DB load is typical OLTP, so we're not writing videos to the DB which would quickly cause us to become I/O-bound), will be able to run DB Server for at least several millions of simultaneous players. It is the order of

DB-Wise, we can run millions of simultaneous players from one single 4S/4U box.

Now, using the data from the *Adding Fault Tolerance Can Make Your MTBF Worse* section above, we can expect that MTBF for such 4S/4U box will be around 3-5 years or so. And with this kind of numbers, from what I've seen, adding Fault Tolerance may *easily* reduce your MTBFs instead of improving it.

Exception – Stock Exchanges

Right above, I have argued to ignore Fault Tolerance mechanisms on the grounds that adding fault tolerance for a single high-quality server tends to *reduce* MTBF instead of supposed increases. One exception to this rule of thumb, is when you're running a stock exchange or a bank (i.e. when any failure involving data loss is utterly unacceptable).

In such a case – be prepared for LOTS of research and analysis to make sure you find a Fault-Tolerant system that DOES work. Personally – if the budget permits, for this kind of systems I'd certainly try to use either HP NonStop, or one of Stratus boxes. From what I've seen –

such fault-tolerant systems tend to have *significant* advantage over the detect-and-switch-to-another-node systems (which, most importantly, suffer from a

If Stratus/NonStop is out of reach but you still need to implement Fault Tolerance – be prepared to a looong process of research and even more importantly, testing of the software-based allegedly-Fault-Tolerant solutions (and for your customers sake, make sure to avoid solutions which are based on customized hardware drivers – these tend to cause *much* more trouble than they're worth).

Fault Tolerance for DBs

C.Y.A.

https://en.wikipedia.org/wiki/Cover_your_ass

Cover your ass or C.Y.A. describes activity, usually in a work-related or bureaucratic context, done by an individual to protect himself or herself from possible subsequent criticism, legal penalties or other repercussions.

In such (BTW, really rare) cases when you DO need Fault Tolerance for your DBs but can't afford a box such as Stratus or NonStop – you basically have two options. The first one is to use RDBMS-provided fault tolerance. As a rule of thumb, I am arguing against it (that is, unless your goal with implementing Fault Tolerance is mere to C.Y.A.). The reason for it is two-fold:

- First, as a rule of thumb, most of these things are notoriously poorly implemented (and even more poorly tested)
    - in particular, "heartbeat" failures can easily bring your system to the knees (and even cause split-brain scenarios, which tend to be *much-worse-than-simple-failure* for DB servers).
    - Moreover, as discussed below in "Fault Tolerance – on Failure Detection" section – the task of failure detection cannot be 100% really solved with only two servers; and two-server configs are still predominant ones in the industry. Multi-node systems tend to fare better in this regard – but see below on latencies.
    - As a result – *very* intensive testing is necessary when you're using these systems, and it is really unclear what is more difficult – to conduct good black-box testing of the 3rd-party system – or to write your own one (so testing won't need to be black-box anymore).
- Second – traditional RDBMS-based Fault-Tolerant configs are not that latency-friendly (which will cause quite a bit of trouble, especially with single-DB-connection architectures I will be arguing for in Vol. VI's chapter on Databases). Having a DIY Fault-Tolerance allows to avoid this problem.
    - This tends to be *especially* true for multi-node systems (which *can* be made immune to heartbeat failures and split-brain conditions but latencies tend to go through the roof <sad-face />).

The second option is to use VM-based Fault Tolerance (such as VMWare FT or Xen Remus) for your OLTP DB Server Box. Two practical things to note in this regard:

- You DO need *at least* 10GBit/s link between your primary and secondary boxes.[213] Make sure to monitor its utilization and upgrade when necessary.
- Due to significant impact of the latency between your-DB-Server-App and your-RDBMS, make sure to run *both* of them within the same VM <sic! />.

In theory – you can also try a DIY Fault Tolerance for a Database Server too (along the same lines as for Game World Servers) – but honestly, with the state including the database – it is going to be a *really big challenge* (even by my heavily-leaning-towards -DIY standards <wink />).

Fault Tolerance – on Failure Detection and Split-Brain Conditions

One thing which is inherent for *any* kind of Fault-Tolerance-with-failover (whether DYI or otherwise) – is failure detection: before we can fail over to a reserve box – we have to detect a failure. Moreover,

if failure detection mechanism can leave both our Server Boxes thinking that they're masters at the same time – we're in Deep Deep Trouble™.

Such as situation is known as a "split-brain condition", and for stateful systems it is fundamentally irrecoverable in general; among other things, it means that

At least for stateful systems, split-brain condition can easily lead to significantly-worse results than simple failure.

Overall, implementing Failure Detection properly (including guarantees against split-brain conditions) tends to be *very tricky.*

There is quite a bit of research on Failure Detection (see, for example, (Bongiovanni n.d.)); however, in practice most of the time Failure Detection use some variation of the heartbeat: every side sends "heartbeats" at pre-defined intervals, and if we don't receive anything from the other side – we consider the other side as failing.

However, naïve heartbeat implementations are vulnerable to heartbeat link failures <ouch! /> (and to make things worse, such a failure in a naïve implementation will very likely lead to that dreaded split-brain condition too <double-ouch! />). Let's consider the following example:

- There are two servers (Master and Slave) sending heartbeats to each other
- if Slave doesn't receive heartbeats from Master – it assumes that Master is dead, and takes over.
    - In quite a few cases (in particular, for clusters), take over is implemented via changing IP address of the Slave (usually followed by an ARP flush) – along the lines discussed in the *DIY Fault Tolerance – Connections and IPs* section

[213] that is, *if* your VM Fault Tolerance is checkpoint-based, but as far as I know, as of mid-2017 both VM FT and Xen Remus *are* checkpoint-based with no plans to change it

- If Master indeed fails – then this kind of simplistic failover does work as intended
- However, if it is the link-used-for-sending-heartbeat which fails (or any of the NICs-serving-heartbeat-link fails, or drivers for such NICs fail, or Master fails partially, leading to about the same result), we're in lots of trouble:
    - not receiving heartbeats anymore, Slave will decide that Master is dead, and will change it's own IP
    - But if Master's non-hearbeat interface is not dead – we'll have two Servers (Master and Slave) with the same IP address (but different MAC addresses) within the same network – and this is enough to effectively make the whole thing unusable even at the network level.

Actually, the problem with the naïve heartbeat system described above, is not really with IP change; in fact, regardless of IPs, if we have two nodes, and they can stop communicating with each other, while accepting Clients – Slave node will decide that the other node is dead, so we'll get a classical split-brain: two separate universes which pretend to work, but each having its own state, which is actually worse than non-working at all; in particular, if one of the Clients gets some unique resource on one Server, and another Client gets the same unique resource on another Server, reconciling them is not really possible (in extreme cases, such reconciliation is not possible except that in court of law <n-times-ouch! />).

As it was already briefly mentioned above, the simplest way to avoid split-brains is to make sure that there is a *single* entity making this "other side is dead" decision (or one single entity should *enforce* this decision for *all* the entities involved), *and* to turn one of the servers off (or disconnect it) entirely. One schema which would work along these lines, would be the following:

- Both Master and Slave are connected to the same *managed Ethernet switch* (as we'll see in Vol. VII's chapter on Preparing for Launch, it is an *extremely* common practice anyway)
- When our heartbeat script on the Slave detects a potential failure of the Master – first thing it does, is connecting to that *managed Ethernet switch* where both our nodes are connected to.
- While connected to the switch, our Slave disconnects Master from the switch (using *shutdown* for CISCO switches, for more on scripting CISCO switches – see, for example, (Scripting a Cisco switch with Python and Expect n.d.)).
    - If this attempt succeeds – it means that our system is no longer redundant, but more importantly – it means that our Slavee *did* take the system over
- Now, our Slave can start changing IPs/taking connections/…; as Master is already disconnected at switch level – any of the actions taken by Slave, cannot possibly lead to the dreaded split-brain (regardless of supposedly-dead Master being really dead, partially dead, or not dead at all).

Another way to prevent split-brain, is so-called quorum-based systems with more than two nodes. While such system *can* be made split-brain-immune, at least in a case of an extremely-generic-failure they still need to use some kind of physical/network disconnect such as the one discussed above, to disable a node which could potentially-got-really-mad.

Of course, when disabling the Ethernet interface, there is still a chance for failure misdetection to become a contributing factor to overall system failure – in particular, if we're turning a working Server off, we're losing redundancy which does lower overall reliability, but well – as a Big Fat Rule of Thumb™, is still MUCH better than risking a split-brain.

Summary for Chapter 10

To summarize our findings from this Chapter:

- Redundancy != better MTBF, and MTBF is *the only thing which really matters*[214]
- Thinking about potential failures *is* important for games.
    - However, for quite a few of the games out there, the only thing we *really MUST* handle – is mitigation of the failures of Game World Servers (ensuring that it does NOT crash the whole system, and that recovery from the failure is at least *somehow* reasonable from player's perspective).
    - Surprisingly, handling DB Server failures is often not required[215] due to such failures being extremely rare (that is, *if* you run your DB on one single box).
- *if* we need to have real Fault Tolerance for Game Servers – it can be achieved
    - modern VM-based Fault Tolerance tends to add significant latencies (easily reaching 100+ ms)
    - There at least two distinct DIY Fault Tolerance schemas, based on deterministic (Re)Actors, which can work at the cost of latency penalty < 1ms
- *If* we need have real Fault Tolerance for DB Servers – it can be achieved
    - If we're speaking about stock exchanges etc. (where such requirements are the most common) – *seriously* consider HP NonStop or Stratus; they're by far the best thing to achieve *real* Fault-Tolerance.
    - As for RDBMS-level solutions – they will take *a lot* of time to make them work anywhere-close to be really reliable.
- Split-brain conditions are *really* nasty (and are observed on a surprisingly *huge* number of naïve heartbeat-based systems merely by disconnecting that heartbeat cable). As effects of split-brain tend to be *much* worse than effects of simple failure – make 100% sure that your system does NOT suffer from them (at the very least – make sure to test how your system handles unplugging of your heartbeat cable).

Bibliography

Bergsma, R. (n.d.). *Migrating an ip-address to another server: clear the arp cache of your neighbors*. Retrieved from https://blog.remibergsma.com/2012/11/15/migrating-an-ip-address-to-another-server-clear-the-arp-cache-of-your-neighbors/

[214] Ok, severity of the impact from the failure also matters – but it still has nothing to do with redundancy

[215] beyond having a replica-or-logs-which-are-a-few-minutes-behind

Bongiovanni, F. (n.d.). Retrieved from http://deptinfo.unice.fr/twiki/pub/Minfo/DistributedAlgo/Cours_FailuresDetectors-Consensus-SelfStabilization.pdf

Determining the Availability and Reliability of Storage Configurations. (n.d.). Retrieved from http://www.dell.com/content/topics/global.aspx/power/en/ps3q02_shetty?c=us

How Fault Tolerance Works. (n.d.). Retrieved from https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.availability.doc_41/c_plan_understand_ft.html

NonStop (server computers). (n.d.). Retrieved from https://en.wikipedia.org/wiki/NonStop_(server_computers)

Scripting a Cisco switch with Python and Expect. (n.d.). Retrieved from https://www.electricmonk.nl/log/2014/07/26/scripting-a-cisco-switch-with-python-and-expect/

FAULT TOLERANT AVAILABILITY FOR CRITICAL APPLICATIONS AND VIRTUALIZED WORKLOADS. (n.d.). Retrieved from http://www.stratus.com/solutions/platforms/ftserver/

Chapter 11. Pre-Coding Checklist: Things Everybody Hates, but Everybody Needs Them Too. From Version Control to Coding Guidelines

Along the course of our Volumes I to III, we've discussed a lot of architectural issues specific and not-so-specific to MOGs, and now you've hopefully already drawn a nice architecture diagram for your upcoming multiplayer game.

However, before actually starting coding, there are still a few things to do. Let's take a look at them one by one.

Version Control

[[TODO: picture with "referring to Skyrim by Bethesda Softworks"]]

Always keep the entire universe required to build your software in version control
-- Neal Ford

To develop pretty much anything, you *do* need a version control system (VCS, also known as source control system). I don't want to go into a discussion *why* do you need it, just saying that there is a consensus out there on version control being necessary for all the meaningful development environments. Even if you're single developer, you still need version control: the version 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 version control are so numerous that nobody out there dares to develop without it.

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

The very first question about version control is "what to put under your version control system?" And as a rule of thumb, the answer goes like

You should put under version control pretty much everything you need to build your game, but usually NOT the results of the builds

[TODO:rabbit_pullquote img="areyoucrazy" quote="hey, what is the mesh we should use with this source code?"]And yes, “pretty much everything” generally includes assets, such as meshes and textures. *Moreover, it is of paramount importance to keep all the assets under the same version control system – and in the same repository - as your source code*. While there are people out there advocating using Git for code, and SVN for assets – *I am firmly against such split-brain approaches*. The main problem along this line is with being unable to synchronize two completely separate repositories, we’ll pretty much inevitably run into situation when answering a question “hey, what is the mesh we should use with this source code??” will take much longer than it should have.

On the other hand, as with most of the rules of thumb out there, there are certain (but usually very narrow) exceptions to both “everything you need to build” and “usually NOT results of the builds” parts of the bold statement above. As an exception to “everything you need to build” part, you *might* want to keep certain egregiously-large-and-barely-connected-to-your-game things (such as intro videos) outside of your version control system,[216] but such cases should be very few and far between. Most importantly,

your game should be buildable from version control system, and the build should be playable

, that's one strict requirement; however, as long as you comply with this rule - you MIGHT bend all the other rules a bit.

BTW, *if* your source significantly depends on the toolchain you’re using (which shouldn’t be the case in theory – but it does happen way too often in practice <sad-face />[217]) – *then* to comply with this rule, you *may* have to get your whole toolchain into your version control (in the extreme case – a whole VM-with-all-the-tools-used-to-reproduce-the-build can become a part of source-controlled universe, but IMO this is rarely necessary for gamedev).

As for exceptions to “not including results-of-your-build” rule of thumb – I’ve seen examples when having YACC-compiled .c files within version 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. On the other hand, keeping your *compiled executables (and worse – complete installers)* within the version control is usually undesirable. At some point, I’ve seen such a policy increasing the size of the version control DB beyond the point of being usable (and while an argument of ‘disk is cheap’ does fly to certain extent – the time necessary to checkout, as well as time necessary to backup your version control, *are* important – and both were *severely* affected by such a policy).

[TODO:rabbit_pullquote img="pointingout" pos="right" quote="Note that I am not arguing along the lines of “whether binaries belong to the version control””]Note that I am *not* arguing along the lines of “whether binaries belong to the version control”. Instead – it is about “whatever-is-necessary-to-build” against “what-we-get-as-a-result-of-our-build”. While “whatever-is-necessary-to-build” *should* include assets, and *may* include compilers-

[216] Make sure to replace them with stubs, so your build is still perfectly playable

[217] this is pretty common for embedded developers, but fortunately, for gamedev it is not *that* common

and-libraries-used-to-build – storing results of our build (whether binary or not) is rarely necessary. First, if you did a good job with storing “whatever-is-necessary-to-build”, you can reproduce *exactly* the same build, so you don’t really need to store the results; second - the value of binary executable / installer for development is extremely limited.

~~To Git or not to Git?~~ To Feature-Branch or not to Feature-Branch?

In the most of the development world, for quite a few years *Git* (or sometimes more-or-less-equivalent *Mercurial*) is considered a golden standard for source/version control. On the other hand, recently I feel a completely unexpected comeback of SVN.

Actually, if we take a closer look at the heated Git-vs-SVN debates, the main line of meaningful arguments is not really about specifics of the source control system, but is rather about branching.[218] And as most of the modern VCS (including Git, Mercurial, and SVN 1.8+) do provide useable branching[219] – the actual line of argument is not about “what our VCS *will allow us* to do” anymore, but is about “what do we *want to do*” with regards to branching, and more generally – with regards to our development flow.
As a result, we’ll discuss development flows *first*, and only then we’ll proceed towards the discussion of the pros and cons of specific VCS.

Before we start, let’s note one all-important property we should keep in mind while discussing development flows and VCS in the context of game development. First, let’s observe that, as noted above – we have to store all the game assets under our version control system. Second – let’s note that assets are generally handled by non-developers (such as artists). When these two observations are combined, it means that

As a rule of thumb, for gamedev we DO need to have our version control to be easily usable by non-developers.

[TODO:rabbit_pullquote img="thumbup" pos="right" quote="In general, it is not a problem to explain the concept of “check-in” and “check-out” to non-developers"]In general, it is *not* a problem to explain the concept of “check-in” and “check-out” to non-developers; after all – even the most conservative gamedev teams are using version control (such as SVN or Perforce), with “check-in” and “check-out” being the cornerstone of all-the-source-control-systems. On the other hand – requiring non-developers to understand rather complicated flows such as GitFlow (and Linus forbid, the concept of “rebasing”) is usually out of question.

[218] ok, there is also a “distributed-vs-centralized” argument, but this we can argue about ad infinitum, while at least for closed-source and in-house version control it doesn’t really matter *that* much

[219] though *each* not without its own gimmicks, see below for details

Development Flows: from Release Branching and Feature Branching to Trunk-Based-Development-with-Continuous-Integration and Feature-Branching-with-Frequent-Integration

Now, let's take a closer look at those development flows. Over the time, at least four rather distinct approaches to development were observed in the wild:[220]

- **Trunk-Based Development** (with Release Branching on the side)**.** Actually, Trunk-Based Development (i.e. "everybody actually works on a trunk, and directly commits to trunk") is a natural thing when you're working on the project alone – and it was used for larger projects for a while too. In practice, Trunk-Based Development becomes rather frustrating as soon as you have a team with 100+ developers making commits to trunk, where *any single commit can break the whole thing*; it has led to infamous situations of "hey, nobody can compile until this <censored> guy has his code fixed" <sad-face />…
  - Pretty often (and especially in waterfall-based development processes), Trunk-Based Development is accompanied with the practice of Release Branching; this practice goes back to times when the software was released with a biennial cycle. From the point of view of source control – it means that whenever we have a release, we needed to have two branches: one to develop bugfixes for this release, and another one – to develop our next release.
- **Feature Branching.** Historically, Feature Branching was hugely popularized by Linux development flow and Git. Essentially, Git is all about the philosophy of a more extreme form of Feature Branching, where *everything* out there can be seen as a patch. From our point of view – Feature Branching does work, but has a problem that integrating Feature Branch can easily become a significant effort; moreover, *the longer the Feature Branch lives without integration – the more difficult it will be to integrate it, up to the point of throwing away the whole development of several month in one of the branches <ouch! />.*
- **Trunk-Based-Development-with-Continuous-Integration**. As the time passed, so-called "Continuous Integration" has emerged (more on it in [[TODO]] section below), which has helped Trunk-Based-Development in a very significant way. The most important difference of Trunk-Based-Development-with-Continuous-Integration from simple Trunk-Based-Development is that with the help of Continuous Integration, *you won't be allowed to check in anything which breaks compile-and-a-bunch-of-very-basic-tests*. As a result – addition of Continuous Integration *mostly* solves that everybody-waiting-for-you-to-fix-your-bug-which-made-it-to-trunk problem <phew />, which is typical for "pure" Trunk-Based Development.
- **Feature-Branching-with-Frequent-Integration.** As noted above, one problem inherent to feature branching, is that branches can easily become out-of-sync, making merge down the road very difficult (up to the point of being outright

[220] Note that the processes described below are not really "all-or-nothing", and different aspects of them can be combined in one single development process; still - more often than not, specific real-world development process tends to lean towards one of these workflows in a more-or-less obvious manner.

impossible). To deal with it – we can adopt a policy of Frequent Integration. With such a policy in place (*and enforced by tools – more on them in [[TODO]] section below)* we don't sit in a Feature Branches for long, merging them back to trunk as soon as possible (as in "usually – daily, at the very most – once per week"). *Moreover, even if we cannot integrate our feature branch yet – we should at least re-sync changes from the trunk into our feature branch*. In turn, such a tool-aided policy *mostly* solves that problem of the long-living-Feature-Branches being difficult to integrate back to the trunk.[221]

Looking at these four approaches I can say that I clearly do *not* like first two of them (though even they can work in practice); as for the two last ones –

At this point I don't dare to claim which of Trunk-Based-Development-with-Continuous-Integration and Feature-Branching-with-Frequent-Integration is better.[222]

In other words – both these workflows will work, and "which one to choose" is more-or-less down to preferences of your team. BTW, if we think of it for a few more seconds – we'll realize that with adding of the Continuous/Frequent Integration, original approaches actually *converged* towards a more-usable-flow (with the difference between the two approaches *declining* as the life time of each Feature Branch is reduced – and reducing the life span of Feature Branch as-much-as-possible is already almost-universally recognized as being a Good Thing™, the only remaining argument is about what "as-much-as-possible" really means).

[TODO:rabbit_pullquote img="surprised" pos="right" quote="at least in theory Trunk-Based-Development-with-Continuous-Integration and Feature-Branching-with-Frequent-Integration can co-exist within the same project"]Moreover, at least in theory Trunk-Based-Development-with-Continuous-Integration and Feature-Branching-with-Frequent-Integration can co-exist within the same project (though personally I would still suggest to pick one of them to start with[223]).

Now, armed with this information – let's discuss Trunk-Based-Development-with-Continuous-Integration and Feature-Branching-with-Frequent-Integration in a bit more detail.

Trunk-Based-Development-with-Continuous-Integration

There is something very strange and unaccountable about a tow-line.
You roll it up with as much patience and care

[221] Let's keep in mind that even if we re-sync each feature branch with trunk, there is still risk of two long-living feature branches conflicting with each other, so integrations with the trunk still need to be frequent enough.

[222] In gamedev world, trunk-based development is currently much more popular – so it should be a tad safer to go this way; on the other hand – there are games developed under both models, so it is not *that* risky either way.

[223] You can be pretty sure that as the time passes, you will encounter situations when your choice is sub-optimal, so you will have to start using both of them <wink />

as you would take to fold up a new pair of trousers,
and five minutes afterwards, when you pick it up,
it is one ghastly, soul-revolting tangle.
-- Jerome K. Jerome, Three Men in a Boat
[sed s/tow-line/source code/g]

Very roughly, Trunk-Based-Development-with-Continuous-Integration goes as follows:

- Essentially, all we have is ‘trunk’ branch, where all the development goes
    - All the usual “check-in”/”check-out” logic applies
    - By default, all check-ins/check-outs go directly into ‘trunk’
        - There are exceptions, but they’re rare
            - One such exception is hotfixes – which usually involve “release branch”
- To avoid those “nobody can work because somebody has checked in one non-compilable file” – we use a Continuous-Integration tool (CI tool)
    - CI tool compiles checked-in code, and runs a bunch of very basic unit tests
    - [TODO: rabbit_pullquote img="assertive" pos="right" quote="to work efficiently – CI tool should work (and give a “green light”) not only on each commit, but before the actual commit happens"]IMPORTANT: to work efficiently – CI tool should work (and give a “green light”) not only *on* each commit, but *before* actual commit happens. As discussed below in the Pre-tested Commit section, such “pre-tested commits” are *extremely* important to reduce the number of that “everybody waits for a committed-bug to be fixed” situation.

Pre-tested commit

https://en.wikipedia.org/wiki/Gated_commit

A gated commit, gated check-in or pre-tested commit is a software integration pattern that reduces the chances for breaking a build (and often its associated tests) by committing changes into the main branch of version control.

Trunk-Based Development, when aided with a CI tool doing pre-tested commits (a.k.a. pre-commit test, delayed commit, etc.), has several important advantages over Feature Branches:

- It is simple. Most importantly – it is simple enough to be used for non-developers (such as artists). If you give your artists Perforce or a TortoiseSVN (without any branches in sight) – chances of them revolting against using version control become pretty slim (but give them Git and ask them to “rebase” to re-sync their branch – and you will certainly have a mortgage-size crisis on your hands).
- As everybody works with already-committed code - it allows for *much* more testing to be performed over that already-committed stuff; this tends to help with ironing those most-pesky integration bugs out.
- Merges are rare – which means that there is less risk of the need to revert a merge (and reverting a merge *is* a well-known counter-intuitive mess at least in Git and Mercurial, more on it below).

On the negative side - Trunk-Based-Development-with-Continuous-Integration exhibits a few not-so-desirable properties too:

- It complicates work on not-so-trivial features. As there are no “feature branches” – we have to choose between doing-everything-in-a-very-incremental-manner (which is possible, but time-consuming) – or developing for a while without committing at all (which is risky and prevents inter-developer collaboration on features).
- Cherry-picking (i.e. deciding which features should go into next build) is pretty much impossible
    - OTOH, “feature flags” a.k.a. “feature toggles” (~=”runtime decision to enable/disable a feature”) allow for rather similar functionality, though with two (admittedly relatively minor) drawbacks
        - While I don’t have problems with enabling and disabling ready-to-be-used features using feature flags – relying on feature flags to make sure that code from half-baked features is never ever used is well, rather risky (if your system can be misconfigured in production – somebody will do it sooner rather than later). In addition, feature flags also creates a risk of such never-used code living in trunk for years.
        - on the Client-Side giving away the code prematurely *may* conflict with our bot fighting efforts (more on it in Vol. VIII’s chapter on Bot Fighting).
- [TODO:rabbit_pullquote pos="right" img="thumbdown" quote="With Trunk-Based-Development-with-Continuous-Integration development model running over several years, code tends to become unnecessarily-tightly-coupled more easily than when using Feature Branches"]With Trunk-Based-Development-with-Continuous-Integration development model running over several years, code tends to become unnecessarily-tightly-coupled more easily than when using Feature Branches. While there is an overall tendency for the unattended code to become entangled (pretty much as unattended tow-lines from epigraph to this section) – with Trunk-Based-Development lacking a concept of “feature”, fighting the spaghetti code tends to require more discipline than for Feature-Branch approach.
    - On the other hand – if you’re using (Re)Actors (which, as I am arguing over this whole book, you should <wink />) – clean inter-(Re)Actor interfaces are enforced in a rather strong manner, so at least between (Re)Actors it is not *that much* of a problem.

Feature-Branches-with-Frequent-Integration

An alternative to Trunk-Based-Development-with-Continuous-Integration is Feature-Branches-with-Frequent-Integration. As a rule of thumb, it goes as follows:

- All the development is done within Feature Branches, which are then merged into the trunk; normally – no direct commits into trunk are allowed.
- Usually, *when/if* going a Feature Branch route, I am arguing for the “Git Flow” branching model by Vincent Driessen described in (Driessen). 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:

- *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/CI tests.[224] 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).
    - As for allowing direct commits (not from feature branches) into develop branch – it is arguable. If we do allow them – we'll be actually using a hybrid between Feature-Branches-with-Frequent-Integration and Trunk-Based-Development-with-Continuous-Integration (and this is not bad, but it is important to understand implications of each model).
    - Whether we use direct commits or not – for *all* the commits into *develop* branch we MUST use that *pre-commit* Continuous-Integration stuff discussed in the *Trunk-Based-Development-with-Continuous-Integration* section above. Actually, it becomes even more important for merges (in particular, because reverting merges is a mess for pretty much all the VCS out there <sad-face />).
- *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; however, it is a *really good idea* to have an *ability* to run CI tests on a feature branch – and to *use* this ability on regular basis.

[[TODO: stop sign]]There are 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

A *major* word of caution with regards to feature branches: there are developers out there who prefer to live within their own feature branch for many weeks and months, often even implementing many different features under the same branch and postponing integration for as long as they can. This is a Really Bad Practice™, and you SHOULD integrate your Feature Branches very often. Moreover, even if merge of your *feature* branch into *develop* branch is not possible – at the very least you *should* sync your branch with trunk (via "sync merge" in

[224] it is important to understand that no amount of automated testing can guarantee that the code is really working

SVN, and via “rebase” in Git) on regular basis (*at the very most* once per 2-3 days), to incorporate changes-made-in-trunk, back into your Feature Branch.

These policies are all good, but the problem is that for a team of 20+, pretty much any policy doesn’t work without ~~police~~ some way to help us to follow it.

Fortunately, there are at least two phenomena which can help with reducing life time of those way-too-long Feature Branches. First, we have to note that the very nature of merge-based source control systems tends to punish those developers 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 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 too many conflicts, but whenever they do happen,

it is the second developer to commit who becomes ~~a rotten egg~~ responsible for resolving conflicts

Print this profound truth in a 144pt font and post it on the wall to help your fellow developers merge their feature branches more frequently.

The second way to deal with those Feature-Branches-which-outlive-their-usefulness, is to have an automated tracker of long-lived *feature* branches – and to send automated reminders at least to those involved and to PM[225] (and if you *really* want to get rid of these too-long-branches – send the reminder to the whole team to create additional peer pressure). Automated notifications, while not being a silver bullet, *do* help to push developers in the right direction.

Pros and cons of Feature-Branches-with-Frequent-Integration over Trunk-Based-Development-with-Continuous-Integration actually mirror respective cons and pros discussed in the *Trunk-Based-Development-with-Continuous-Integration* section above. On the plus side, Feature-Branches-with-Frequent-Integration:

- streamlines development of the not-so-trivial features
- ensures cleaner code in the long run (in particular, it reduces unnecessary tightly coupling)
- provides an ability to cherry-pick features-to-be-compiled-in
    - This, in turn, is a practical prerequisite to rather-important Replay-Based Testing as discussed in Vol. II’s chapter on (Re)Actors.

On the negative side, this model is not really ideal either:

- It is too complicated for non-developers such as artists.
    - To deal with it – we have to say that:
        - At each given moment, each artist[226] works in one specific feature branch.

[225] =Project Manager

[226] or at least “each checkout by artist”

        - We provide simple scripts[227] checking-out and checking-in into this branch from his working directory
        - This way – from artist’s perspective, the whole flow is *the same* as with trunk-based development, and is simple enough to deal with without going into the complicated world of branches.
- While merges are less frequent with Feature-Branch-based development – each merge takes more time (especially if it was hold for too long).
    - See above about “how to push developers to commit/push more often”.
- [TODO:rabbit_pullquote img="omg" pos="right" quote="amount of naturally occurring de-facto integration testing of the committed code is reduced (and reducing the amount of testing – especially integration testing – is never a good thing)"]As commits into *develop*-branch-used-by-everybody tend to happen less frequently than for Trunk-Based Development – it means that amount of naturally occurring de-facto integration testing of the committed code is reduced (and reducing amount of testing - especially integration testing[228] - is never a good thing).
- Merges are frequent, which raises chances of having a merge revert (and they *are* way too messy at least in Git and Mercurial).

Comparing Feature-Branches-with-Frequent-Integration and Trunk-Based-Development-with-Continuous-Integration

Now, let’s compare our two candidate workflows side by side; such a comparison can be seen in Table 11.1:

	Trunk-Based-Development-with-Continuous-Integration	**Feature-Branching-with-Frequent-Integration**
Development Flow	Simple😃	Good🙂 for developers, too complicated😠 for non-gamedevs (the latter should be addressed as described above😐)
Merges	Simple, more frequent	More rare, larger
Geared Towards	Trivial/smaller features	Not-so-trivial/larger features
Naturally Occurring Integration Testing	Quite a bit😃	Limited😐
Code Quality	More tightly coupled😐	Less tightly coupled😃
Risk of Merge Revert	Very low😃	Higher🙂
Cherry Picking	Not possible🙂	Doable😃
Replay-Based Testing	Difficult😐	Doable😃

[227] GUI, whatever-else

[228] that’s where the sneakiest bugs are usually found

As we can see – pros and cons of these two approaches are pretty well-balanced, so "what to choose" becomes more-or-less a matter of whatever-team-prefers (that is, *as long as* things such as CI and monitoring of the long-living feature branches are implemented). Also – it is certainly possible to use a "hybrid" approach with more-trivial features going directly to the trunk, and more-complicated-stuff living in their own Feature Branches for up to a few weeks. [[TODO: elaborate on "hybrid", where you'll end up anyway <wink />]]

Choosing Version Control System

After we finished our (admittedly very limited) discussion on development flows – we can proceed to discussing specific version control systems. As of 2017, the following four version control systems are widely used for game development (listed in historical order of their respective first releases): Perforce, SVN, Git, and Mercurial. While, as noted above, *most* of gamedev industry is still leaning towards Perforce and SVN – there were successful games using Git and Mercurial too.

Perforce

I have to admit that I never used Perforce myself; still, I'll try to summarize arguments which gamedevs routinely provide for using Perforce:

- Unlike most of the version control systems – Perforce is oriented not only towards coders, but also towards non-coders such as designers and artists. And I have to agree that providing designers/artists with a friendly environment is indeed extremely important.
- Huge projects (those with *lots* of asset binary files, totaling terabytes) are handled without issues.
- Locking files is possible.
    - As asset files (whether they're binary or text – more on it below) are usually not really mergeable – having two artists to work on the same file is a Bad Idea™. This is where universally-frown-upon-in-programmers-world "lock file" feature comes handy.

On the minus side:

- Perforce branching is reported to be rather ugly (up to the point of being outright unusable); even worse - *data loss* has been reported to happen during Perforce merges <double-ouch! />[229].
- Perforce keeps track of your working copy on the server; while not a problem for LAN – it is a problem when you have to work remotely (which is more and more often these days)
    - While working offline is possible with Perforce, it is subject to "reconciliation" process when you're back online, which is well, ugly.
- Perforce has been reported to require to resort to out-of-the-source-control file copying and/or sharing (which is an inherently Bad Thing™) on quite a few occasions.

[229] to be fair – other users have reported working with Perforce for many years without problems, though it is unclear how much branching they were using

- Continuous Integration tools are relatively reluctant in supporting Perforce; on the other hand, with Jenkins, TeamCity and Bamboo supporting Perforce – it is not that bad either.
- “Locking files” feature can be abused (in particular, you should have a policy of “not using exclusive checkout” for the code).
- You cannot just delete file in your working copy – you should do it ONLY via Perforce client; otherwise – you’re in quite a bit of trouble. While it is certainly *not* that big deal – but certainly an inconvenience.
- At hundreds-of-dollars-per-user - pricing can get not-so-insignificant (especially if you have part-time users).

SVN

I have to admit that for a long while, I have been a fan of SVN – and I still admire it. From a technical standpoint, SVN is your typical centralized version control system (based on a single centralized server), and is great because:

- model is simple
- it is easily usable by non-developers.
    - For non-developers on Windows (and large chunk of your designers will be on Windows) Tortoise SVN rulezzz!
- Handles large multi-terabyte projects well.
- File locking is available.
- Offline work is possible and easy (though offline commits aren’t possible, and neither is offline access to history beyond one last version)
- IMO, SVN *sync merge* is more intuitive than Git’s *rebase* (though I admit that this point is debatable and flame-war ridden).
- SVN is built under a strong perception of history being immutable. While it *is* possible to mess with SVN history, it is difficult (or even impossible?) to do without messing with SVN files directly (i.e. without having admin-level access to svn server box).
- [[TODO: partial checkouts]]
- Path-based access control is possible, including restricting reads on a per-directory/per-file basis. [[TODO: refer to explanation why it is necessary]]

On the minus side:

- While merges reportedly improved on the way towards SVN 1.8 or so – they’re still not as fluid as in Git. At the very least – as far as I know, you still SHOULD avoid renaming files in your branches (otherwise – chances are you’ll get an infamous “tree conflict” <sad-face />).[230] It is not *that* big deal – but a significant inconvenience if doing feature branching.
- Commits while you’re offline are not possible. This is not as bad as with Perforce (actually, if you’re offline just while you’re typing in while on a train back home – it is

[230] While SVN 1.10 is expected to address this problem by a significantly improved conflict resolver – SVN 1.10 is not out yet, so it is unclear whether this feature will make it to 1.10, and how exactly it will work in practice if it does make it.

not noticeable at all), but if you're going to be offline for a while as you are developing[231] – it can become a problem.

- As with Perforce, locking can be abused. To mitigate it, it is possible to:
  - Outright prohibit locking of source files (IIRC, pre-commit script should do it).
  - for non-mergeable files - make sure to write a script sending reminders (CC: PM) such as "you're holding this file since yesterday – are you sure you really need it for this long?")
- For an open-source project – SVN's model doesn't lend itself well to "pull requests"
  - OTOH, I didn't see much "pull requests" for intra-company development, even less for gamedev.

Git

Definition of git:
a foolish or worthless person
-- Merriam-Webster dictionary

After working with SVN for a while, I had to switch to Git – and found it being clearly better-suited for Feature-Branch development model; moreover – Git is also perfectly usable for Trunk-Based-Development - *as long as it is only developers who work on the repository*. On the other hand, for gamedev-with-assets-and-artists-involved – Git, while being usable, is clearly not the best option.

Pros of Git include:

- Being branch-centered from the very beginning, branch handling in Git is good.
  - Still, reverting branch merge is ugly even in Git <sad-face />, more on it in [[TODO]] section below.
- Offline work is very straightforward, you have full capabilities of commit and having your history.
  - Of course, it comes at the cost of extra *push* operation, so if you don't work offline *often* – it is not that big deal (especially in 2017, where you have Internet pretty much all the time)

List of Git negatives, at least when it is used for game development, is longer:

- Git is not really friendly to non-developers such as artists (that's to put it very mildly). The whole model is significantly more convoluted than that of centralized version control systems such as Perforce or SVN, and without ability to merge those-files-artists-are-working-on – it becomes convoluted-for-no-apparent-reason for their purposes <sad-face />.
- Whatever-you're-doing, you have to have the whole repository on your local box (except for Git-LFS files, more on them below); if the whole-project-including-history is large (as in "1T large") – it can take a looong while to download it.
- As a result, some developers have started to support multiple Git repos – one for "lean and mean" code, and another one for docs etc. TBH, I do *not* like the very

[231] Say, if your company sends you on a cruise while you're developing <wink />

idea of having several repos (it starts a slippery road towards "let's keep all the code in Git, and all the assets in SVN" – which usually qualifies as a Really Bad Idea™ because of lack of sync between two repositories, as discussed above).

- Git-LFS is a kind of crutch (and is not really following the "distributed" nature of the rest of Git).
- Handling of huge-projects-with-lots-of-binary-files is rather ugly with Git. While it did improve with Git Large File Storage (Git-LFS) – it is still not clear how Git-LFS behaves for multi-terabyte real-world projects. TODO: locking in Git-LFS 2.0 [https://github.com/git-lfs/git-lfs/wiki/File-Locking]
- File locking for non-mergeable files is not available (see below on advisory locks – but they don't really work well for non-developers).
- Per-file access control is not supported (at least not out-of-the-box). While this is not that much of a problem for open source projects – it *is* quite an issue, especially for gamedev where we have to resort to security-by-obscurity <sad-face />.
- I *positively hate* an ability to mess up with (*"amend"* in Git-speak) already-committed data. IMNSHO, having history immutable is a Really Good Thing™ for *any* version control system.

For a more detailed discussion on problems-of-Git-for-gamedev-purposes – see, for example, (chris@enemyhideout 2016).

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 (representing so-called "assets"; more on assets and asset pipeline in Vol. V's chapter on Graphics 101). More precisely, it is not only about binary files, but also includes *any* file which cannot be effectively merged by Git (even simple text-based Wavefront .obj file is not really mergeable in a sense that tracking differences in these files is pretty much useless).

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, but this is so much work that doing it for all the files your artists and game designers need, is hardly realistic <sad-face />; still – make sure that if you're using Unity, you're using their SmartMerge (that's regardless of using Git or not).

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, while locks are supported by Perforce and SVN (and there is a Lock Extension for Mercurial too) - Git's position with regards of locks is (a) that there won't be mandatory locks, ever, and (b) that advisory locks are just a way of communication so that should be done out-of-Git <sic! />. The latter statement leads to having chat channels or mailing lists just for the purposes of locking <ouch! />. I strongly disagree with such approaches, because IMNSHO:

all the stuff which is related to source-controlled code, SHOULD be within version 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 manually-editable 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, *at the very least* this logic must be wrapped into a "I want to work with this file - lock 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").[232] And if you like your artists better than that, you can make a Windows shell extension which calls this script for them and displays nice “Locked” icon.

The approach of lock-files described above is known to work (though having a drawback of creating commits just for locking purposes), but still remains quite a substantial headache. Actually, the headache can be so significant that it *might* be better to use Perforce, SVN, or Mercurial-with-Lock-Extension (all of which support mandatory locking) just for this reason.

Let's also note that there is also an issue which is often mentioned in this context, the one about storing large files in Git, but IMO this is a much more minor problem, which can be *mostly* resolved by using Git LFS plugin.

Issues with reverting Git branch merge commit

One of Git peculiarities is related to revert of Git commit of merging branches. While revert of committed branch merge is not a picnic in *any* version control system, in Git it is IMO particularly nasty and counter-intuitive.

For a proper discussion of it – take a look at (kernel.RevertFaultyMerge n.d.); here I’ll provide only a very short overview. In short – after reverting Git committed branch merge, your system is left in not exactly the same state than it was before the merge(!) – so you need to remember about this reverted merge when you’re doing re-merge, otherwise you’ll get *very* unexpected results <sad-face />. In practice, it means that with Git you generally should avoid reverting branch merges at all costs. To make things worse – in Git-world this behavior is not considered a bug-which-has-to-be-eventually-fixed (but rather a feature-which-makes-those-who-know-about-it-gurus), so chances of it being fixed are estimated about as high as chances of cold day in hell <sad-face />.

Mercurial

[232] and of course, another script "I'm done with file", which will be doing remove-lock-file-commit-and-push

Last but not least on the list of our contenders-for-version-control, is Mercurial. While Mercurial is *ideologically very similar to Git,* it certainly has a very different look and feel. In particular, the following pros can be observed about Mercurial in the context of gamedev:

- Mercurial is (almost-)usable by non-developers
    - TortoiseHG helps a lot for those poor Windows-based souls.
- Branching is good (though reverting branch merge is still a mess <sad-face />)
- Offline work is very straightforward too
- Commits are not mutable[233]
- Lock Extension is available (though not distributed with Mercurial by default)
- Per-file access control is supported (though not for reading)

List of Mercurial cons is also impressive:

- Large files (beyond 100M or so) are handled very inefficiently; Mercurial needs about file-size-multiplied-by-5x-to-10x RAM to operate, so having a 1G asset file is likely to bring quite a few systems to their knees <sad-face />.
- Same as with Git, pretty-much-whatever-you're-doing (except for large files), you have to have the whole repository on your local box.
- Just as with Git-LFS, Mercurial Large Files Extension is a crutch, going against its overall distributed nature.
- Access control restricting reading within repository is not possible <sad-face />.[234]

From what I've heard, one big reason why gamedevs are not using Mercurial, is because of that issues with large asset files (see, for example, (SirGru 2015)).

On Open-Source Gamedev

If by any (admittedly rather slim) chance you're planning to release an open-source game - another extremely-important factor is added into play: namely, "how many people you'll be able to attract to work on your open-source project?" And in this regard, GitHub is a very clear leader by far, with BitBucket and GitLab fighting for distant second place.

Now, we should observe that all these three services are running Git (only BitBucket providing Mercurial option on the side). Moreover, all the competition-running-SVN such as OSDN and Assembla are lagging far far behind these three (as for Sourceforge – it is *no longer recommended* due to certain really ugly decisions they made a few years ago (Hoffman n.d.)).

This means that

for open-source games, Git does have a Very Significant Advantage™

[233] except for "hg rollback" which doesn't go beyond one last commit

[234] And this is a fundamental restriction of all those distributed version control systems which have a copy of the whole repository on each box <sad-face />

Comparison of Four Major Version Control Systems for Gamedev Purposes

Now, we can summarize our discussion about different version control systems in the context of game development, in the following Table 11.2:

	Perforce	**SVN**	**Git**	**Mercurial**
Non-dev friendly	Excellent😃	Good😃	Poor😠	Kinda Acceptable😐
Trunk-based Development	Excellent😃	Excellent😃	Overcomplicated 😐	Good😃
Feature Branches	Poor😠	Good😃	Excellent😃	Excellent😃
Non-mergeable files	Excellent😃	Excellent😃	Afterthought, no locking😠	Afterthought, issues with large files😠
Terabyte-size projects	Excellent😃	Good😃	With Git-LFS only😠	With LargeFiles extension only 😠
Offline Work	Acceptable😐	Good😃	Excellent😃	Excellent😃
Access Control	Good😃	Good😃	Restricting read-only access is not feasible😠	Restricting read-only access is not feasible😠
CI Support	Good (Jenkins, Team City, Bamboo) 😃	Good (Jenkins, Team City, Bamboo)😃	Excellent😃 (Jenkins, Team City, Bamboo, Travis)	Good (Jenkins, Team City, Bamboo)😃
Open-Source Repositories	None I know about😠	OSDN,😐 Assembla,😐 CloudForge😐[235]	GitHub😃, Bitbucket😃, GitLab😃	Bitbucket,😃 OSDN,😐 Assembla😐

As we can see – unlike for generic software development (where Git still arguably rulezz), for gamedev we have to say that non-distributed version control systems (such as Perforce and SVN) tend to be a more logical choice than DVCS such as Git and Mercurial. When choosing between Perforce and SVN – I'd prefer SVN, but I have to admit that *if* you're heading for trunk-based development – Perforce becomes perfectly competitive too.

On the other hand, *if* your game is going to be open-source – Git (or at least Mercurial) get an all-important-for-open-source advantage of additional exposure.

[235] I don't mean OSDN, Assembla, or CloudForge are bad technically; it is just that their *popularity* (and this is what matters for crowd-source development) is lacking at this point

Version Control: 3rd-party Hosting vs In-House

In XXI century, overall trend is to have more and more services outsourced; however – while *sometimes* outsourcing is indeed a good idea, *some other times* it doesn't really work. When it comes to outsourcing version control for a game, keep in mind the following pros and cons of such outsourcing (also known as "cloud-based version control", "SaaS", etc.):

- Pro: less headaches, plain and simple. With an in-house version control, you need to spend time on configuring it, backing it up, and storing backups safely. It is not *that* much work – but somebody has to do it if you keep your system in-house.
- Pro: upgrades happen automagically, so you don't need to spend time on them
- Pro: unless you have a serious admin which handles it – it *is* less difficult to mess up your version control system.
    - BTW, I'd say that for Git requirements for your admin are *higher* than for other systems (in other words, Git is substantially easier to mismanage – in particular, due to the mutable histories <ouch! />).
- Con: for a game with lots of assets, *and* with 3rd-party hosting - you can be for a looooong wait for each checkout. Even more so if you're using Git or Mercurial (and if you're not careful enough to keep *all* your assets within Git-LFS or Mercurial LargeFiles – it can *easily* become catastrophic).
- Con: upgrades happen automagically, so you can't schedule them (so if a problem occurs affecting your system – it will happen at the worst possible time, like "on the day of the release"). Granted, it is rarely a problem for hosted version control – but *is* quite a problem for other 3rd-party systems such as Issue Tracking.
- Con: While 3rd-party hosts such as GitLab or BitBucket don't have a reason to steal your code – by their nature they are *extremely* juicy attack targets. And as we as gamedevs have to resort to "security by obscurity" much more often than we'd like to (more on it in Vol. VIII) – the damage from some-hacker-cracking-into-GitLab-or-BitBucket-and-publishing-all-the-source-code-found can be enormous.

Overall, when it comes to version control, it is not *that* much difference between 3rd-party hosting and in-house system (~="you won't do *too* wrong choosing any of these routes"). BTW, if you happen to like UI of GitLab or BitBucket but are not fond of 3rd-party hosting for any of the reasons mentioned above – keep in mind that you can have them installed in-house (a.k.a. "self hosted") too.

Version Control and 3rd-party Libraries

One subtle issue with regards to version control system is how to handle those 3rd-party libraries you're going to use. Ideally, 3rd-party libraries should be present in your version control system as links-pointing-to-specific-version of the library, with your version 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 of their project), 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 becomes *your* responsibility to update this link-to-specific-version to newer versions, at those points when you're comfortable with doing it (and re-running all of your tests using that newer-and-supposedly-better version of the library).

How to handle those 3rd-party libraries you're going to use?

`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. One potential caveat on this way is that, as Git uses whole-repository checkouts by design,[236] it means that if your 3rd-party library is large (which TBH, is rarely the case for pure libraries) – it will cause quite a bit of traffic.

Another thing to keep in mind with regards to `git submodule` is that it works only if the library is sitting within a Git repository. On the other hand, 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 (see, for example, (StackOverflow.SvnAsGitSubmodule)). A similar trick can be used with Mercurial too (see (HgGitMirror) on creating Git mirror from Mercurial).

With SVN, similar result can be achieved via *svn externals*. They do work – at least as long as all your dependencies are in SVN. Unfortunately for SVN users – most of the external-stuff-we-want-to-reuse uses Git (and linking to Git repo AFAIK will normally require import of Git into svn, <ouch />). On the other hand – fortunately for SVN users, most of the external-stuff-we-want-to-reuse resides on GitHub, and with GitHub providing SVN access to their Git repos – *svn externals* have been reported to work pretty well.

As for Perforce and Mercurial – I admittedly don't have first-hand experience with them, but from what I heard - reportedly similar things are possible both for Perforce (look for "stream imports"), and for Mercurial (via "subrepositories").

BTW, about open-source and non-open-source 3rd-party libraries: there is another (MUCH more important) issue with them, make sure to read "3rd-party Libraries: Licensing" section below.

[236] unless we're speaking about Git-LFS

Protecting Source Code

One important thing to keep in mind when implementing your version control system, is that for the vast majority of the MOGs out there – your source code, if leaked to cheaters, will make your life *much* more difficult (in extreme cases – the whole ecosystem of your game can fall apart <really-sad-face />). In particular, leaks of your Server-Side code may facilitate attacks on your Servers, and leaks of Client-Side code – will almost certainly facilitate all kinds of bots.

Eliminating this risk entirely (especially in a large organization) is not really possible (hey, even RSA got hacked in 2011 – and those guys did know their security). However, we can (and SHOULD) reduce *both* chances of the security compromise *and* effects if it happens.

Reducing Chances of Compromise: Firewalls, Antiviruses, and IDS

First of all, let's discuss things which can/should be done to reduce chances of being compromised. There is absolutely no rocket science here – and the things to do are very common.

One very obvious measure to improve your security is to have firewall in your office (yes, I know teams which don't have it). Moreover, I am arguing for having your version control server in a separate trust zone from the trust-zone-where-developers-reside. As for the other trust zones (such as DMZ for mail/web servers) – they should be configured pretty much as for any other setup (just make sure to keep all such zones miles away from trust zone for your VCS).

Another obvious thing to do is to run an antivirus on all developers' computers (including those BYOD devices which you'll allow almost-inevitably). In addition, it *might* be also a good idea to have your developers run VMs on BYOD devices to separate "home" and "work" environments (and associated security risks; BTW, it also tends to help with achieving better work-life balance, which is often badly affected by BYOD).

Besides – it is certainly a good idea (and *the must* for serious gamedev shops) to run an IDS (="Intrusion Detection System") in your office LAN (*and* within that-trust-zone-which-handles-VPN-to-access-your-VCS – this should allow you to catch infections on developer's BYOD devices). BTW, don't see IDS as a burden – it is more like "an opportunity to catch those viruses/trojan/backdoors which can cost you many months to recover from").

BYOD

https://en.wikipedia.org/wiki/Bring_your_own_device

Bring your own device (BYOD)—also called bring your own technology (BYOT), bring your own phone (BYOP), and bring your own Personal Computer (BYOPC)—refers to the policy of permitting employees to bring personally owned devices (laptops, tablets, and smart phones) to their workplace, and to use those devices to access privileged company information and applications.

Mitigating Impact from Compromises

The second vector of improving security of your source code revolves around the notion that even if security compromise does happen – you need to make sure that the negative impact is minimized.

In practice, if your team is larger than 5-10 developers or so – it is often a good (though guaranteed to be *very* unpopular) idea of restricting access of your developers only to those portions of your code which they're working with; one example would be separating your repository into "Client" (with potential further subdivisions – applying if your team is even larger - such as "Client/3D", "Client/Game Logic", "Client/UI", etc.), "Interface", and "Server" (with potential further subdivisions such as "Server/AI", "Server/Gameplay", "Server/Infrastructure").

BTW, this access restriction has nothing to do with you distrusting your developers (among other things, such a distrust can easily hurt morale – which is never ever a good thing). Instead, I am speaking about mitigating risks from an *accidental* compromise of one of developer's PCs (and these DO happen – even more so if you allow BYOD, and you usually have to). In other words – I am not arguing for having areas which are off-limits for your team (well, maybe except for cheating detection Client code[237] – but even this is arguable); rather – I am arguing for restricting *default* access to certain repositiories. In practice, I am often found arguing for a policy that

Every developer SHOULD have a right to look at any piece of code. However, obtaining access to those portions he doesn't routinely work with – SHOULD require some additional jumping through the hoops, with such jumping

[237] and automatically-generated obfuscation code – more on it in Vol. VIII's chapter on Bot Fighting

sufficient to prevent a backdoor-running-on-this-developer's-computer, from obtaining the access.

One way to do it – is to have restricted access to repositories, but provide an ability to go to the admin (better in person than over e-mail – to avoid that backdoor-sitting-on-developers-computer sending the e-mail), and request access to the specific-repository-you-want; as long as the access is requested for 2 days or so – no questions should be asked by admin. Note that to have any positive effect on security, the access must be revoked *automatically* after it expires (and there MUST NOT be a way to get access for time such as many months – at least not without appropriate justification).

Admittedly, such a policy *is* going to be a point of contention, but as soon as your team grows over 50+ people – I *strongly* advise to use it. For larger and more popular games – spearphishing your development teams (and spearphishing is notoriously difficult to avoid in 50+-member teams) is known to be quite a problem, and limiting impact of potential breaches is the least we can do to mitigate it.

Continuous Integration

One thing which was briefly mentioned above, and which you should start using as soon as possible for a pretty much 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 Vol. VII's chapter on DevOps for MOGs).

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: for each commit into the version control system, a build is automatically run – followed by 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 can be seen as an extension of a centuries-old practice of "night builds" (which are known since the times of Ancient Programmers), but instead of builds being made overnight, they're made in real-time, further improving integrity and stability of your code.

In general, while there is pretty much a consensus that Continuous Integration is almost a must-have for any serious development, there is one important peculiarity with "how to implement it".

Pre-tested Commits

Traditional (~="the old") way of doing Continuous Integration is to run build+tests automagically *right after* the commit. Then, if the build or tests fail, a red flag is raised, and then one of two things happens:

a) the commit is automatically rolled back, or
b) the developer who's committed the Bad Thing™, is notified – and "nobody has a higher priority task than fixing the build"[238].

While this was a standard way of doing things at least since "Continuous Integration" term was coined, in practice it is still far from being ideal. Option (b) frequently causes that dreaded "nobody can work until build is fixed" situations, which are outright disruptive.

And option (a), however nice it may look on paper, doesn't work well either, in particular because of the peculiarities surrounding handling of reverting-merge-commits (for discussion on Git, see Issues with reverting Git branch merge commit section above, but other VCS are also non-ideal in this department). The problem is that reverting merge commit with a subsequent automated-revert-due-to-build-failure, apparently *may* change the state of your VCS, so it is not really a no-op as we intuitively expect <ouch! />; moreover – this commit-followed-by-revert *may* carry a very substantial maintenance cost for the future.

As a result of these problems (which in turn can further lead to "commit fear" or "merge fear", which are Bad Things™ per se) – I became a strong proponent of running the build *before* the commit– and committing only *if* the build+tests are ok. Moreover, while developer running local build before committing has been a standard practice for years, it (as any other manual process) it is prone both to different configurations and to human errors, which means that *pre-commit build+tests should be automated.*

Fortunately, I am certainly not alone with this observation <wink />, and in recent years, CI tools started to support such pre-commit tests (also known as "pre-tested commits" and "gated commits"). This ability is *sooo* important, that IMNSHO it should be a clear prerequisite when you're choosing CI tools for your project (BTW, at least Jenkins and TeamCity do support pre-tested commits). On the other hand, if *really* necessary – you usually can do the same thing yourself using pre-commit hooks.[239]

3rd-party Libraries: Licensing

DISCLAIMER: I am not a laywer (not even close), and nothing in this book should be understood as legal advice; whenever in doubt (and you should always be in doubt when dealing with legal matters) - make sure to consult your lawyer.

One really important thing to remember when developing your game is that

[238] the quote is attributed to Kent Beck by (Fowler)

[239] Note that, as discussed above, post-commit scripts with revert will likely cause problems because of reverting merges peculiarities.

no 3rd-party library should ever be used without taking into account its license.

Even open-source libraries can come with all kinds of nasty-for-our-purposes 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*.[240] 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[241]- then only the Client-Side code will usually need to be published. 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

Further two things to be aware of when re-using freely-available source code, 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[242]).

On the other hand, anything which goes under BSD license, MIT license, or Apache license - can usually be used without any licensing problems (YMMV, batteries not included, and this is not a legal advice).

And of course, if you're using commercial libraries - make sure that you're complying with terms of their respective libraries too (and no, paying for the library does not necessarily mean that you are allowed to use it as you wish).

[240] in practice, it is more complicated than that, but if you want legally precise answers - you better ask your legal team

[241] distribution of Server-Side code may happen, for example, if you're selling your Server-Side as an engine, but merely running it on your Servers does not, as far as I understand, qualify as “distribution” under GPL/LGPL

[242] one exception is to rely on “fair use” doctrine, but as far as I know, code – even non-commercial one - rarely qualifies for “fair use”.

Development Process – Agile still Rulezz (regardless of how you name it)

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. In general, 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.[243]

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. 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 <smile />.

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

BTW, if you happen to consider Agile as a disease (like, for example, (Halliwell)) – that's most likely 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, as a rule of thumb, are outright detrimental for the project), I don't agree that such guys make the concepts behind agile development, less useful even by a tiny bit.

One thing which should be noted about agile criticisms (such as (Halliwell n.d.)), is that there is no real disagreement between developers 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?" While this is a perfectly valid point, there is still value in using *some* terms to describe those-things-which-we're-all-doing-anyway. 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.☺

[243] it applies to any kind of development, whether game or not

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
Do you need Product Owner (as a role)	You SHOULD. It is damn important to 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. In other words –

if you have developed an allergy to the word "agile",

just /s/agile/common sense/gi, and go ahead.

After all, it is not the *name* which matters – but rather those *practices* you're using.[244]

Which exactly are those agile/common-sense techniques worth using – depends *a lot* on the nature of your project, *and* on the people you have on your team; which means that "it is one of those things you should find out yourself".

Issue Tracking System

in any project larger than "Hello, world!" there will be bugs and other issues; this obviously applies to any kind of game too

Whether we like it or not, in any project larger than "Hello, world!" there will be bugs and other issues; this obviously applies to any kind of game too. 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.

If your game project is hosted on GitHub, *and* your team is really small (like <5 developers) – you MIGHT get away with GitHub's built-in issue tracker. If you're hosting your own version control server (or if your team is larger), you're likely to use some 3rd-party issue tracking.

The choice of the issue tracking system is usually not *that* important (="it is difficult to make *too big* mistake here"), but there are still several things to keep in mind:

- Stick to a widely-used one. This rule more or less leaves us with a choice from {Redmine|JIRA|Mantis|Trac|Bugzilla[245]|Zoho}.
- Unlike version control (which is a *developers'* tool), issue tracking is at least 50% a *management* tool (more if we throw in project management features which are often provided by issue tracking systems). As a result – make sure to ask your Project Manager about issue tracking system they prefer (hint: most of the time, they will prefer JIRA).
    - This is the point where all the Project Management features of the system come into play; TBH, as a developer I don't care about these features too much – but I do recognize their need, and am ready to deter to the Project Manager's judgement in this regard.[246]
- Support for artifacts which are used in 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

[244] well, naming can matter too – in particular, to communicate and to create buzz, but if the name "agile" starts to hurt many people in your team – I have no problems with changing the name, while keeping all the relevant practices.

[245] Yes, it is still alive and kicking

[246] If the Project Manager is bad – then the whole project is in trouble to start with, so given a choice, it is better not to work on such projects <sad-face />.

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 them into account when choosing your issue tracking system. Also keep in mind that some of the plugins which implement this functionality (even for otherwise-free systems(!)) can become pricey, so it is better to double-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 <wink />, these days issue tracking systems are pretty much about-the-same from purely issue-tracking point of view.[247]

- On the question "whether to use in-house issue tracking system or 3rd-party-hosted one" – the same reasoning applies as the one in the Version Control: 3rd-party Hosting vs In-House section above.

Issue Tracking: No Bypassing Allowed

Actually, when speaking about issue tracking from developer's point of view - there is one thing which is *much* more important than a choice of specific issue tracking system:

Whatever you do, *100%* of 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 an "each and every commit MUST mention its own issue" policy.

IMNSHO, it is perfectly normal for a Business Analyst (Game Designer, Manager, etc. etc.) to come into developer's cubicle and saying "hey, we need such and such feature, let's do it". What is *not* normal though - 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 *very firmly* against it, and *strongly* suggest to have an issue open for the feature, and to have all the relevant 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 (which very often turns out to be a prerequisite to understanding whether it is ok to change certain aspect of the feature).

Even for a team of 5 developers, it is crucial to know why each-and-every change in the code has been made, and there should be one single source of this information - your issue tracking system.

[247] I realize 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

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,[248] 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.

BTW, your *guidelines.txt* file certainly belongs to your version 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).

One last (but certainly not least) thing you should establish before you start coding, is coding guidelines for your specific 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 variable as *lpszName* – the sky won't fall, but I suggest to drop these prefixes entirely. My rationale is simple – these prefixes tend to distract from the nature of what we're doing – which in turn significantly reduces code readability.

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,[249] it won't make that much difference anyway (that is, as long as you're using it consistently across the whole project).

Project Peculiarities

[248] this book included even if you consider it reputable enough; in Vol. V's chapter on C++ there will be an example of my personal guidelines for C++, but as with any other source – don't copy it blindly

[249] personally, I don't, preferring to use *data_member_* for constructor parameters

For pretty much every project you will have some peculiarities. For example, if we are programming within (Re)Actor model as described in Vol. II's chapter on (Re)Actors, then for your Game Logic, threads will be pretty much out of question –so let's write it down into our *guidelines.txt* file (to the part which tells about Game Logic). 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,[250] it needs to be consistent for the whole project, so it also belongs to *guidelines.txt*. And so on, and so forth.

For C++, my personal set of Coding Guidelines will be discussed in Vol. V's chapter on C++, 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.[251]

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 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, which in turn requires different guidelines.

For example, even if all your code is written in C++ using (Re)Actor model, the guidelines for Infrastructure Code (the one outside of (Re)Actors) and Game Logic (implementing specific (Re)Actors) is going to be *very* different. Infrastructure Code is going to use threads (one way or another), will probably provide logging facilities so it will need to have direct file access, will probably access OS-specific services too (which makes prohibition on OS-specific stuff unlikely), etc. In contrast, Game Logic is basically going just to call whatever-is-provided-by-Infrastructure-Code (concentrating on Game Logic rather than on "how to interact with OS").

BTW, such separation is not specific to (Re)Actors. In a different example, simulation/graphics code and payment processing code, even if both are written in C++, are still going to be *very* different (and require *very* different guidelines as a result). Among other things – simulation/graphics code is traditionally *very* performance-oriented (and also having things approximated is very common); however, for the payment processing, exactly the opposite stands – for payment processing code premature optimization *is* the root of all evil, and any approximation should be banned (actually, most of the time all uses of *float* are banned outright for payment processing purposes).

[250] FWIW, my answer is 'neither – use {fmt} instead', see Vol. V's chapter on C++ for further discussion

[251] this can be 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" <wink />

As a result, I *strongly* suggest to use different guidelines for different parts 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, payment processing, and simulation code.

Enforcement and Static Analysis Tools

All the rules and guidelines are perfectly useless if nobody cares to follow them. Even if only a few people on the team ignore 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 <sad-face />.

All the rules and guidelines are perfectly useless if nobody cares to follow them

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 complains 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`".
    - Supposedly more modern alternatives to `astyle` include `uncrustify` and `clang-format`. I won't go into discussions which one is better (TBH, I have no idea myself ;-)); however, what is perfectly clear is that using *some* of them consistently across the whole project is MUCH better then using none.
- `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 real and pretty nasty 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)). Keep in mind though that quite a few of these tools are known to cause too much trouble compared to benefits-they-provide (one of common problems of static analysis tools is having too many false positives, which

in turn leads to ignoring *all* their warnings), so don't hold your breath until you tested the tool and see that it works for you.

As a rule of thumb, if you don't understand what these tools are for – well, probably you'll be able to live without them. On the other hand – if you have a person on the team who wants to spend time running them on regular basis – it is usually *very* beneficial in the long run to let her do this.

On the other hand – beware that not all the alleged violations reported by these tools need to be addressed. As such tools tend to be overzealous – they are trying to report everything and even more – and blindly trusting them in this regard won't work. As a result – take all their complaints with a grain of salt,

and do NOT hesitate to raise a question of "this complaint of this tool is stupid, let's disable it for the whole project".

Otherwise – you can easily start going along the road paved with good intentions (for more discussions on "how best practices can become witch-hunts" – see, for example, (Hare 2015)).

Bibliography

Baryshnikov, Maksim. n.d. "Engineering Decisions Behind World of Tanks Server."

Beardsley, Jason. n.d. "Seamless Servers: The Case For and Against." In *Massively Multiplayer Game Development*.

Bergsma, Remi. n.d. *Migrating an ip-address to another server: clear the arp cache of your neighbors.* https://blog.remibergsma.com/2012/11/15/migrating-an-ip-address-to-another-server-clear-the-arp-cache-of-your-neighbors/.

Bongiovanni, Francesco. n.d. http://deptinfo.unice.fr/twiki/pub/Minfo/DistributedAlgo/Cours_FailuresDetectors-Consensus-SelfStabilization.pdf.

Bray, Brandon. n.d. *The .NET Framework 4.5 includes new garbage collector enhancements for client and server apps.* https://blogs.msdn.microsoft.com/dotnet/2012/07/20/the-net-framework-4-5-includes-new-garbage-collector-enhancements-for-client-and-server-apps/.

chris@enemyhideout. 2016. *Why Git is Not Good for Games.* http://enemyhideout.com/2016/06/why-git-is-not-good-for-games/.

Corbet, Jonathan. n.d. *"NUMA scheduling progress".* https://lwn.net/Articles/568870/.

Cybersource. n.d. *"Linux vs Windows. Total Cost of Ownership Comparison".* https://static.lwn.net/images/pdf/cybersource-tco-study.pdf.

n.d. *Determining the Availability and Reliability of Storage Configurations.* http://www.dell.com/content/topics/global.aspx/power/en/ps3q02_shetty?c=us.

n.d. *DPDK.* http://dpdk.org.

Driessen, Vincent. n.d. *"A successful Git branching model".* http://nvie.com/posts/a-successful-git-branching-model/.

Duquette, Patrick. n.d. "6.2 Implementing a Seamless World Server." In *Game Programming Gems 5*.

n.d. *FAULT TOLERANT AVAILABILITY FOR CRITICAL APPLICATIONS AND VIRTUALIZED WORKLOADS.* http://www.stratus.com/solutions/platforms/ftserver/.

Fowler, Martin. n.d. "Continuous Integration." http://martinfowler.com/articles/continuousIntegration.html.

Halliwell, Luke. n.d. *"The Agile Disease".* https://lukehalliwell.wordpress.com/2008/11/16/the-agile-disease/.

Hare, 'No Bugs'. 2015. "Best Practices vs Witch Hunts." *Overload* (125).

HgGitMirror. n.d. *"Create a Git Mirror".* http://hgtip.com/tips/advanced/2009-11-09-create-a-git-mirror/.

Hoffman, Chris. n.d. *Why big open-source projects are fleeing SourceForge's free software hub.* https://www.pcworld.com/article/2938017/why-big-open-source-projects-are-fleeing-sourceforges-free-software-hub.html.

n.d. *How Fault Tolerance Works.* https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.availability.doc_41/c_plan_understand_ft.html.

IDC. n.d. *"Windows 2000 Versus Linux in Enterprise Computing".* https://www.cetic.be/IMG/pdf/TCO.pdf.

n.d. *Introduction to Receive Side Scaling.* https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/introduction-to-receive-side-scaling.

kernel.RevertFaultyMerge. n.d. https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.txt.

Klitzke, Evan. 2013. *Migrating Uber from MySQL to PostgreSQL.* https://www.yumpu.com/en/document/view/53683323/migrating-uber-from-mysql-to-postgresql.

—. 2016. *Why Uber Engineering switched from Postgres to MySQL.* https://eng.uber.com/mysql-migration/.

Lameter, Christoph. n.d. *"NUMA (Non-Uniform Memory Access): An Overview".* https://queue.acm.org/detail.cfm?id=2513149.

Lightstreamer. n.d. http://www.lightstreamer.com/.

Ligoum, Dmitry. n.d. "private communications with."

n.d. *London Stock Exchange gets the facts and dumps Windows for Linux.* http://www.itwire.com/opinion-and-analysis/the-linux-distillery/28359-london-stock-exchange-gets-the-facts-and-dumps-windows-for-linux.

n.d. *netmap - the fast packet I/O framework.* http://info.iet.unipi.it/~luigi/netmap/.

n.d. *New techniques to develop low-latency network apps.* https://channel9.msdn.com/Events/Build/BUILD2011/SAC-593T.

'No Bugs' Hare. n.d. *"Memory Leaks and Memory Leaks".* http://ithare.com/memory-leaks-and-memory-leaks/.

n.d. *NonStop (server computers).* https://en.wikipedia.org/wiki/NonStop_(server_computers).

Noyes, Katherine. n.d. *"Five Reasons Linux Beats Windows for Servers".* http://www.pcworld.com/article/204423/why_linux_beats_windows_for_servers.html.

OpenSource. n.d. *"Open Source Initiative. Licenses by Name".* https://opensource.org/licenses/alphabetical.

n.d. *Predicting the Performance of Virtual Machine Migration.* https://www.cl.cam.ac.uk/~sa497/akoush-mascots10.pdf.

Redis.CAS. n.d. http://redis.io/topics/transactions#cas.

RFG. n.d. *"TCO for Application Servers: Comparing Linux with Windows and Solaris".* http://www-03.ibm.com/linux/whitepapers/robertFrancesGroupLinuxTCOAnalysis05.pdf.

n.d. *Scaling in the Linux Networking Stack.* https://www.kernel.org/doc/Documentation/networking/scaling.txt.

n.d. *Scripting a Cisco switch with Python and Expect.* https://www.electricmonk.nl/log/2014/07/26/scripting-a-cisco-switch-with-python-and-expect/.

SirGru. 2015. *Mercurial with Largefiles Why it is not a solution for game development.* http://www.ennoble-studios.com/tuts/mercurial-with-largefiles.html.

StackOverflow.C#LambdaLoop. n.d. *"Captured variable in a loop in C#" where="StackOverflow".* http://stackoverflow.com/questions/271440/captured-variable-in-a-loop-in-c-sharp.

StackOverflow.PythonLambdaLoop. n.d. *"What do (lambda) function closures capture in Python?".* http://stackoverflow.com/questions/2295290/what-do-lambda-function-closures-capture-in-python.

StackOverflow.SvnAsGitSubmodule. n.d. *"Is it possible to have a Subversion repository as a Git submodule?".* http://stackoverflow.com/questions/465042/is-it-possible-to-have-a-subversion-repository-as-a-git-submodule.

Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli. n.d. "Architectural Breakdown of End-to-End Latency in a TCP/IP Network."

Verma, Abhishek. 2016. *Cassandra on Mesos Across Multiple Datacenters at Uber (Abhishek Verma).* C* Summit 2016. https://www.slideshare.net/DataStax/cassandra-on-mesos-across-multiple-datacenters-at-uber-abhishek-verma-c-summit-2016.

Wikipedia. 2017. *Cluster Launch Failure.* https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure.

Wikipedia.StaticCodeAnalysis.Tools. n.d. *"List of tools for static code analysis".* https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis" where="Wikipedia.

Zubek, Robert. n.d. *"Engineering Scalable Social Games".* http://gdcvault.com/play/1012230/Engineering-Scalable-Social.

—. n.d. *"Private communications with".*