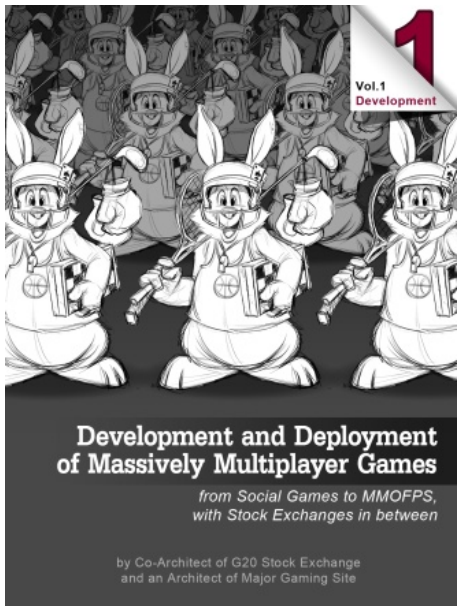
Chapter VI(a). Server-Side MMO Architecture. Naïve, Web-Based, and Classical Deployment Architectures

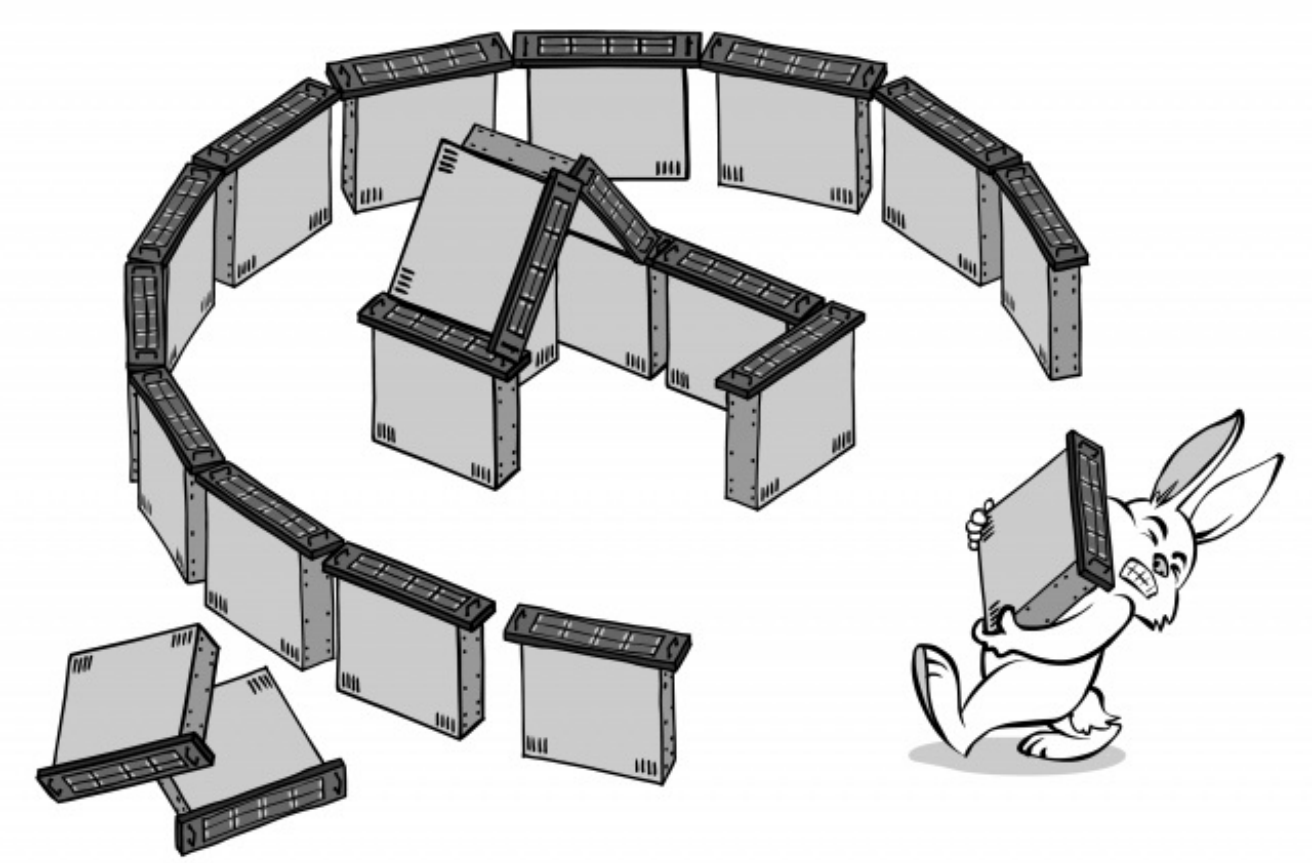
posted December 21, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

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

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



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



Deployment Architectures, Take 1

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

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

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

Don't Do It: Naïve Game Deployment Architectures

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

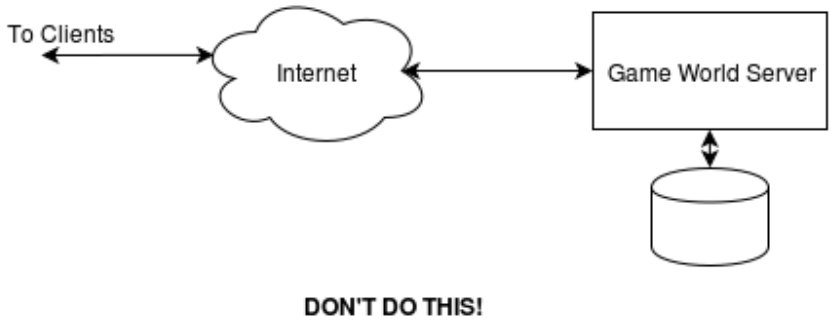


Fig VI.1

It is dead simple: there is a server, and there is a database to store persistent state. And later on, as one single Game World server proves to be insufficient, it naturally evolves into something like the diagram on Fig VI.2:

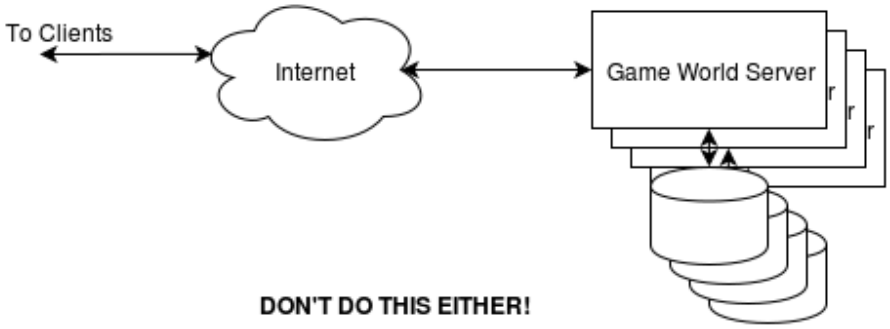


Fig VI.2

with each of Game World servers having its own database.

My word of advice about such naïve deployment architectures:

DON'T DO THIS!

Such a naïve approach won't work well for a vast majority of games. The problem here (usually ranging from near-fatal to absolutely-fatal depending on specifics of your game) is that this architecture doesn't allow for interaction between players coming from different servers. In particular, such an architecture becomes absolutely deadly if your game allows some way for a player to choose who he's playing with (or if you have some kind of merit-based tournament system), in other words – if you're *not* allowed to arbitrary separate your players (and in most cases you will need some kind of interaction at least because of the social network integration, see Chapter II for further discussion in this regard).

CSR
Customer service representatives interact with customers to

For the naïve architecture shown on Fig VI.2, any interaction between separate players coming from separate databases, leads to huge mortgage-crisis-size problems. Inter-DB interaction, while possible (and we'll discuss it in Chapter [[TODO]]) won't work well around these lines and between completely independent databases. You're going to have lots and lots of problems, ranging from delays due to improperly implemented inter-DB transactions (apparently this is not

provide answers to inquiries involving a company's product or services.
— Wikipedia —

that easy), to your CSRs going crazy because of two different users having the same ID in different databases. Moreover, if you start like this, you will even have trouble merging the databases later (the very first problem you will face will be about collisions in user names between different DBs, with much more to follow).

To summarize relevant discussion from Chapter II and from present Chapter:

A. You WILL need inter-player interaction between arbitrary players. If not now, then later. B. Hence, you SHOULD NOT use "naïve" architecture shown above.

Fortunately, there are relatively simple and practical architectures which allow to avoid problems typical for naïve approaches shown above.

Web-Based Game Deployment Architecture

If your game satisfies two conditions:

- first, it is reeeeallyyyy sloooow-paaaaaced (in other words, it is not an MMOFPS and even not a poker game) and/or "asynchronous" (as defined in Chapter I, i.e. it doesn't need players to be present simultaneously),
- *and* second, it has little interaction between players (think farming-like games with only occasional inter-player interaction),

then you might be able to get away with Web-Based server-side architecture, shown on Fig VI.3:

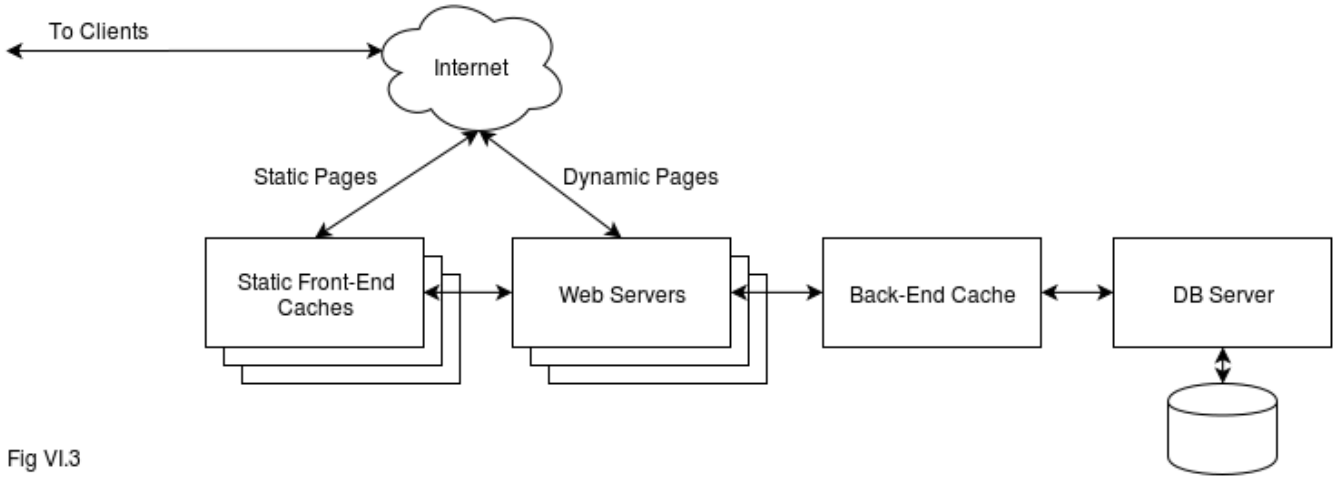


Fig VI.3

Web-Based Deployment Architecture: How It Works

The whole thing looks alongside the lines of a heavily-loaded web app – with lots of caching, both at front-end (to cache pages), and at a back-end. However, there are also significant differences (special thanks to Robert Zubek for sharing his experiences in this regard, [Zubek2016]).

The question "which web server to use" is not *that* important here. On the other hand, there exists an interesting and not-so-well-known web server, which took an extra mile to improve communications in game-like environments. I'm speaking about [Lightstreamer]. I didn't try it myself, so I cannot vouch for it, but what they're doing with regards to improving interactivity over TCP, is really interesting. We'll discuss some of their tricks in Chapter [{TODO]].

Peculiarities in Web-Based Game architectures are mostly about the way caching is built. First, on Fig VI.3 both front-end caching and back-end caching is used. Front-end caching is your usual page caching (like nginx in reverse-proxy mode, or even a CDN), though there is a caveat. As your current-game-data changes very frequently, you normally don't want to cache it, so you need to take an effort and clearly separate your static assets (.SWFs, CSS, JS, etc. etc.) which can (and should) be cached, and dynamic pages (or AJAX) with current game state data which changes too frequently to bother about caching it (and which will likely go directly from your web servers) [Zubek2010].

At the back-end, the situation is significantly more complicated. According to [Zubek2016], for games you will often want not only to use your back-end cache as a cache to reduce number of DB reads, but also will want to make it a write-back cache (!), to reduce the number of DB writes. Such a write-back cache can be implemented either manually over memcached (with web servers writing to memcached only, and a separate daemon writing 'dirty' pages from memcached to DB), or a product such as Redis or Couchbase (formerly Membase) can be used [Zubek2016].

CAS

Compare-And-Swap is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of

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

"One Big

One Big Advantage of having write-back cache (and of the in-memory state of Classical deployment architecture described below) is related to the huge reduction in number of DB updates. For example, if we'd need to save each and every click on the simulated farm with

Advantage of having write-back cache (and of the in-memory state of Classical deployment architecture described below) is related to the huge reduction in number of DB updates.

> **that memory location to a given new value.**
>
> *— Wikipedia —*

25M daily users (each coming twice a day and doing 50 modifying-farm-state clicks each time in a 5-minute session), we could easily end up with 2.5 billion DB transactions/day (which is infeasible, or at least non-affordable). On the other hand, if we're keeping write-back cache, we can write the cache into DB only once per 10 minutes, we'd reduce the number of DB transactions 50-fold, bringing it to much more manageable 50 million/day.

For faster-paced games (usually implemented as a Classical Architecture described below, but facing the same challenge of DB being overloaded), the problem surfaces even earlier. For example, to write each and every movement of every character in an MMORPG, we'd have a flow of updates of the order of 10 DB-transactions/sec/player (i.e. for 10'000 simultaneous players we'd have 100'000 DB transactions/second, or around 10 billion DB transactions/day, once again making it infeasible, or at the very least non-affordable). On the other hand, with in-memory states stored in-memory-only (and saving to DB only major events such as changing zones, or obtaining level) – we can reduce the number of DB transactions by 3-4 orders of magnitude, bringing it down to much more manageable 1M-10M transactions/day.

As an additional benefit, such write-back caches (as long as you control write times yourself) and in-memory states also tend to play well with handling server failures. In short: for multi-player games, if you disrupt a multi-player "game event" (such as match, hand, or fight) for more than a few seconds, you won't be able to continue it anyway because you won't be able to get all of your players back; therefore, you'll need to roll your "game event" back, and in-memory states provide a very natural way of doing it. See "Failure Modes & Effects" section below for detailed discussion of failure modes under Classical Game Architecture.

A word of caution for stock exchanges. If your game is a stock exchange, you generally do need to save everything in DB (to ensure strict correctness even in case of Game Server loss), so in-memory-only states are not an option, and DB savings do not apply. However, even for stock exchanges at least Classical Game architecture described below has been observed to work very well despite DB transaction numbers being rather large; on the other hand, for stock exchanges transaction numbers are usually not that high as for MMORPG, and price of the hardware is generally less of a problem than for other types of games.

Write-Back Caches: Locking

As always, having a write-back cache has some very serious implications, and will

cause lots of problems whenever two of your players try to interact with the same cached object. To deal with it, there are three main approaches: "optimistic locking", "pessimistic locking", and transactions. Let's consider them one by one.

Optimistic Locking. This one is directly based on memcached's CAS operation.[1] The idea of using CAS for optimistic locking goes along the following lines. To process some incoming request, Web Server does the following:

- reads whole "game world" state as a single blob from memcached, alongside with "cas token". "cas token" is a thing which is actually a "version number" for this object.
- we're optimists! 🙂 so Web Server is processing incoming request ignoring possibility that some other Web Server also got the same "game world" and is working on it
  - Web Server is NOT allowed to send any kind of reply back to user (yet)
- Web Server issues cas operation with both new-value-of-"game-world"-blob, and the same "cas token" which it has received
  - if "cas token" is still valid (i.e. nobody has written to the blob before current Web Server has read it), memcached writes new value, and returns ok.
    - Then our Web Server may send reply back to whoever-requested-it
  - if, however, there was a second Web Server which has managed to write after we've read our blob – memcached will return a special error
    - in this case, our Web Server MUST discard all the prepared replies
    - in addition, it MAY read new value of "game world" state (with new "cas token"), and try to re-apply incoming request to it
      - this is perfectly valid: it is just "as if" incoming request has came a little bit later (which can always happen)

Optimistic locking is simple, is lock-less (which is important, see below why), and has only one significant drawback for our purposes. That is, while it works fine as long as collision probability (i.e. two Web Servers working on the same "game world" at the same time) is low, but as soon as probability grows (beyond, say 10%) – you will start getting a significant performance hit (for processing the same message twice, three times, and so on and so forth). For slow-paced asynchronous games it is very unlikely to become a problem, and therefore by default I'd recommend optimistic locking for web-based games, but you still need to understand limitations of the technology before using it.

---

[1] a supposedly equivalent optimistic locking for Redis is described in [Redis.CAS]

Pessimistic Locking. This is pretty much a classical multi-threaded mutex-based locking, applied to our "how to handle two concurrent actions from two different Web Servers over the same "game world" problem.

In this case, game state (usually stored as a whole in a blob) is protected by a sorta-mutex (so that two web servers cannot access it concurrently). Such a mutex can be implemented, for example, over something like memcached's CAS operation [Zubek2010]. For pessimistic locking, Web Server acts as follows:

- obtains lock on mutex, associated with our "game world" (we're pessimists ☹ , so we need to be 100% sure before processing, that we're not processing in vain).
  - if mutex cannot be obtained – Web Server MAY try again after waiting a bit
- reads "game world" state blob
- processes it
- writes "game world" state blob
- releases lock on mutex

This is a classical mutex-based schema and it is very robust when applied to classical multi-thread synchronization. However, when applying it to web servers and memcached, there is a pretty bad caveat ☹ . The problem here is related to "how to detect hanged/crashed web server – or process – which didn't remove the lock" question, as such a lock will effectively prevent all future legitimate interactions with the locked game world (which reminds me of the nasty problems from the early-90ish pre-SQL FoxPro-like file-lock-based databases).

For practical purposes, such a problem can be resolved via timeouts, effectively breaking the lock on mutex (so that if original mutex owner of the broken mutex comes later, he just gets an error). However, allowing to break mutex locks on timeouts, in turn, has significant further implications, which are not typical for usual mutex-based inter-thread synchronizations:

- first, if we're breaking mutex on timeout – there is a problem of choosing the timeout. Have it too low, and we can end up with fake timeouts, and having it too high will cause frustrated users
- second, it implies that we're working EXACTLY according to the pattern above. In particular:
  - having more than one memcached object per "game world" is not allowed
  - "partially correct" writes of "game state" are not allowed either, even if they're intended to be replaced "very soon" under the same lock

In practice, these issues are rarely causing too much problems when using memcached for mutex-based pessimistic locking. On the other hand, as for memcached we'd need to simulate mutex over CAS, I still suggest optimistic locking (just because it is simpler and causes less memcached interactions).

Transactions. Classical DB transactions are useful, but dealing with concurrent transactions is really messy. All those transaction isolation levels (with interpretations subtly different across different databases), locks, and deadlocks are not a thing which you really want to think about.

Fortunately, Redis transactions are completely unlike classical DB transactions and are coming without all this burden. In fact, Redis transaction is merely a sequence of operations which are executed atomically. It means no locking, and an ability to split your "game world" state into several parts to deal with traffic. On the other hand, I'd rather suggest to stay away from this additional complexity as long as possible, using Redis transactions only as means of optimistic locking as described in [Redis.CAS]. Another way of utilizing capabilities of Redis transactions is briefly mentioned in "Web-Based Deployment Architecture: FSMs" section below.

Web-Based Deployment Architecture: FSMs

You may ask: how finite state machines (FSMs) can possibly be related to the web-based stuff? They seem to be different as night and day, don't they?

Actually, they're not. Let's take a look at both optimistic and pessimistic locking above. Both are taking the whole state, generating new state out of it, and storing this new state. But this is exactly what our FSM::process_event() function from Chapter V does! In other words, even for web-based architecture, we can (and IMHO SHOULD) write processing in an event-driven manner, taking state and processing inputs, producing state and issuing replies as a result.

As soon as we've done it this way, the question "Should we use optimistic locking or pessimistic one", becomes a deployment implementation detail

In other words, if we have an FSM-based (a.k.a. event-driven) game code, we can change the wrapping infrastructure code around it, and switch it from optimistic locking to pessimistic one (or vice versa). All this without changing a single line *within* any of FSMs!

Moreover, if using FSMs, we can even change from Web-Based Architecture to Classical one and vice versa without changing FSM code

If by any chance reading the whole "game world" state from cache becomes a problem (which it shouldn't, but you never know), it MIGHT still be solved via FSMs together with Redis-style transactions mentioned above. Infrastructure code (the one outside of FSM) may, for example, load only a part of the "game world" state depending on type of input request (while locking all the other parts of the state to avoid synchronization problems), and also MAY implement some kind on-demand exception-based state loading along the lines of on-demand input loading discussed in [[TODO]] section below.

Web-Based Deployment Architecture: Merits

Unlike the naïve approach above, Web-Based systems may work. Their obvious advantage (especially if you have a bunch of experienced web developers on your team) is that it uses familiar and readily-available technologies. Other benefits are also available, such as:

- easy-to-find developers
- simplicity and being relatively obvious (that is, until you need to deal with locks, see above)
- web servers are stateless (except for caching, see below), so failure analysis is trivial: if one of your web servers goes down, it can be simply replaced
- can be easily used both for the games with downloadable client and for browser-based ones

Web-Based Architecture (as well as any other one), of course, also has downsides, though they may or may not matter depending on your game:

- there is no way out of web-based architecture; once you're in – switching to any other one will be impossible. Might be not that important for you, but keep it in mind.
- it is pretty much HTTP-only (with an option to use Websockets); migration to plain TCP/UDP is generally not feasible.
- as everything will work via operations on the whole game state, different parts of your game will tend to be tightly coupled. Not a big problem if your game is trivial, but may start to bite as complexity grows.
- as the number of interactions between players and game world grows, Web-Based Architecture becomes less and less efficient (as distributed-mutex-locked accesses to retrieve whole game state from the back-end cache and write it back as a whole, don't scale well). Even medium-paced "synchronous" games such as casino multi-players, are usually not good candidates for Web-Based Architecture.
- you need to remember to keep all the accesses to game objects synchronized;

if you miss one – it will work for a while, but will cause very strange-looking bugs under heavier load.

- you'll need to spend A LOT of time meditating over your caching strategy. As the number of player grows, you're very likely to need a LOT of caching, so start designing your caching strategies ASAP. See above about peculiarities of caching when applied to games (especially on write-back part and mutexes), and make your own research.
- as the load grows, you will be forced to spend time on finding a good and really-working-for-you solution for that nasty web-server-never-releases-mutex problem mentioned above. While not that hopeless as ensuring consistency within pre-SQL DBF-like file-lock-based databases, expect quite a chunk of trouble until you get it right.

Still,

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

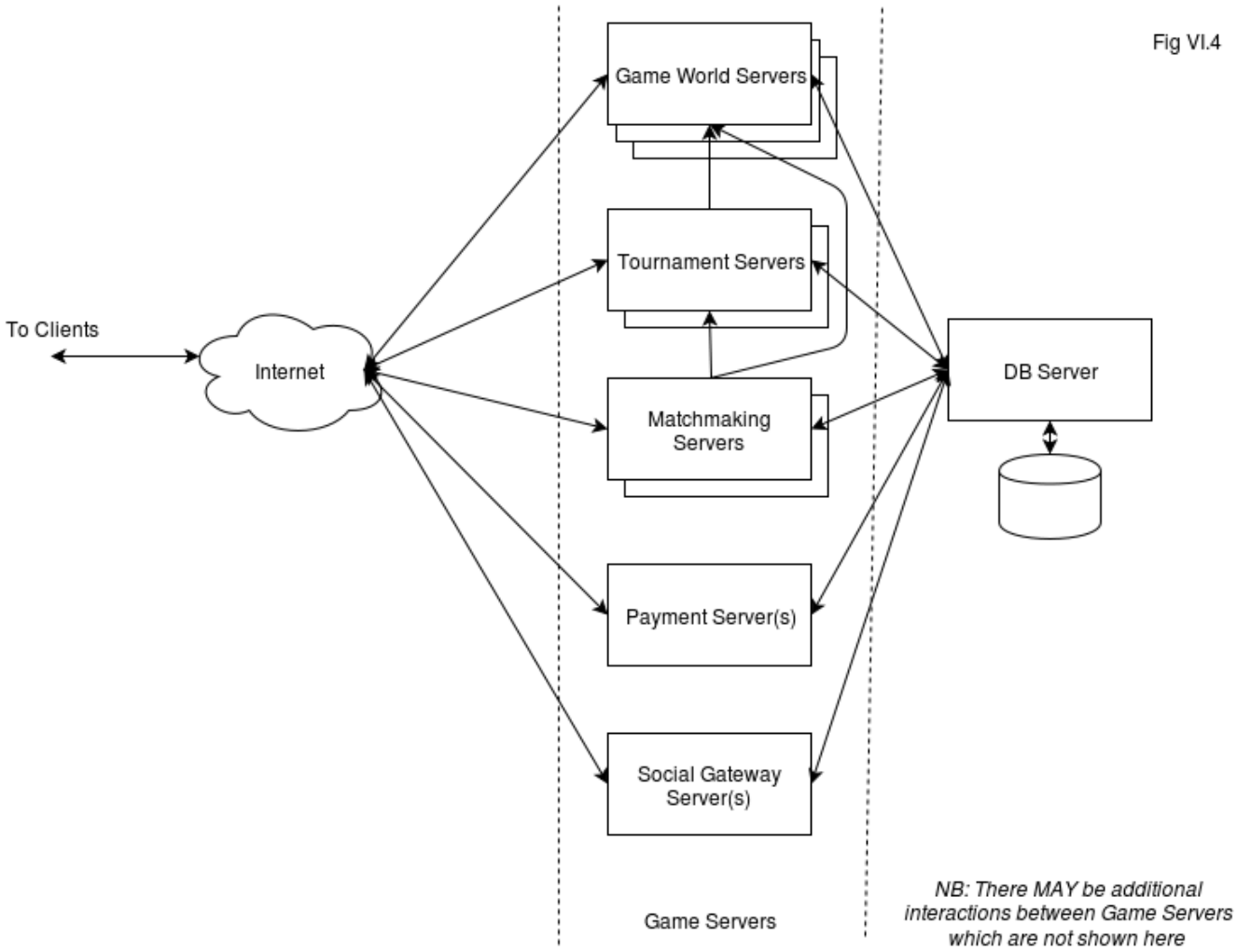
While Classical Architecture described below (especially with Front-End Servers added, see [[TODO]] section) can also be used for slow-paced games, implementing it yourself just for this purpose is a Really Big Headache and might be easily not worth the trouble if you can get away with Web-Based one. On the other hand,

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

(see above).

Classical Game Deployment Architecture

Fig VI.4 shows a classical game deployment diagram.



In this deployment architecture, clients are connected to Game Servers directly, and Game Servers are connected to a single DB Server, which hosts system-wide persistent state. Each of Game Servers MIGHT (or might not) have it own database (or other persistent storage) depending on the needs of your specific game; however, usually Game Servers store only in-memory states with all the persistent storage going into a single DB residing on DB Server.

Game Servers

Game Servers are traditionally divided according to their functionality, and while you can combine different types of functionality on the same box, there are often good reasons to avoid combining too many different things together.

Different types of Game Servers (more strictly – different types of functionality hosted on Game Servers) should be mapped to the entities on your Entities&Relationships Diagram described in Chapter II. You should do this mapping for your specific game yourself. However, as an example, let's take a look at a few of *typical* Game Servers (while as always, YMMV, these are likely to be present for quite a few games):

Game World Servers. Your game worlds are running on Game World Servers, plain and simple. Note that "Game World" here doesn't necessarily mean a "3D

game world with simulated physics etc.". Taking a page from a casino-like games book, "Game World" can be a casino table; going even further into realm of stock exchanges, "Game World" may be a stock exchange floor. Surprisingly, from an architecture point of view, all these seemingly different things are very similar. All of them represent a certain state (we usually name it "game world") which is affected by player's actions in real time, and changes to this state are shown to all the players.[2]

"Usually, when a player launches her client app, the client by default connects to one of Matchmaking Servers.

Matchmaking Servers. Usually, when a player launches her client app, the client by default connects to one of Matchmaking Servers. In general, matchmaking servers are responsible for redirecting players to one of your multiple game worlds. In practice, they can be pretty much anything: from lobbies where players can join teams or select game worlds, to completely automated matchmaking. Usually it is matchmaking servers that are responsible for creating new game worlds, and placing them on the servers (and sometimes even creating new servers in cloud environments).

Tournament Servers. Not always, but quite often your game will include certain types of "tournaments", which can be defined as game-related entities that have their own life span and may create multiple Game World instances during this life span. Technically, these are usually reminiscent of Matchmaking Servers (they need to communicate with players, they need to create Game Worlds, they tend to use about the same generic protocol synchronization mechanics, see Chapter [[TODO]] for details), but of course, Tournament Servers need to implement tournament rules of the specific tournament etc. etc.

Payment Server and Social Gateway Server. These are necessary to provide interaction of your game with the real world. While these server might look an "optional thing nobody should care about", they're usually playing an all-important role in increasing popularity of your game and monetization, so you'd better to account for them from the very beginning.

"Payment Server and

The very nature of Payment Servers and Social Gateway Server is to be "gateways to the real world", so they're usually exactly what is written on the tin: gateways. It means that their primary function is usually to get some kind of input from the player and/or other Game Servers, write something to DB (via DB Server), and make some request according to some-external-protocol (defined by payment provider or by social network). On the other hand, implementing them when you need to support multiple payment/social providers (each

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

with their own peculiarities, you can count on it) – is a challenge; also they tend to change a lot due to requirements coming from business and marketing, changes in provider's APIs, need to support new providers etc. And of course, at least for payment servers, there are questions of distributed transactions between your DB and payment-provider DB, with all the associated issues of recovery from "unknown-state" transactions, and semi-manul reconciliation of reports at the end of month. As a result, these two seemingly irrelevant-to-gameplay servers tend to have their own teams after deployment; more details on payment servers will be discussed in Chapter [[TODO]].

One of the things these servers should do, is isolating Game World Servers and preferably Matchmaking Servers from the intimate details about specifics of the payment providers and social networks. In other words, Game World Servers shouldn't generally know about such things as "a guy has made a post of Facebook, so we need to give him bonus of 25% extra experience for 2 days". Instead, this functionality should be split in two: Social Gateway Server should say "this guy has earned bonus X" (with explanation in DB why he's got the bonus, for audit purposes), and Game World Server should take "this guy has bonus X" statement and translate it into 25% extra experience.

---

[2] restrictions may apply to which parts of the state are shown to which players. One such example is a server-side fog-of-war, that we'll discuss in Chapter [[TODO]]

Implementing Game Servers under QnFSM architecture

In theory, Game Servers can be implemented in whatever way you prefer. In practice, however, I strongly suggest to have them implemented under Queues-and-FSMs (QnFSM) model described in Chapter V. Among the other things, QnFSM provides very clean separation between different modules, enables replay-based debug and production post-mortem, allows for different deployment scenarios without changing the FSM code (this one becomes quite important for the server side), and completely avoids all those pesky inter-thread synchronization problems at logical level; see Chapter V for further discussion of QnFSM benefits.

Fig VI.5 shows a diagram with an implementation of a generic Game Server under QnFSM:

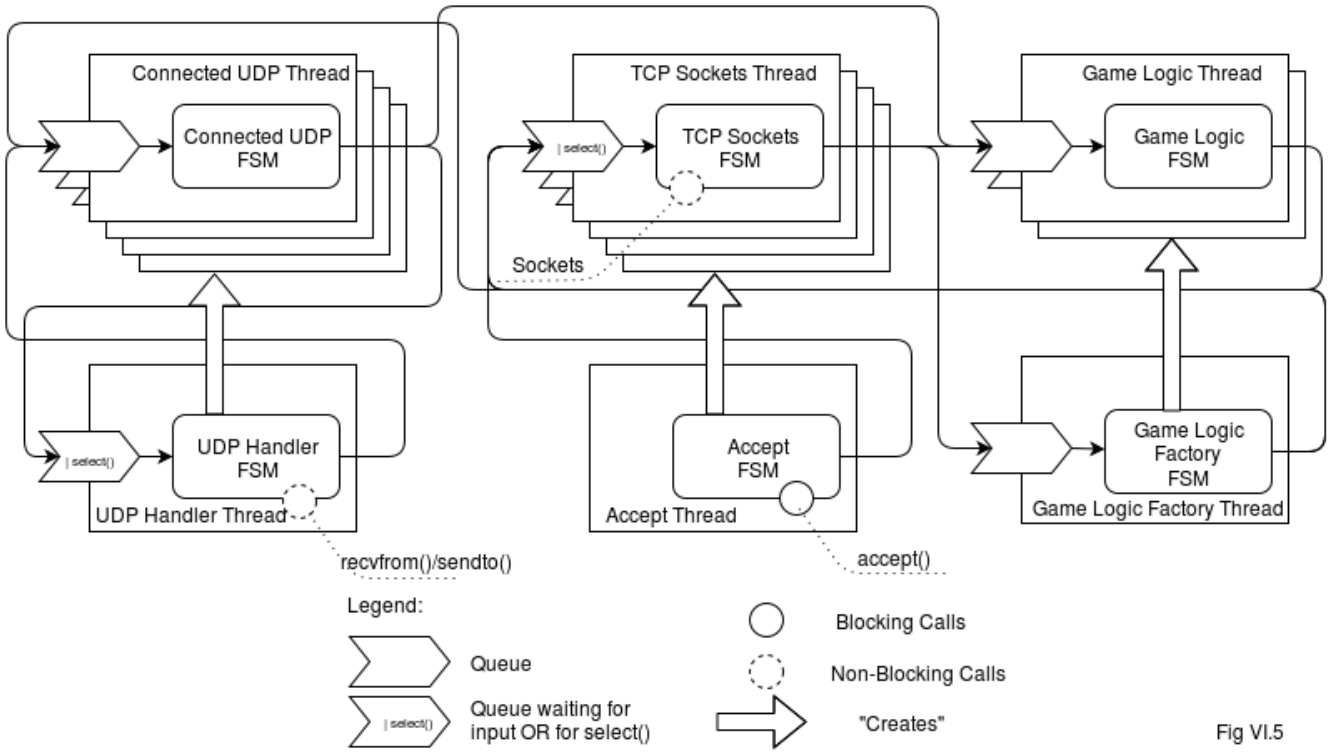


Fig VI.5

If it looks complicated at the first glance – well, it should. First of all, the diagram represents quite a generic case, and for your specific game (and at least at first stages) you may not need all of that stuff, we'll discuss it below. Second, but certainly not unimportant, writing anywhere-close-to-scalable server is not easy.

Now let's take a closer look at the diagram on Fig VI.5, going in an unusual direction from right to left.

Game Logic and Game Logic Factory. On the rightmost side of the diagram, there is the most interesting part – things, closely related to your game logic. Specifics of those Game Logic FSMs are different for different Game Servers you have, and can vary from "Game World FSM" to "Payment Processing FSM" with anything else you need in between. It is worth noting that while for most Game Logic FSMs you won't need any communications with the outside world except for sending/receiving messages (as shown on the diagram), for gateway-style FSMs (such as Payment FSM or Social Gateway FSM) you will need some kind of external API (most of the time they go over outgoing HTTP, though I've seen quite strange things, such as X.25); it doesn't change the nature of those gateway-style FSMs, so you still have all the FSM goodies (as long as you "intercept" all the calls to that external API, see Chapter V for details). [[TODO! – discussion on blocking-vs-non-blocking APIs for gateway-style FSMs]]

"When a Matchmaking server needs to create a new game world on server X, it sends a request to the Game Logic

Game Logic Factory is necessary to create new FSMs (and if

necessary, new threads) by an external request. For example, when a Matchmaking server needs to create a new game world on server X, it sends a request to the Game Logic Factory which resides on server X, and Game Logic Factory creates game world with requested parameters. Deployment-wise, usually there is only one instance of the Game Logic Factory per server, but technically there is no such strict requirement.

Factory which resides on server X, and Game Logic Factory creates game world with requested parameters.

TCP Sockets and TCP Accept. Going to the left of Game Logic on Fig VI.5, we can see TCP-related stuff. Here the things are relatively simple: we have classical accept() thread, that passes the accepted sockets to Socket Threads (creating Socket Threads when it becomes necessary).

The only really important thing to be noted here is that each Socket Thread[3] should normally handle more than one TCP socket; usually number of TCP sockets per thread for a game server should be somewhere between 16 and 128 (or "somewhere between 10 and 100" if you prefer decimal notation to hex). On Windows, if you're using WaitForMultipleObjects()[4], you're likely to hit the wall at around 30 sockets per thread (see further discussion in Chapter [[TODO]]), and this has been observed to work perfectly fine. Having one thread (even worse – two, one for recv() and another one for send()) per socket on the server-side is generally not advisable, as threads have substantial associated overhead (both in terms of resources, and in terms of context switches). In theory, multiple sockets per thread may cause additional latencies and jitter, but in practice for a reasonably well written code running on a non-overloaded server I wouldn't expect additional latencies and jitter of more than single-digit microseconds, which should be non-observable even for the most fast-paced games.

---

[3] and accordingly, Socket FSM, unless you're hosting multiple Socket FSMs per Socket Thread, which is also possible

[4] which IMHO provides the best balance between performance and implementation complexity (that is, if you need to run your servers on Windows), see Chapter [[TODO]] for further details

UDP-related FSMs. UDP (shown on the left side of Fig VI.5) is quite a weird beast; in some cases, you can use really simple things to get UDP working, but in some other cases (especially when high performance is involved), you may need to resort to quite heavy solutions to achieve scalability. The solution on Fig VI.5 is on the simpler side, so you MIGHT need to get into more complicated things to achieve performance/scalability (see below).

Let's start explaining things here. One problem which you [almost?] universally will

have when using UDP, is that you will need to know whether your player is connected or not. And as soon as you have a concept of "UDP connection" (for example, provided by your "reliable UDP" library), you have some kind of connection state/context that needs to be stored somewhere. This is where those "Connected UDP Threads" come in.

KISS principle

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

— Wikipedia —

So, as soon as we have the concept of "player connected to our server" (and we need this concept at least because players need to be subscribed to the updates from our server), we need those "Connected UDP Threads". Not exactly the best start from KISS point of view, but at least we know what we need them for. As for the number of those threads – we should limit the number of UDP connections per Connected UDP Thread; as a starting point, we can use the same ballpark numbers of UDP connections per thread as we were using for TCP sockets per thread: that is, between 16-128 UDP connections per thread.

UDP Handler Thread and FSM is a very simple thing – it merely gets whatever-comes-in-from-recvfrom(), and passes it to an appropriate Connected UDP Thread (as UDP Handler FSM also creates those Connected UDP Threads, it is not a problem for it to have a map of incoming-packet-IP/port-pairs to threads).

"You MAY find that your UDP Handler Thread becomes a bottleneck, causing incoming packets to drop

However, you MAY find that this simpler approach doesn't work for you (and your UDP Handler Thread becomes a bottleneck, causing incoming packets to drop while your server is not overloaded yet); in this case, you'll need to use platform-specific stuff such as recvmmsg(),[5] or to use multiple recvfrom()/sendto() threads. The latter multi-threaded approach will in turn cause a question "where to store this mapping of incoming-packet-IP/port-pairs to threads". This can be addressed either using shared state (which is a deviation from pure FSM model, but in this particular case it won't cause too much trouble in practice), or via separate UDP Factory Thread/FSM (with UDP Factory FSM storing the mapping, and notifying recvfrom() threads about the mapping on request, in a manner somewhat similar to the one used for Routing Factory FSM described in [[TODO]] section below).

[5] see further discussion on recvmmsg() in Chapter [[TODO]]

Websocket-related FSMs and HTTP-related FSMs (not shown). If you need to

support Websocket clients (or, Stevens forbid, HTTP clients) in addition to, or instead of TCP or UDP, this can be implemented quite easily. Basic Websocket protocol is very simple (with basic HTTP being even simpler), so you can use pretty much the same FSMs as for TCP, but implementing additional header parsing and frame logic within your Websocket FSMs. If you think you need to support HTTP protocol for a synchronous game – think again, as implementing interactive communications over request-response HTTP is difficult (and tends to cause too much server load), so Websockets are generally preferable over HTTP for synchronous games and are providing about-the-same (though not identical) benefits in terms of browser support and being firewall friendly; see further discussion on these protocols in Chapter [[TODO]]. For asynchronous games, HTTP (with simple polling) MAY be a reasonable choice.

CUDA/OpenCL/Phi FSM (not shown). If your Game Worlds require simulation which is very computationally heavy, you may want to use your Game World servers with CUDA (or OpenCL/Phi) hardware, and to add another FSM (not shown on Fig VI.5) to communicate with CUDA/OpenCL/Phi GPGPU. A few things to note in this regard:

- We won't discuss how to apply CUDA/OpenCL/Phi to your simulation; this is your game and a question "how to use massively parallel computations for your specific simulation" is utterly out of scope of the present book.
- Obtaining strict determinism for CUDA/OpenCL FSMs is not trivial due to potential inter-thread interactions which may, for example, change the order of floating-point additions which may lead to rounding-related differences in the last digit (with both results practically the same, but technically different). However, for most of gaming purposes (except for replaying server-side simulation forever-and-ever on all the clients), even this "almost-strict-determinism" may be sufficient. For example, for "recovery via replay" feature discussed in "Complete Recovery from Game World server failures: DIY Fault-Tolerance in QnFSM World" section below, results during replay-since-last-state-snapshot, while not guaranteed to be *exactly* the same, are not too likely to result in macroscopic changes which are too visible to players.
- Normally, you're not going to ship your game servers to your datacenter. Well, if the life of your game depends on it, you might, but this is a huuuge headache (see below, as well as Chapter [[TODO]] for further discussion)
    - As soon as you agree that it is not your servers, but leased ones or cloud ones (see also Chapter [[TODO]]), it means that you're completely dependent on your server ISP/CSP on supporting whatever you need.

CSP
Cloud Service Provider

    - Most likely, with 3rd-party ISP/CSP it will be Tesla or GRID GPU (both by NVidia), so in this case you should be ok with CUDA rather than OpenCL.
    - The choice of such ISPs which can lease you GPUs, is limited, and they

tend to be on an expensive side :-(. As of the end of 2015, the best I was able to find was Tesla K80 GPU (the one with 4992 cores) rented at $500/month (up to two K80's per server, with the server itself going at $750/month). With cloud-based GPUs, things weren't any better, and started from around $350/month for a GRID K340 (the one with 4×384=1536 total cores). Ouch!

- If you *are* going to co-locate your servers instead of leasing them from ISP[6], you should still realize that server-oriented NVidia Tesla GPUs (as well as AMD FirePro S designated for servers) are damn expensive. For example, as of the end of 2015, Tesla K80 costs around $4000(!); at this price, you get 2xGK210 cores, 24GB RAM@5GHz, clock of 562/875MHz, and 4992 CUDA cores. At the same time, desktop-class GeForce Titan X is available for about $1100, has 2 of newer GM200 cores, 12GB RAM@7GHz, clock of 1002/1089MHz, and 3072 CUDA cores. In short – Titan X gets you more or less comparable performance parameters (except for RAM size and double-precision calculations) at less than 30% of the price of Tesla K80. It might look as a no-brainer to use desktop-class GPUs, but there are several significant things to keep in mind:
    - the numbers above are *not* directly comparable; make sure to test your specific simulation with different cards before making a decision. In particular, differences due to RAM size a double-precision maths can be very nasty depending on specifics of your code

"In short – Titan X gets you more or less comparable performance parameters (except for RAM size and double-precision calculations) at less than 30% of the price of Tesla K80.

    - even if you're assembling your servers yourself, you are still going to place your servers into a 3rd-party datacenter; hosting stuff within your office is not an option (see Chapter [[TODO]])
        - space in datacenters costs, and costs a lot. It means that tower servers, even if allowed, are damn expensive. In turn, it usually means that you need a "rack" server.
        - Usually, you cannot just push a desktop-class GPU card (especially a card such as Titan X) into your usual 1U/2U "rack" server; even if it fits physically, in most cases it won't be able to run properly because of overheating. Feel free to try, and *maybe* you will find the card which runs ok, but don't expect it to be the-latest-greatest one; thermal conditions within "rack" servers are extremely tight, and air flows are traditionally very different from the desktop servers, so throwing in additional 250W or so with a desktop-oriented air flow to a non-GPU-optimized server isn't likely to work for more than a few minutes.

- IMHO, your best bet would be to buy rack servers which are specially designated as "GPU-optimized", *and ideally – explicitly supporting those GPUs that you're going to use.* Examples of rack-servers-supporting-desktop-class-GPUs range from[7] 1U server by Supermicro with up 4x Titan X cards,[8] to 4U boxes with up to 8x Titan X cards, and monsters such as 12U multi-node "cluster" which includes total of 10×6-core Xeons and 16x GTX 980, the whole thing going at humble $40K total, by ExxactCorp. In any case, before investing a lot to buy dozens of specific servers, make sure to load-test them, and *load-test a lot* to make sure that they won't overheat under many hours of heavy load and datacenter-class thermal conditions (where you have 42 such 1U servers with one lying right on top of each other, ouch!, see Chapter [[TODO]] for further details).

To summarize: if your game cannot survive without server-side GPGPU simulations – it can be done, but be prepared to pay *a lot more than you would expect based on desktop GPU prices*, and keep in mind that deploying CUDA/OpenCL/Phi on servers will take much more effort than simply making your software run on your local Titan X ☹ . Also – make sure to start testing on real server rack-based hardware as early as possible, you do need to know ASAP whether hardware of your choice has any pitfalls.

"If your game cannot survive without server-side GPGPU simulations – it can be done, but be prepared to pay *a lot more than you would expect based on desktop GPU prices*

---

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

[7] I didn't use any of these, so I cannot really vouch for them, but at least you, IMHO, have reasonably good chances if you try; also make sure to double-check if your colocation provider is ready to host these not-so-mainstream boxes

[8] officially Supermicro doesn't support Titans, but their 1U boxes can be bought from 3rd-party VARs such as Thinkmate with 4x Titan X for a total of $10K, Titans included; whether it really works with Titans in datacenter environment 24×7 under your type of load – you'll need to see yourself

Simplifications. Of course, if your server doesn't need to support UDP, you won't need corresponding threads and FSMs. However, keep in mind that usually your connection to DB Server SHOULD be TCP (see "On Inter-Server Communications" section below), so if your client-to-server communication is UDP, you'll usually need to implement both. On the other hand, our QnFSM architecture provides a very good separation between protocols and logic, so usually you can safely start with a TCP-only server, and this will almost-certainly be enough to test your game intra-LAN (where packet losses and latencies are negligible), and implement UDP

support later (without the need to change your FSMs). Appropriate APIs which allow this kind of clean separation, will be discussed in Chapter [[TODO]].

On Inter-Server Communications

One of the questions you will face when designing your server-side, will be about the protocol used for inter-server communications. My take on it is simple:

even if you're using UDP for client-to-server communications, seriously consider using TCP for server-to-server communications

Detailed discussion on TCP (lack of) interactivity is due in Chapter [[TODO]], but for now, let's just say that poor interactivity of TCP (when you have Nagle algorithm disabled) becomes observable only when you have packet loss, and if you have non-zero packet loss within your server LAN – you need to fire your admins.[9]

On the positive side, TCP has two significant benefits. First, if you can get acceptable latencies without disabling Nagle algorithm, TCP is likely to produce much less hardware interrupts (and overall context switches) on the receiving server's side, which in turn is likely to reduce overall load of your Game Servers and even more importantly – DB Server. Second, TCP is usually much easier to deal with than UDP (on the other hand, this may be offset if you already have implemented UDP support to handle client-to-server communications).

---

[9] to those asking "if it is zero packet loss, why would we need to use TCP at all?" – I'll note that when I'm speaking about "zero packet loss", I can't rule out two packet lost in a day which can happen even if your system is really really well-built. And while a-few-dozen-microsecond additional delay twice a day won't be noticeable, crashing twice a day is not too good

QnFSM on Server Side: Flexibility and Deployment-Time/Run-Time Options.

When it comes to the available deployment options, QnFSM is an extremely flexible architecture. Let's discuss your deployment and run-time options provided by QnFSM in more detail.

Threads and Processes

First of all, you can have your FSMs deployed in different configurations depending on your needs. In particular, FSMs can be deployed as multiple-FSMs-per-thread, one-FSM-per-thread-multiple-threads-per-process, or one-FSM-per-process configurations (all this without changing your FSM code at all).[10]

"FSMs can be deployed as multiple-FSMs-per-thread, one-FSM-per-thread-multiple-threads-per-process, or one-FSM-per-process configurations (all this without changing your FSM code at all)

In one real-world system with hundreds of thousands simultaneous players but lightweight processing on the server-side and rather high acceptable latencies, they've decided to have some of game worlds (those for novice players) deployed as multiple-FSMs-per-thread, another bunch of game worlds (intended for mature players) – deployed as a single-FSM-per-thread (improving latencies a bit, and providing an option to raise thread priority for these FSMs), and those game worlds for pro players – as a single-FSM-per-process (additionally improving memory isolation in case of problems, and practically-unobservedly improving memory locality and therefore performance); all these FSMs were using absolutely very same FSM code, but it was compiled into different executables to provide slightly different performance properties.

Moreover, in really extreme cases (like "we're running a Tournament of the Year with live players"), you may even pin a single-FSM-per-thread to a single core (preferably the same where interrupts from you NIC come on this server) and to pin other processes to other cores, keeping your latencies to the absolute minimum.[11]

---

[10] Restrictions apply, batteries not included. If you have blocking calls from within your FSM, which is common for DB-style FSMs and some of gateway-style FSMs, you shouldn't deploy multiple-FSMs-per-thread

[11] yes, this will further reduce latencies in addition to any benefits obtained by simple increase of thread priority, because of per-core caches being intact

Communication as an Implementation Detail

With QnFSM, communication becomes an implementation detail. For example, you can have the same Game Logic FSM to serve both TCP and UDP. Not only it can come handy for testing purposes, but also may enable some of your players (those who cannot access your servers via UDP due to firewalls/weird routers etc.) to play over TCP, while the rest are playing over UDP. Whether you want this capability (and whether you want to match TCP players only with TCP players to

make sure nobody has an unfair advantage) is up to you, but at least QnFSM does provide you with such an option at a very limited cost.

Moving Game Worlds Around (at the cost of client reconnect)

Yet another flexibility option which QnFSM can provide (though with some additional headache, and a bit of additional latencies), is to allow moving your game worlds (or more generally – FSMs) from one server to another one. To do it, you just need to serialize your FSM on server A (see Chapter V for details on serialization), to transfer serialized state to a server's B Game Logic Factory, and to deserialize it there. Bingo! Your FSM runs on server B right from the same moment where it stopped running on server A. In practice, however, moving FSMs around is not that easy, as you'll also need to notify your clients about changed address where this moved FSM can be reached, but despite being an additional chunk of work, this is also perfectly doable if you really want it.

Online Upgrades

"Yet another two options provided by QnFSM, enable server-side software upgrades without stopping the server.

Yet another two options provided by QnFSM, enable server-side software upgrades while your system is running, without stopping the server.

The first of these options is just to start creating new game worlds using new Game Logic FSMs (while existing FSMs are still running with the old code). This works as long as changes within FSMs are minor enough so that all external inter-FSM interfaces are 100% backward compatible, and the life time of each FSM is naturally limited (so that at some point you're able to say that migration from the old code is complete).

The second of these online-upgrade options allows to upgrade FSMs while the game world is still running (via serialization – replacing the code – deserialization). This second option, however, is much more demanding than the first one, and migration problems may be difficult to identify. Therefore, severe automated testing using "replay" technique (also provided by QnFSM, see Chapter V for details) is strongly advised. Such testing should use big chunks of the real-world data, and should simulate online upgrades at the random moments of the replay.

On Importance of Flexibility

Quite often we don't realize how important flexibility is. Actually, we *rarely* realize how important it is until we run into the wall because of *lack* of flexibility. Deterministic FSMs provide *a lot* of flexibility (as well as other goodies such as post-mortem) at a relatively low development cost. That's one of the reasons why I

am positively in love with them.

DB Server

DB Server handles access to a database. This can be implemented using several very different approaches.

The first and the most obvious model is also the worst one. While in theory, it is possible to use your usual ODBC-style blocking calls to your database right from your Game Server FSMs, do yourself a favor and skip this option. It will have several significant drawbacks: from making your Game Server FSMs too tightly coupled to your DB to having blocking calls with undefined response time right in the middle of your FSM simulation (ouch!). In short – I don't know any game where this approach is appropriate.

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

DB API and DB FSM(s)

A much better alternative (which I'm arguing for) is to have at least one FSM running on your DB server, to have your very own message-based DB API (expressed in terms of messages or non-blocking RPC calls) to communicate with it, and to keep all DB work where it belongs – on DB Server, within appropriate DB FSM(s). An additional benefit of such a separation is that you shouldn't be a DB guru to write your game logic, but you can easily have a DB guru (who's not a game logic guru) writing your DB FSM(s).

DB API exposed by DB Server's FSM(s), SHOULD NOT be plain SQL (which would violate all the decoupling we're after). Instead, your DB API SHOULD be specific to your game, and (once again) should be expressed in game terms such as "take PC Z and place it (with all it's gear) into game world #NN". All the logic to implement this request (including pre-checking that PC doesn't belong to any other game world, modifying PC's row in table of PCs to reflect the number of the world where she currently resides, and reading all PC attributes and gear to pass it back) should be done by your DB FSM(s).

In addition, all the requests in DB API MUST be atomic; no things such as "open cursor and return it back, so I can iterate on it later" are ever allowed in your DB API (neither you will really need such things, this stands in spite of whatever-your-DB-guru-may-tell-you).

As soon as you have this nice DB API tailored for your needs, you can proceed with writing your Game Server FSMs, without worrying about exact implementation of

your DB FSM(s).

Meanwhile, at the King's Castle…

As soon as we have this really nice separation between Game Server's FSMs and DB FSM(s) via your very own message-based DB API, in a sense, the implementation of DB FSM will become an implementation detail. Still, let's discuss how this small but important detail can be implemented. Here I know of two major approaches.

Single-connection approach. This approach is very simple. You have run just one FSM on your DB Server and process everything within one single DB connection:

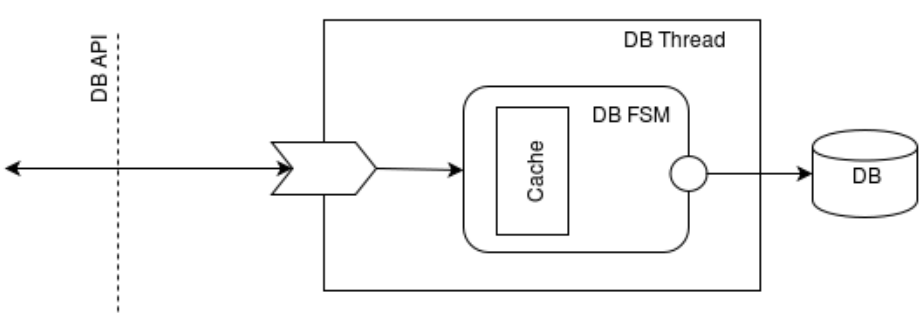


Fig VI.6

"Application-level cache has been observed to provide 10x+ performance improvement over DB cache even if all the necessary performance-related optimizations are made on the DB side

Here, there is a single DB FSM which has single DB connection (such as an ODBC connection, but there are lots of similar interfaces out there), which performs all the operations using blocking calls. A very important thing in this architecture is application-level cache, which allows to speed things up very considerably. In fact, this application-level cache has been observed to provide 10x+ performance improvement over DB cache even if all the necessary performance-related optimizations (such as prepared statements or even stored procedures) are made on the DB side. Just think about it – what is faster: simple hash-based in-memory search within your DB FSM (where you already have all the data, so we're speaking about 100 CPU clocks or so even if the data is out of L3 cache), or marshalling -> going-to-DB-side-over-IPC -> unmarshaling -> finding-execution-plan-by-prepared-statement-handle -> executing-execution-plan -> marshaling results -> going-back-to-DB-FSM-side-over-RPC -> unmarshaling results. In the latter case, we're speaking at least a few dozens of microseconds, or over 1e4 CPU clocks, over two orders of magnitude difference.[12] And with single connection to DB which is able to write data, keeping cache coherency is trivial. The main thing which gets cached for games is usually ubiquitous USERS (or PLAYERS) table, as well as some of small game-specific near-constant tables.

Despite all the benefits provided by caching, this schema clearly sounds as an heresy from any-DB-gal-out-there point of view. On the other hand, in practice it works surprisingly well (that is, as soon as you manage to convince your DB gal that you know what you're doing). I've seen such single-connection architecture[13] handling 10M+ DB transactions per day for a real-world game, and it were real transactions, with all the necessary changes, transactions, audit tables and so on.

Actually, at least at first stages of your development, I'm advocating to go with this single-connection approach.

It is very nice from many different points of view.

- First, it is damn simple.
- Second, there is no need to worry about transaction isolation levels, locks and deadlocks
- Third, it can be written as a real deterministic FSM (with all the associated goodies); moreover, this determinism stands (a) both if you "intercept calls" to DB for DB FSM itself, or (b) if we consider DB itself as a part of the FSM state, in the latter case no call interception is required for determinism.
- Fourth, the performance is very good. There are no locks whatsoever, the light is always green, so everything goes unbelievably smoothly. Add here application-level caching, and we have a winner! The single-connection system I've mentioned above, has had an *average* transaction processing time in below-1ms range (once again, with real-world transactions, commit after every transaction, etc.).

"There is no need to worry about transaction isolation levels, locks and deadlocks

The only drawback of this schema (and the one which will make DB people extremely skeptical about it, to put it very mildly) is an apparent lack of scalability. However, there are ways to modify this single-connection approach to provide virtually unlimited scalability[14] The ways to achieve DB scalability for this single-connection model will be discussed in Vol. 2.

One thing to keep in mind for this single-connection approach, is that it (at least if we're using blocking calls to DB, which is usually the case) is very sensitive to latencies between DB FSM and DB; we'll speak about it in more detail in Chapter [[TODO]], but for now let's just say that to get into any serious performance (that is, comparable to numbers above), you'll need to use RAID card with BBWC in write-

back mode[15], or something like NVMe, for the disk which stores DB log files (other disks don't really matter much). If your DB server is a cloud one, you'll need to look for the one which has low latency disk access (such things are available from quite a few cloud providers).

---

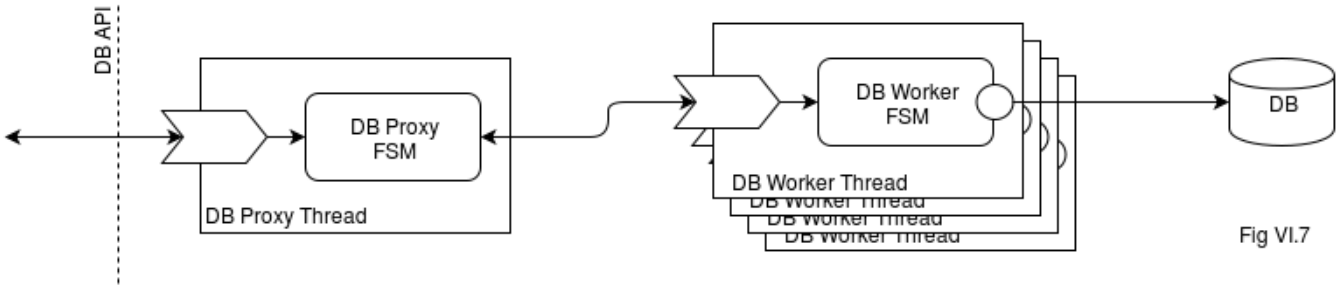
[12] with stored procedures the things become a bit better for DB side, but the performance difference is still considerable, not to mention vendor lock-in which is pretty much inevitable when using stored procedures

[13] with a full cache of PLAYERS table

[14] while in practice I've never went above around 100M DB transactions/day with this "single-connection-made-scalable" approach, I'm pretty sure that you can get to 1B pretty easily, and then it MAY become tough, as the number is too different from what-I've-seen so some unknown-as-of-now problems can start to develop. On the other hand, I daresay reaching these numbers is even more challenging with traditional multiple-connection approach

[15] don't worry, it is a perfectly safe mode for this kind of RAID, even for financial applications

Multiple-Connections approach. This approach is much more along the lines of traditional DB development, and is shown on Fig VI.7:



In short: we have one single DB-Proxy FSM (with the same DB API as discussed above),[16] which does nothing but dispatches requests to DB-Worker FSMs; each of these DB-Worker FSMs will keep its own DB connection and will issue DB requests over this connection. Number of these DB-Worker FSMs should be comparable to the number of the cores on your DB server (usually 2*number-of-cores is not bad starting number), which effectively makes this schema a kind of transaction monitor.

"The upside of

The upside of this schema is that it is inherently somewhat-scalable, but that's about it. Downsides, however, are numerous. The most concerning one is the cost of code maintenance in face of all those changes of logic, which are run in multiple connections. This inevitably leads us to a well-known but way-too-often-ignored discussion about transaction isolation levels, locks, and deadlocks at DB level. And if you don't know what it is – believe me, you Really don't

this schema is that it is inherently somewhat-scalable, but that's about it.

want to know about them. And updating DB-handling code when you have lots of concurrent access (with isolation levels above UR), is possible, but is extremely tedious. Restrictions such as "to avoid deadlocks, we must always issue all our SELECT FOR UPDATEs in the same order – the one written in blood on the wall of DB department" can be quite a headache to put it mildly.

Oh, and don't try using application-side caching for multiple-connections (i.e. even DB-Proxy SHOULD NOT be allowed to cache). While this is theoretically possible, re-ordering of replies on the way from DB to DB-Proxy make the whole thing way too complicated to be practical. While I've done such a thing myself once, and it worked without any problems (after several months of heavy replay-based testing), it was the most convoluted thing I've ever written, and I clearly don't want to repeat this experience.

But IMNSHO the worst thing about using multiple DB connections, is that while each of those DB FSMs can be made deterministic (via "call interception"), the whole DB Server cannot possibly be made deterministic (for multiple connections), period. It means that it may work perfectly under test, but fail in production while processing exactly the same sequence of requests.

Worse than that, there is a strong tendency for improper-transaction-isolation bugs to manifest themselves only under heavy load.

So, you can easily live with such a bug (for example, using SELECT instead of SELECT FOR UPDATE) quietly sitting in, but not manifesting itself until your Big Day comes, and then it crashes your site.[17] Believe me, you really don't find yourself in such a situation, it can be really (and I mean Really) unpleasant.

In a sense, working with transaction isolation levels is akin to working with threads: about the same problems with lack of determinism, bugs which appear only in production and cannot be reproduced in test environment, and so on. On the other hand, there are DB guys&gals out there who're saying that they can design a real-world multi-connection system which works under the load of 100M+ write transactions per day and never deadlocks, and I don't doubt that they can indeed do it. The thing which I'm not so sure about, is whether they really can maintain such quality of their system in face of new-code-required-twice-a-week, and I'm even less sure that you'll have such a person on your game team.

In addition, the scalability under this approach, while apparent, is never perfect (and no, those TPC-C linear-scalability numbers don't prove that linear scalability is achievable for real-world transactions). In contrast, single-connection-made-

scalable approach which we'll discuss in Vol. 2, can be extended to achieve perfect linear scalability (at least in theory).

---

[16] in particular, it means that we can rewrite our DB FSM from Single-connection to Multiple-connections without changing anything else in the system

[17] And it is not a generic "all the problems are waiting for the worst moment to happen" observation (which is actually purely perception), but a real deal. When probability of the problem depends on site load in a non-linear manner (and this is the case for transaction isolation bugs), chances of it happening for the first time exactly during your heavily advertised Event of the Year are huge.

DB Server: Bottom Line.

Unless you happen to have on your team a DB gal with real-world experience of dealing with locks, deadlocks, and transaction isolation levels for your specific DB under at least million-per-day DB write-transaction load – go for single-connection approach

If you do happen to have such a DB guru who vehemently opposes going single-connection – you can try multi-connection, at least if she's intimately familiar with SELECT-FOR-UPDATE and practical ways of avoiding deadlocks (and no, using RDBMS's built-in mechanism to detect the deadlock 10 seconds after it happens, is usually not good enough).

And in any case, stay away from any things which include SQL in your Game Server FSMs.

Failure Modes & Effects

When speaking about deployment, one all-important question which you'd better have an answer to, is the following: "What will happen if some piece of hardware fails badly?" Of course, within the scope of this book we won't be able to do a formal full-scale FMEA for an essentially unknown architecture, but at least we'll be able to give some hints in this regard.

FMEA

Failure mode and effects analysis (FMEA) was one of the first systematic techniques for failure analysis.

— Wikipedia —

Communication Failures

So, what can possibly go wrong within our deployment architecture? First of all, there are (not shown, but existing) switches (or even firewalls) residing between our servers; while these can be made redundant, their failures (or transient

software failures of the network stack on hosts) may easily cause occasional packet loss, and also (though extremely infrequently) may cause TCP disconnects on inter-server connections. Therefore, to deal with it, our Server-to-Server protocols need to account for potential channel loss and allow for guaranteed recovery after the channel is restored. Let's write this down as a requirement and remember until Chapter [[TODO]], where we will describe our protocols.

Server Failures

"Note that the stuff marked as 'High Availability', doesn't help with losing in-memory state: what we need to avoid losing in-memory state, is 'Fault-Tolerant' techniques.

In addition, of course, any of the servers can go badly wrong. There are tons of solutions out there claiming to address this kind of failures, but you should keep in mind that usually, the stuff marked as "High Availability", doesn't help with losing in-memory state: what you need *if* you want to avoid losing in-memory state, is "Fault-Tolerant" techniques (see "Server Fault Tolerance: King is Dead, Long Live the King!" section below).

Fortunately, though, for a reasonably good hardware (the one which has a reasonably good hardware monitoring, including fans, and at least having ECC and RAID, see Chapter [[TODO]] for more discussion on it), such fatal server failures are extremely rare. From my experience (and more or less consistently with manufacturer estimates), failure rate for reasonably good server boxes (such as those from one of Big Three major server vendors) is somewhere between "once-per-5-years" and "once-per-10-years", so if you'd have only one such server (and unless you're running a stock exchange), you'd be pretty much able to ignore this problem completely. However, if you have 100 servers – the failure rate goes up to "once or twice a month", which is unacceptable if such a failure leads to the whole site going down.

Therefore, at the very least you should plan to make sure that single failure of the single server doesn't bring your whole site down. BTW, most of the time it will be a Game World Server going down, as you're likely to have much more of these than the other servers, so at first stages you may concentrate on containment of Game World server failures. Also we can note that, counter-intuitively, failures of DB Server are not *that* important to deal with;[18] not because they have less impact (they do have much more impact), but because they're much less likely to happen that a failure of *one-of-Game-World-servers*.

---

[18] that is, beyond keeping a DB backup with DB logs being continuously moved to another location, see Chapter [[TODO]] for further discussion

Containment of Game World server failures

"If Game World server fails, it can be restarted from scratch, losing all the changes since last save-to-DB, but at least preserving previous results.

The very first (and very obvious) technique to minimize the impact of your Game World server failure on the whole site, is to make sure that your Game World reports relevant changes (without sending the whole state) to DB Server as soon as they occur. So that if Game World server fails, it can be restarted from scratch, losing all the changes since last save-to-DB, but at least preserving previous results. These saves-to-DB are the best to be done at some naturally arising points within your game flow.

For example, if your game is essentially a Starcraft- or Titanfall-like sequence of matches, then the end of each match represents a very natural save-to-DB point. In other words, if Game World server fails within the match – all the match data will be lost, but all the player standings will be naturally restored as of beginning of the match, which isn't too bad. In another example, for a casino-like game the end of each "hand" also represents the natural save-to-DB point.

If your gameplay is an MMORPG with continuous gameplay, then you need to find a way to save-to-DB all the major changes of the players' stats (such as "level has been gained", or "artifact has changed hands"). Then, if the Game Server crashes, you may lose the current positions of PCs within the world and a few hundred XP per player, but players will still keep all their important stats and achievements more or less preserved.

Two words of caution with regards to save-to-DB points. First,

For synchronous games, don't try to keep the whole state of your Game Worlds in DB

"If you disrupt the game-event-currently-in-progress for more than 0.5-2

Except for some rather narrow special cases (such as stock exchanges and some of slow-paced and/or "asynchronous" games as defined in Chapter I), saving all the state of your game world into DB won't work due to performance/scalability reasons (see discussion in "Taming DB Load: Write-Back Caches and In-Memory States" section above). Also keep in mind that even if you would be able to perfectly preserve the current state of the game-event-currently-in-progress (with game event being "match", "hand", or an "RPG fight") without killing your DB, there is another very big practical problem of psychological rather than technical nature. Namely, if you disrupt the game-event-

minutes, for almost-any synchronous multi-player game you won't be able to get the same players back, and will need to rollback the game event anyway.

currently-in-progress for more than 0.5-2 minutes, for almost-any synchronous multi-player game you won't be able to get the same players back, and will need to rollback the game event anyway.

For example, if you are running a bingo game with a hundred of players, and you disrupt it for 10 minutes for technical reasons, you won't be able to continue it in a manner which is fair to all the players, at the very least because you won't be able to get all that 100 players back into playing at the same time. The problem is all about numbers: for two-player game it might work, for 10+ – succeeding in getting all the players back at the same time is extremely unlikely (that is, unless the event is about a Big Cash Prize). I've personally seen a large commercial game that handled the crashes in the following way: to restore after the crash, first, it rolled forward its DB at the DB level to get perfectly correct current state, and then it rolled all the current game-events back at application level, exactly because continuing these events wasn't a viable option due to the lack of players.

Trying to keep all the state in DB is a common pitfall which arises when the guys-coming-from-single-player-casino-game-development are trying to implement something multiplayer. Once again: don't do it. While for a single-player casino game having state stored in DB is a big fat Business Requirement (and is easily doable too), for multi-player games it is neither a requirement, nor is feasible (at least because of the can't-get-the-same-players-together problem noted above). Think of Game World server failure as of direct analogy of the fire-in-brick-and-mortar-casino in the middle of the hand: the very best you can possibly do in this case is to abort the hand, return all the chips to their respective owners (as of the beginning of the hand), and to run out of the casino, just to come back later when the fire is extinguished, so you can start an all-new game with all-new players.

The second pitfall on this way is related to DB consistency issues and DB API.

Your DB API MUST enforce logical consistency

For example, if (as a part of your very own DB API) you have two DB requests, one of which says "Give PC X artifact Y", and another one "Take artifact Y from PC X", and are trying to report an occurrence of "PC X took over artifact Y from PC XX" as two separate DB requests (one "Take" and one "Give"), you're risking that in case of Game World server failure, one of these two requests will go through, and the other one won't, so artifact will get lost (or will be duplicated) as a result. Instead of using these two requests to simulate "taking over"

"You should have a special

occurrence, you should have a special DB request "PC X took over artifact Y from PC XX" (and it should be implemented as a single DB transaction within DB FSM); this way at least the consistency of the system will be preserved, so whatever happens – there is still exactly one artifact. The very same pattern MUST be followed for passing around anything of value, from casino chips to artifacts, with any other goodies in between.

DB request "PC X took over artifact Y from PC XX" (and it should be implemented as a single DB transaction within DB FSM)

Server Fault Tolerance: King is Dead, Long Live the King!

If you want to have your servers to be really fault-tolerant, there are some ways to have your cake and eat it too.

However, keep in mind, that all fall-tolerant solutions are complicated, costly, and in the games realm I generally consider them as an over-engineering (even by my standards).

Fault-Tolerant Servers: Damn Expensive

Historically, fault-tolerant systems were provided by damn-expensive hardware such as [Stratus] (I mean their hardware solutions such as ftServer; see discussion on hardware-vs-software redundancy in Chapter [[TODO]]) and [HPIntegrityNonStop] which have everything doubled (and CPUs often quadrupled(!)) to avoid all single points of failure, and these tend do work very well. But they're usually way out of game developer's reach for financial reasons, so unless your game is a stock exchange – you can pretty much forget about them.

Fault-Tolerant VMs

Fault-Tolerant VMs (such as VMWare FT feature or Xen Remus) are quite new kids on the block (for example, VMWare FT got beyond single vCPU only in 2015), but they're already working. However, there are some significant caveats. *Take everything I'm saying about fault-tolerant VMs with a really good pinch of salt, as all the technologies are new and evolving, and information is scarce; also I admit that I didn't have a chance to try these things myself* 🙁 .

When you're using a fault-tolerant VM, the Big Picture looks like this: you have two commodity servers (usually right next to each other), connect them via 10G Ethernet, run VM on one of them (the "primary" one), and when your "primary" server fails, your VM magically reappears on the "secondary" box. From what I can see, modern Fault-Tolerant VMs are using one of two technologies: "virtual lockstep" and "fast checkpoints".

Unfortunately, each of them has its own limitations.

> "Modern Fault-Tolerant VMs are using one of two technologies: 'virtual lockstep' and 'fast checkpoints'. Unfortunately, each of them has its own limitations.

Virtual Lockstep: Not Available Anymore?

The concept of virtual lockstep is very similar to our QnFSM (with the whole VM treated as FSM). Virtual lockstep takes one single-core VM, intercepts all the inputs, passes these inputs to the secondary server, and runs a copy VM there. As any other fault-tolerant technology, virtual lockstep causes additional latencies, but it *seems* to be able to restrict its appetite for additional latency to a sub-ms range, which is acceptable for most of the games out there. Virtual lockstep is the method of fault-tolerance vSphere prior to vSphere v6 was using. The downside of virtual lockstep is that it (at least as implemented by vSphere) wasn't able to support more that one core. For our QnFSMs, this single-core restriction wouldn't be *too much* of a problem, as they're single-threaded anyway (though balancing FSMs between VMs would be a headache), but there are lots of applications out there which are still heavily-multithreaded, so it was considered an unacceptable restriction. As a result, vSphere, starting from vSphere 6, has changed their fault-tolerant implementation from virtual lockstep to checkpoint-based implementation. *As of now, I don't know of any supported implementations of Virtual Lockstep* ☹ .

Checkpoint-Based Fault Tolerance: Latencies

To get around the single-core limitation, a different technique, known as "checkpoints", is used by both Xen Remus and vSphere 6+. The idea behind checkpoints is to make a kind of incremental snapshots ("checkpoints") of the full state of the system and log it to a safe location ("secondary server"). As long as you don't let anything out of your system before the coming-later "checkpoint" is committed to a secondary server, all the calculations you're making meanwhile, become inherently unobservable from the outside, so in case of "primary" server failure, it is not possible to say whether it didn't receive the incoming data at all. It means that for the world outside of your system, your system (except for the additional latency) becomes almost-indistinguishable[19] from a real fault-tolerant server such as Stratus (see above). In theory, everything looks perfect, but with VM checkpoints we *seem* to hit the wall with checkpoint frequency, which defines the minimum possible latency. On systems such as VMWare FT, and Xen Remus, checkpoint intervals are measured in dozens of milliseconds. If your game is ok with such delays – you're fine, but otherwise – you're out of luck ☹ . For more details on checkpoint-based VMs, see [Remus].

Saving for latencies (and the need to have 10G connections between servers, which is not that big deal), checkpoint-based fault tolerance has several significant advantages over virtual lockstep; these include such important things as support

for multiple CPU cores, and N+1 redundancy.

---

[19] strictly speaking, the difference can be observed as some network packets may be lost, but as packet loss is a normal occurrence, any reasonable protocol should deal with transient packet loss anyway without any observable impact

Complete Recovery from Game World server failures: DIY Fault-Tolerance in QnFSM World

If you're using FSMs (as you should anyway), you can also implement your own fault-tolerance. I should confess that I didn't try this approach myself, so despite looking very straightforward, there can be practical pitfalls which I don't see yet. Other than that, it should be as fault-tolerant as any other solution mentioned above, *and* it should provide good latencies too (well in sub-ms range).

As any other fault-tolerant solution, for games IMHO it is an over-engineering, but if I'd feel strongly about the failures causing per-game-event rollbacks, this is the one I'd try first. It is latency friendly, it allows for N+2 redundancy (saving you from doubling the number of your servers in case of 1+1 redundancy schemas), and it plays really well alongside our FSM-related stuff.

The idea here is to have separate Logging Servers logging all the events to all the FSMs residing on your Game World servers; then, you will essentially have enough information on your Logging Servers to recover from Game World server failure. More specifically, you can do the following:

- have an additional Logging Server(s) "in front of Game Servers"; these Logging Server(s) perform two functions:
    - log all the messages incoming to all Game Server FSMs
        - these include: messages coming from clients, messages coming from other Game Servers, and messages coming from DB Server
        - moreover, even communications between different FSMs residing on the same Game Server, need to go via Logging Server and need to be logged
    - timestamp all the incoming messages
- all your Game Server FSMs need to be strictly-deterministic
    - in particular, Game Server FSMs won't use their own clocks, but will use

timestamps provided by Logging Servers instead

- In addition, from time to time each of Game Server FSMs need to serialize its whole state, and report it to Logging Server

- then, we need to consider two scenarios: Logging Server failure and Game Server failure (we'll assume that they never fail simultaneously, and such an event is indeed extremely unlikely unless it is a fire-in-datacenter or something)

    - if it is Logging Server which fails, we can just replace it with another (re-provisioned) one; there is no game-critical data there

    - if it is Game Server which fails, we can re-provision it, and then roll-forward each and every FSM which was running on it, using last-reported-state and logs-saved-since-last-reported-state saved on the Logging Server. Due to the deterministic nature of all the FSMs, the restored state will be exactly the same as it was a few seconds ago[20]

        - at this point, all the clients and servers which were connected to the FSM, will experience a disconnect

        - on disconnect, the clients should automatically reconnect anyway (this needs to account for IP change, what is a medium-sized headache, but is doable; in [[TODO]] section we'll discuss Front-End servers which will isolate clients from disconnects completely)

        - issues with server-to-server messages should already be solved as described in "Communication Failures" subsection above

"if it is Game Server which fails, we can re-provision it, and then roll-forward each and every FSM which was running on it

In a sense, this "Complete Recovery" thing is conceptually similar to EventProcessorWithCircularLog from Chapter V (but with logging residing on different server, and with auto-rollforward in case of server failure), or to a traditional DB restore-and-log-rollforward.

Note that only hardware problems (and software bugs outside of your FSMs, such as OS bugs) can be addressed with this method; bugs within your FSM will be replayed and will lead to exactly the same failure 🙁 .

Last but not least, I need to re-iterate that I would object any fault-tolerant schema for most of the games out there on the basis of over-engineering, though I admit that there might be good reasons to try achieving it, especially if it is not too expensive/complicated.

---

[20] or, in case of almost-strictly-deterministic FSMs such as those CUDA-based ones, it will be almost-exactly-the-same

[[TODO!]] DIY VIrtual-Lockstep

Classical Game Deployment Architecture: Summary

To summarize the discussion above about Classical Game Deployment Architecture:

- It works
- It can and should be implemented using QnFSM model with deterministic FSMs, see discussion above for details
- Your communication with DB (DB API) SHOULD use game-specific requests, and SHOULD NOT use any SQL; all the SQL should be hidden behind your DB FSM(s)
- Your first DB Server SHOULD use single-connection approach, unless you happen to have a DB guy who has real-world experience with multi-connection systems under at least millions-per-day write(!) transaction loads
  - Even in the latter case, you SHOULD try to convince him, but if he resists, it is ok to leave him alone, as long as external DB API is still exactly the same (message-based and expressed in terms of whatever-your-game-needs). This will provide assurance that in the extreme case, you'll be able to rewrite your DB Server later.

[[To Be Continued…

This concludes beta Chapter VI(a) from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter VI(b), "Modular Architecture: Server-Side. Throwing in Front-End Servers.]]

[–] References

[Lightstreamer] http://www.lightstreamer.com/
[Redis.CAS] http://redis.io/topics/transactions#cas
[Zubek2016] Robert Zubek, "Private communications with"
[Zubek2010] Robert Zubek, "Engineering Scalable Social Games", GDC2010

[Stratus] "Stratus Technologies", Wikipedia
[HPIntegrityNonStop] "HP Integrity NonStop", Wikipedia
[Remus] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

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

Chapter VI(b). Server-Side Architecture. Front-End Servers a... »

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