




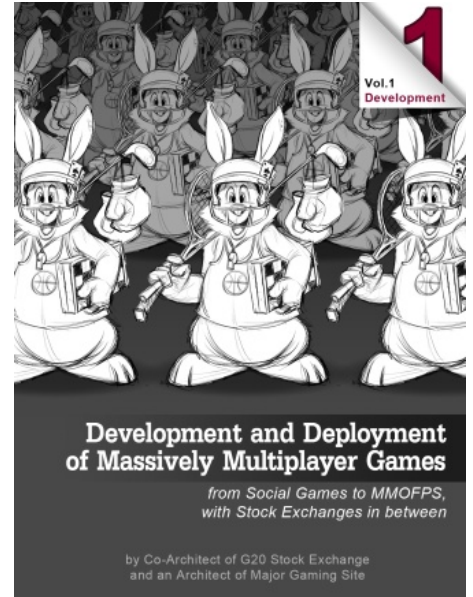
## IT Hare on Software

# Chapter VI(b). Server-Side Architecture. Front-End Servers and Client-Side Random Load Balancing

posted December 28, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko 

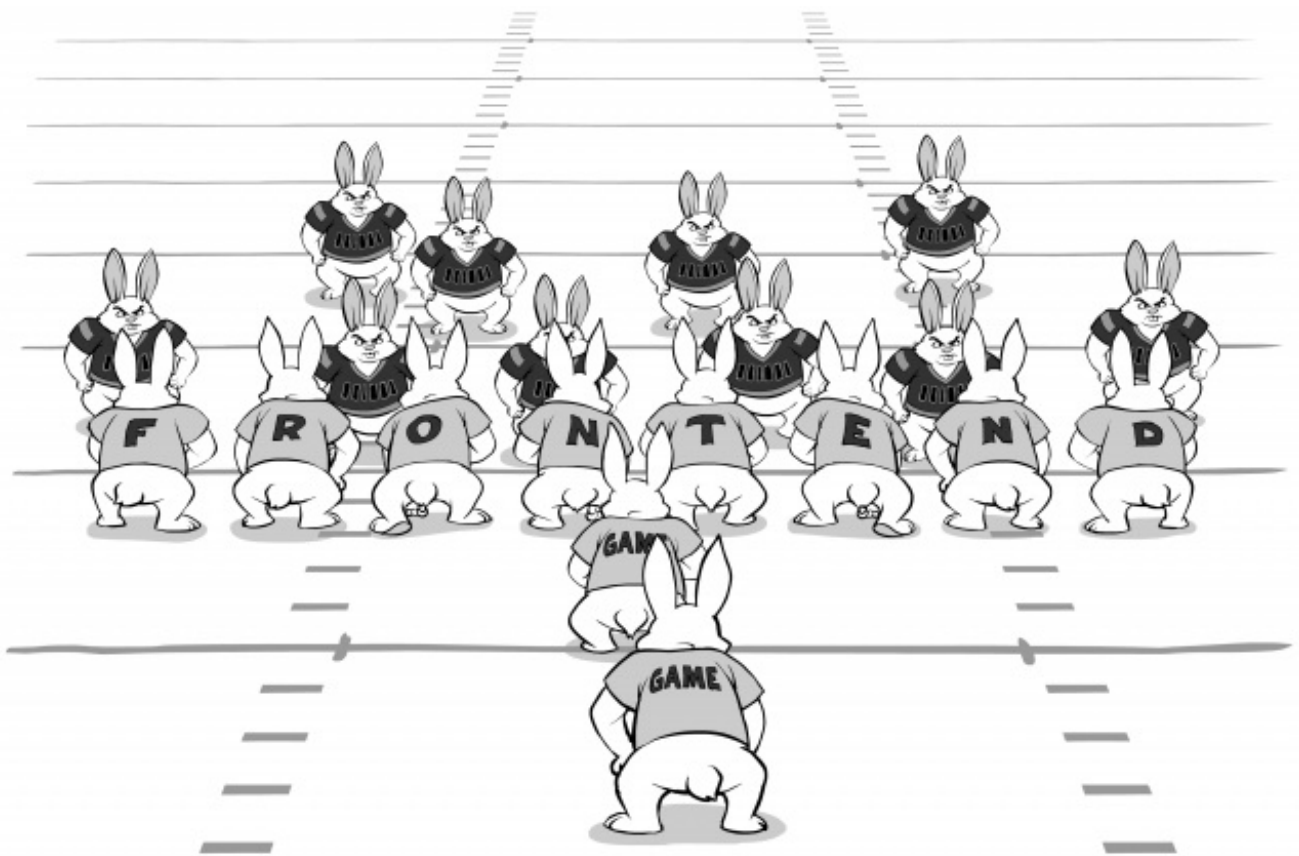
[[This is Chapter VI(b) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further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

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



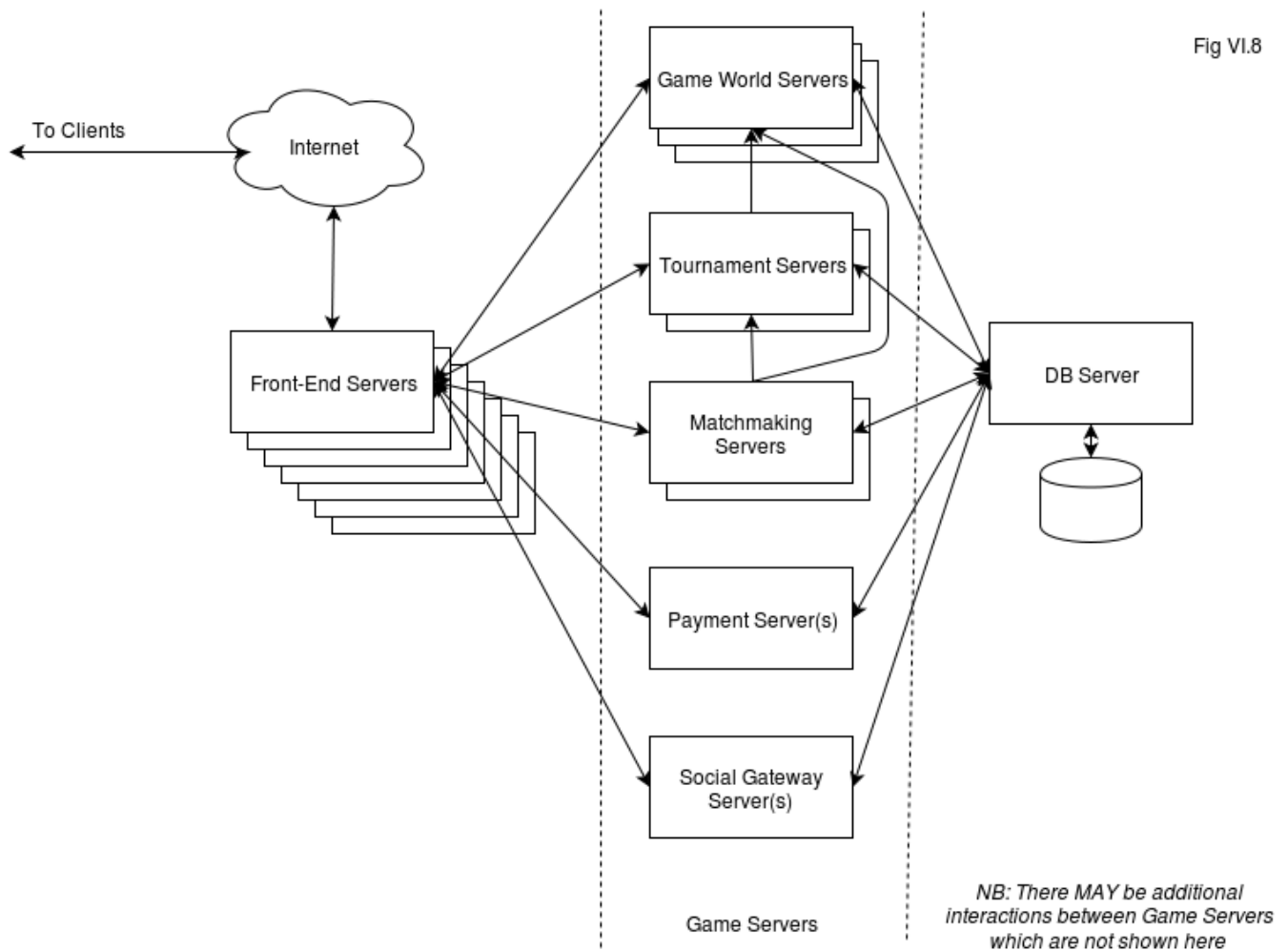
## Enter Front-End Servers

*[Enter Juliet] Hamlet: Thou art as sweet as the sum of the sum of Romeo and his horse and his black cat! Speak thy mind! [Exit Juliet]*  
— a sample program in Shakespeare Programming Language —



Our Classical Deployment Architecture (especially if you do use FSMs) is not bad, and it will work, but there is still quite a bit of room for improvement for most of the games out there. More specifically, we can add another row of servers in front of the Game Servers, as shown on Fig VI.8:

Fig VI.8



As you see, compared to the Classical Deployment Architecture (see Fig VI.4 above) we've just added a row of Front-End Servers in front of our Game Servers. These additional Front-End Servers are intended to deal with all the communication stuff when it comes from the clients. All those pesky “whether the player is connected or not” questions (including keep-alive handling where applicable, see Chapter [[TODO]] for details on keep-alives), all that client-to-server encryption (if applicable), with all those keys etc., all those rather more-or-less strange reliable-UDP protocols (again, if applicable), and of course, routing messages between the clients and different Game Servers – all the communication with clients is handled here.

In addition, usually these Front-End servers store a copy of relevant Game Worlds when it is necessary, and are acting as “concentrators” for the game world updates; i.e. even if a Game Server has 100'000 people watching some game (like final of some tournament or something), it will need to send updates only to a few Front-End servers, and Front-End servers will take care of data distribution to all the 100'000 people. This ability comes really handy when you have some kind of Big Final game, with thousands of people willing to watch it (and you don't really need to make it a video broadcast, which is not-so-convenient for existing players and damn expensive, but you can do it right within your client).



**“ In addition, usually these Front-End servers store a copy of**

More on it below, and implementation of this “concentrator” paradigm is discussed in more detail in Chapter [TODO].

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

**Front-End Servers MUST be easily replaceable without significant inconveniences to players**

**relevant Game Worlds when it is necessary, and are acting as “concentrators” for the game world updates**

That is, if any of Front-End Servers fails for whatever reason – the most a player should see, is a disconnect for a few seconds. While still disruptive, it is very much better than scenarios such as “the whole game world went down and we need to restore it from backup”. In other words, whenever Front-End server crashes for whatever reason, all the clients who were connected there, need to detect the crash (or even worse, “black hole”) and automagically reconnect to some other Front-End server; in this case all the player can see, is a momentarily disconnect (which is also a nuisance, but is much better than to see your game hang).

## Front-End Servers: Benefits

Whenever we’re adding another layer of complexity, there is always a question “Do we really need it?” From what I’ve seen, having easily replaceable Front-End Servers in front of your Game Servers is very valuable and provides quite a few benefits. More specifically:

- Front-End Servers take some load off your Game Servers, while being easily replaceable
  - it means that you can have less Game Servers
    - this, combined with the observation that Front-End Servers are easily replaceable, means that you improve reliability of your site as a whole; instances when some of your Game World Servers go down, will occur more rarely (!)
  - having a copy of relevant game world(s) on your Front-End Servers, takes even more load off your Game Servers, and makes Game Server load independent on the number of observers
- you can use really cheap boxes for your Front-End Servers; strictly speaking, you don’t even need ECC and RAID for them (and you certainly do need them for your Game Servers). As noted above, Front-End Servers are easily replaceable, so if one goes down – its load is automagically redistributed among the others (see Chapter [TODO] for further details). If you’re going to



deploy into the cloud – you may want to consider cheaper offers for your front-end servers (even if they're coming from different CSP).<sup>1</sup>

## **really cheap boxes for your Front-End Servers**

- they allow your client to have a single connection point to the whole site; benefits of this approach include better control over player's "last mile" so that priorities between different data streams can be controlled, eliminating difficult-to-analyze "partial connections", and hiding more implementation details of your site from the hostile world outside; more on single client connection in Chapter [TODO]
- they allow for trivial client-side load balancing (no hardware load balancers needed, etc. etc.), more discussion on the load balancing below in "On Client-Side Load Balancing and Law of Big Numbers" section below
- having a copy of relevant game world(s) on your Front-End Servers allows to have virtually unlimited number of observers who want to watch some of the games being played on your site (such as a Big Final or something<sup>2</sup>) Best of all, this will happen *without affecting game server's performance (!)*. Moreover, usually you won't need to organize anything for your Big Final, the system (if built properly) can take care of it itself, in (roughly) the following manner:
  - whenever somebody comes to watch a certain game, his client requests this game from the Front End Server
  - if Front End Server doesn't have a copy of the requested game, it requests it from the relevant Game Server, alongside with updates to the game world state
  - from this point on, Front End Server will keep an "in-sync" copy of the game world, providing it (with updates) to all the clients which have requested it
  - it means that from this point on, even if you have 100'000 observers watching some game on this Game Server, all the additional load is handled by your Front-End Servers, without affecting your Game Server
  - for further details, see Chapter [TODO].
- Front-End Servers allow for better security later on (acting essentially as a kind of DMZ, see Chapter [TODO] for details).

---

<sup>1</sup>keep in mind that you still need top-notch connectivity

<sup>2</sup>and as Big Finals are a good way to attract attention, this does provide you an edge over your competitors, etc. etc.

## **Front-End Servers: Latencies and Inter-Player Latency Differences**



**“You can have processing time of your Front End server application-layer of the order of single-digit microseconds.**

As for the negative side of having Front End Servers, I can think only of two such drawbacks. The first one is additional latency introduced by your Front End Server. More specifically, we’re speaking about the time which is necessary for the packet incoming from a client at application layer, to get processed by your Front End Server, to go into TCP stack on Front End Server side,<sup>3</sup> to get out of TCP stack on Game Server side, and to reach application layer in your Game Server (plus the time necessary to go in the opposite direction).

Let’s take a look at this additional latency. From my experience, if you’re using a reasonably good communication layer library, you can have processing time of your Front End server application-layer of the order of single-digit microseconds.<sup>4</sup> Then, we have an end-to-end TCP connection from your Front End Server to your Game Server; latencies of such a connection (over 10GB Ethernet) have been measured at around 8  $\mu$ s [Larsen2007]. Adding these two delays together and multiplying it by two to get RTT, would mean that we’re still staying well below 100  $\mu$ s. However, there are some further considerations (such as switch delays, differences between different operating systems, differences between games, etc.) which make me uncomfortable to say that you will have no problem achieving 100  $\mu$ s delay (i.e. either you may, or you may not). On the other hand, I am ready to say that if you’re careful enough with your implementation, reducing the delay introduced by Front-End Servers, down to 1ms is achievable in all but most weird cases.

To summarize:

- if additional latency of around 1 millisecond is ok for you – don’t worry about additional latencies and go for Front-End Servers; this certainly covers all genres with the only potential exception being MMOFPS
- if additional latency you can live with, is well below 1 millisecond (which is difficult for me to imagine as it is still over an order of magnitude less than 1/60 sec frame update time, but in MMOFPS world pretty much anything can happen) – think about it a bit more and try to find out (ideally – experimentally) what kind of latency you can achieve in practice; if your experiments show that latencies are indeed unacceptable, you MIGHT need to drop those Front-End Servers because of the latency they’re introducing<sup>5</sup>
- YMMV, no warranties of any kind, batteries not included

The second (IMHO more theoretical, but as usual, YMMV) potential issue with having Front-End Servers would arise if some of your Front-End Servers are overloaded (or they’re running using significantly different hardware), so those

players connected to less-loaded Front-End Servers, will have lower latencies, and therefore will have an advantage.

On the one hand, I didn't see situations where it makes any practical difference in real-world deployments (i.e. as I've seen it, if some of the Front-End Servers are overloaded, it means that most of the other ones are already at 90%+ of capacity, which you should avoid anyway; see [[TODO!]] section for further discussion of load balancing). On the other hand, YMMV and in theory you might get hit by such an effect (though I certainly don't see it coming into play for anything but MMOFPS).



**“If/when such inter-player latency becomes a real problem, you MAY need to implement some kind of affinity for players of certain Game Worlds to certain Front-End servers**

If such inter-player latency differences become the case (and only when/if it becomes a real problem), you MAY need to implement some kind of affinity for players of certain Game Worlds to certain Front-End servers (more on affinity in “On Affinity” section below). However, keep in mind that large-scale affinity tends to remove most of the benefits provided by Front-End Servers, so if you feel that you're going to implement affinity for each-and-every-game – you'll probably be better without Front-End Servers (implementing affinity only for a small percentage of your games, such as “high profile tournaments” will cause less trouble, see “On Affinity” section below for further discussion).

---

<sup>3</sup> yes, I'm arguing for TCP connections for inter-server communications in most cases, see “On Inter-Server Communication” section above. On the other hand, UDP is also possible if you really really prefer it.

<sup>4</sup> note that this might become a non-trivial exercise, see further discussion in Chapter [[TODO]]. On the other hand, I've done it myself.

<sup>5</sup> in theory, you may also want to experiment with something like Infiniband, which BTW would fit nicely in overall QnFSM architecture with communications neatly isolated from the rest of the code, but most likely it won't be worth the trouble

## **Client-Side Random Balancing and Law of Big Numbers**

As soon as you have several Front-End servers where your clients are coming, you have a question “how to ensure that all the Front-End Servers are loaded equally”, i.e. a typical load balancing question. Load balancing in general is quite a big topic at least over last 20 years. Three most common techniques out there are the following: DNS Round-Robin, Client-Side Random Balancing, and Server-Side (usually hardware-based) Load Balancers. With the industry producing those hardware boxes behind the last one, there is no wonder that it becomes more and

more popular at least in the enterprise world. Still, let's take a closer look at these load balancing solutions.

## DNS Round-Robin



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

DNS round-robin is based on a traditional DNS requests. Whenever a client requests address frontend.yoursite.com to be resolved into IP address, a DNS request is sent (this stands with or without DNS round-robin) to your (or “your DNS provider’s”) DNS server. If DNS server is configured for DNS round-robin, it returns *different* IP addresses to different DNS requests, in a round-robin fashion<sup>6</sup> hence the name.

DNS Round-Robin, when applied to balancing browsers across different web servers, has two major disadvantages. First of all, there is a problem with caching DNS servers along the path of the request (which is a very standard part of DNS handling). That is, even if your server is faithfully returning all your IPs in a round robin fashion, one of these returned IPs can get cached by a Big Fat DNS server (think Comcast or AT&T), and then get distributed to many thousands of clients; in this case distribution of your clients across your servers will be skewed towards that “lucky” IP which got cached by the Big Fat DNS server 😞 . The second problem with using DNS round-robin

for web servers, is that if one of your servers is down, usual web browser won't try another server on the list, so usually in web server realm round-robin DNS doesn't provide server fault tolerance.

Fortunately, as we DO have a client, we can solve both these problems very easily. Moreover, these techniques will also work for your browser-based games (that is, *after* you've got your JS loaded and it started execution).

---

<sup>6</sup> strictly speaking, it is a little bit more complicated than that, as DNS packets contain a list of servers, but as virtually everybody out there ignores all the entries in returned packet except for the very first one, it is more or less equivalent to returning only one IP per request – that is, unless you have your own client which can do the choice itself, see “Client-Side Balancing”

## Client-Side Random Balancing



To improve on DNS round-robin, a very simple idea can be used. We won't rotate anything on the server side; instead, we will distribute *exactly the same* list of servers to all the clients. This list may be hardcoded into your clients (and that's what I've used personally with big success), or the list can be distributed via DNS as a simple list of IPs for desired name (and retrieved on client via `getaddrinfo()` or equivalent). Which way to prefer – doesn't matter to us now, but we'll discuss relevant issues in Chapter [TODO].



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

As soon as the client gets the list of IPs, everything is very simple. Client simply takes random item from the IP list, and tries connecting to this randomly chosen IP. If connection attempt is unsuccessful (or connection is lost, etc.) – client gets another random item from the list and tries connecting again.

One note of caution – while you don't really need a cryptographic-quality random generator to choose the IP from the list, you DO want to avoid situations when your random number generator (the one used for this purpose) is essentially just some function of coarse-grained time. One Really Bad example would be something like

```
1  int myrand() {//DON'T DO THIS!
2      srand(time(0));
3      return rand();
4  }
```

In such a case, if you get mass disconnect (and as a result all your players will attempt to reconnect at about the same time), your IP distribution will likely get skewed due to too few differences between the clients trying to get their IP addresses; if all the clients attempt to connect within 5 seconds, with such a bad *myrand()* function you'll get at most 5 different IPs (less if you're unlucky). Other than such extremely bad cases, pretty much any RNG should be fine for this purpose. Even a trivial linear congruential generator, seeded with `time(0)` *at the moment when the program was launched* (and NOT at the moment of request, as in example above), should do in practice, though adding some kind of milliseconds or some other randomly looking or client-specific data to the mix is advisable “just in case”.

**Law of Large Numbers**  
**According to the law, the**

**Client-Side Random Balancing: a Law of Large Numbers, and comparison with DNS Round-Robin**

Unlike DNS round-robin (which in theory provides “ideal” balancing), client-side random balancing relies on the statistical Law of Large Numbers to achieve flat distribution of clients between the servers. What the law basically says is

**average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed.**

— Wikipedia —

that for independent measurements, the more experiments you're performing – the more flat distribution you'll get. [[TODO!: add stuff about binomial distribution, and an example]]

In practice, despite being “non-ideal” in theory, client-side random balancing achieves much more flat distribution than DNS round-robin. The reason for it is two-fold. First, as soon as the number of clients is large (hundreds and up), client-side random balancing becomes sufficiently flat for practical purposes (and if your system is provisioned for thousands of players, and only a few have come yet – the distribution won't be too flat, but the inequality involved won't be able to hurt, and the balance will improve as the number grows). On the positive side, however, client-side random balancing doesn't suffer from DNS caching issue described above. Even if you're using DNS to distribute IP lists (and this list gets cached) – with client-side balancing *all the IP lists circulating in the system are identical by design*, so caching (unlike with DNS round-robin) doesn't change client distribution at all.

To summarize: personally, I would be very cautious to use DNS Round-Robin for production load balancing. On the other hand, I've seen Client-Side Random Balancing to work extremely well for a game which grew from a few hundreds of simultaneous players into hundreds of thousands; it worked without any problems whatsoever, providing almost-perfect balancing all the time. That is, if the average load across the board was 50%, you could find some servers at 48% and some at 52%, but not more than that.<sup>7</sup>...

As for the second disadvantage mentioned above for DNS Round-Robin as applied to web browsers (which was inability of most of the browsers to provide fault tolerance in case when one of the servers crashes) – this evaporates as soon as we have the whole list on the client-side, can detect failure, and can select another item from the list.

---

<sup>7</sup> this, of course, stands only when you have run your servers identically for sufficient time; if one of the servers has just entered service, it will take some hours until it reaches the same load level than the others. If really necessary, this effect can be mitigated, though mitigation is rather ugly and I've never seen it necessary in practice

## Server-Side Load Balancers

An approach which is very different from both round-robin DNS and client-side

random balancing, is to use server-side load balancers. Load balancer is usually an additional box, sitting in front of your servers, and doing, as advertised, load balancing.

Server-side load balancers do have significantly more balancing capabilities with regards to scenarios when different clients cause very different loads (so that server-side balancers can work even if the Law of Large Numbers doesn't work anymore). However, on the one hand, these additional balancing capabilities are usually completely unnecessary for games (where Law of Large Numbers tends to stand very firmly), and on the other hand, such load balancer boxes tend to be damn expensive (double that if you want redundancy, and you certainly want it), they do not allow inter-datacenter balancing and fault tolerance (by design), and they introduce additional not-so-well-controlled latencies.<sup>8</sup>...



**“These additional balancing capabilities are usually completely unnecessary for games (where Law of Large Numbers tends to stand very firmly)”**

Oh, and BTW – when speaking about redundancy and the cost of their boxes, quite a few hardware manufacturers will tell you “hey, you can use our balancer in active/active configuration, so you won't waste anything!”. Well, while you *can* indeed use many server-side load balancers in active/active configuration, you still **MUST** have at least one redundant box to handle the load if one of those boxes fails. In other words, if all you have is two boxes in active/active configuration, when both are working, overall load on each of them **MUST** be well below 50%, there is no way around it if you want redundancy.

As a result of all the considerations above, for game load-balancing purposes I have never seen any practical uses for server-side load balancer boxes (as always, YMMV and batteries are not included). Even if you're using Web-Based Deployment Architecture (in the way described above), you should be able to stay away from them (though YMMV even more).

---

<sup>8</sup> most of load balancers are designed to balance web sites where anything below 100ms is pretty much nothing, so at the very least make sure to discuss and measure the latency (under your kind of load!) before buying such a box

## Balancing Summary

From my experience, client-side random balancing (aimed towards front-end servers) worked really good, and I've never seen any reasons to use something different. Round-robin DNS is almost universally inferior to client-side balancing, and hardware-based server-side balancers are too complicated and expensive,

usually without any real reason to use them in gaming environment. As note above, one exception when you MAY need server-side balancers, is if you're using Web-Based Deployment Architecture.

One last word about load balancing: it *is* possible to use more than one of the methods listed here (and it might even work for you); however, implications of such combined use of more than one method of load balancing, are way too convoluted to discuss them in this book.

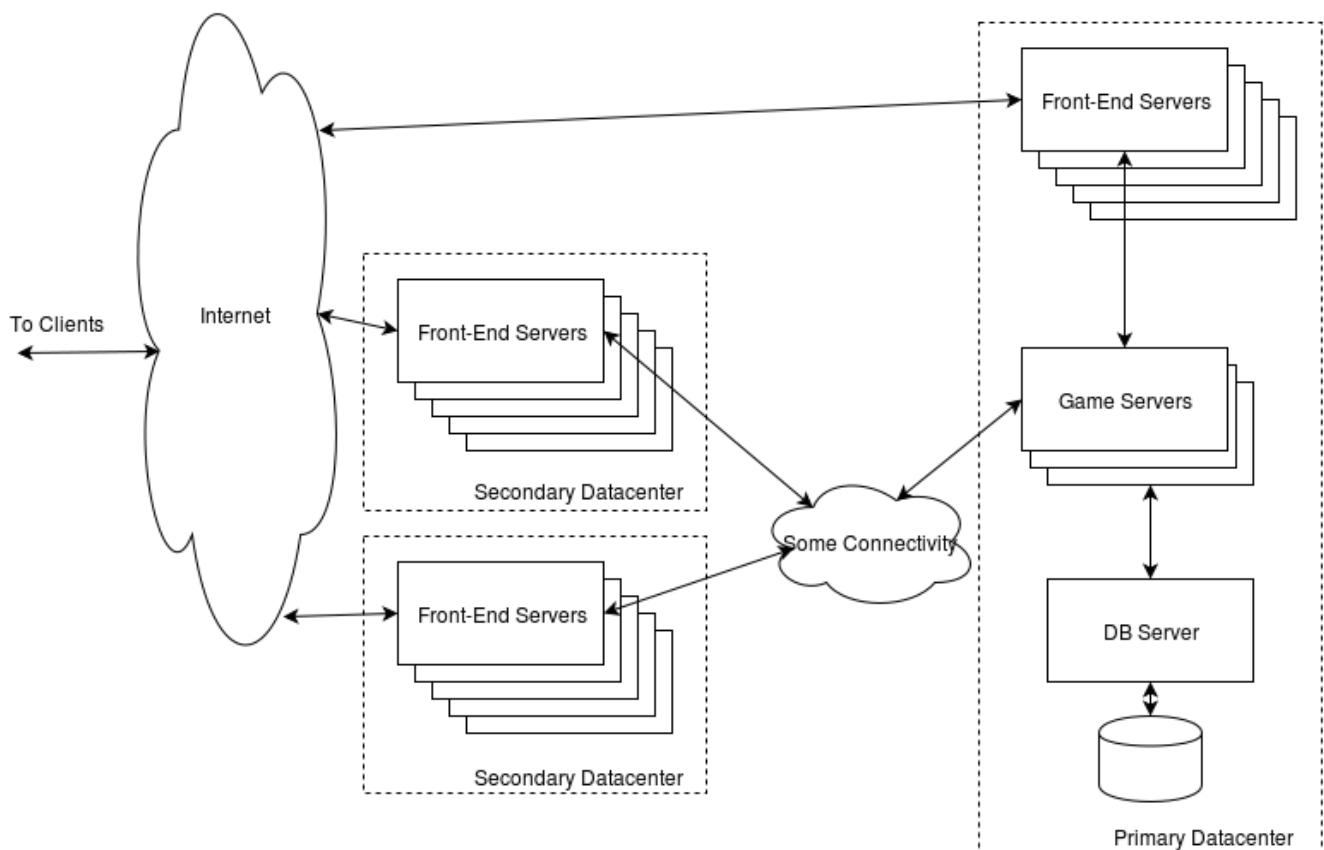
## Front-End Servers as a CDN

It is possible to use Front-End Servers as a kind of CDN (or even use them to build your own CDN). Even if you're running all your Game Servers from one single datacenter, for certain kinds of games it might be a good idea to have your Front-End Servers sitting in different datacenters (and acting as different "entry points" to your clients), as shown on Fig VI.9:

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

— Wikipedia —

Fig VI.9



RARELY WORTH THE TROUBLE

The idea here is pretty much like the one behind classical CDN: to reduce latencies

for end-users. On the other hand, we need to note that

**unlike classical CDN, the content with our game-sorta-CDN is not static, so gain in latencies is possible only because of better peering, with gains usually being in single-digit milliseconds**

There is still a different reason to use such deployment architectures – in case if you want to protect yourself from Internet connectivity in your primary datacenter going down (provided that “Some Connectivity” survives); in practice, if you have a decent datacenter, it should never happen. More precisely – your datacenter WILL occasionally experience transient faults of around 1.5-2 minutes long (typical BGP convergence time), so if you’re looking for excuses to use this nice diagram on Fig VI.9 *and* your client can detect the fault and redirect to a different datacenter significantly faster than that, it MAY make some difference to your players.

Implementation-wise, there are several considerations for such CDN-like multi-datacenter Front-End Server configurations:

- you **MUST** have very good connectivity between your data centers (“some connectivity” on Fig. VI.9). At the very least, you should have inter-ISP peering explicitly set by both of your ISPs (to each other) to ensure the best data flow for this critical path
  - strictly speaking, “some connectivity” does not necessarily need to be Internet-based; you often can save additional few milliseconds by getting something like “dedicated” Frame-Relay between your datacenters, but this will likely cost you in the range of tens of thousands per month 😞 .
- traffic on “some connectivity” can be an order (or even two) of magnitude lower than that going to the clients due to Front-End Servers acting as “concentrators”
- you **SHOULD** account for secondary datacenter to go down (in particular, in case of inter-datacenter connectivity going down). The simplest way to deal with it is to have enough capacity in your primary datacenter (both traffic-wise and CPU-wise) to handle *all* of your clients, but this tends to be expensive. As an alternative, shutting down some activities in case of such a failure may be possible depending on specifics of your game.



**“CDN-like arrangements of Front-End Servers MAY save some of your players a few milliseconds in**

Bottom line for CDN-like arrangements. CDN-like arrangements of Front-End Servers may save some of your players a few milliseconds in latency (that is, if you have a really

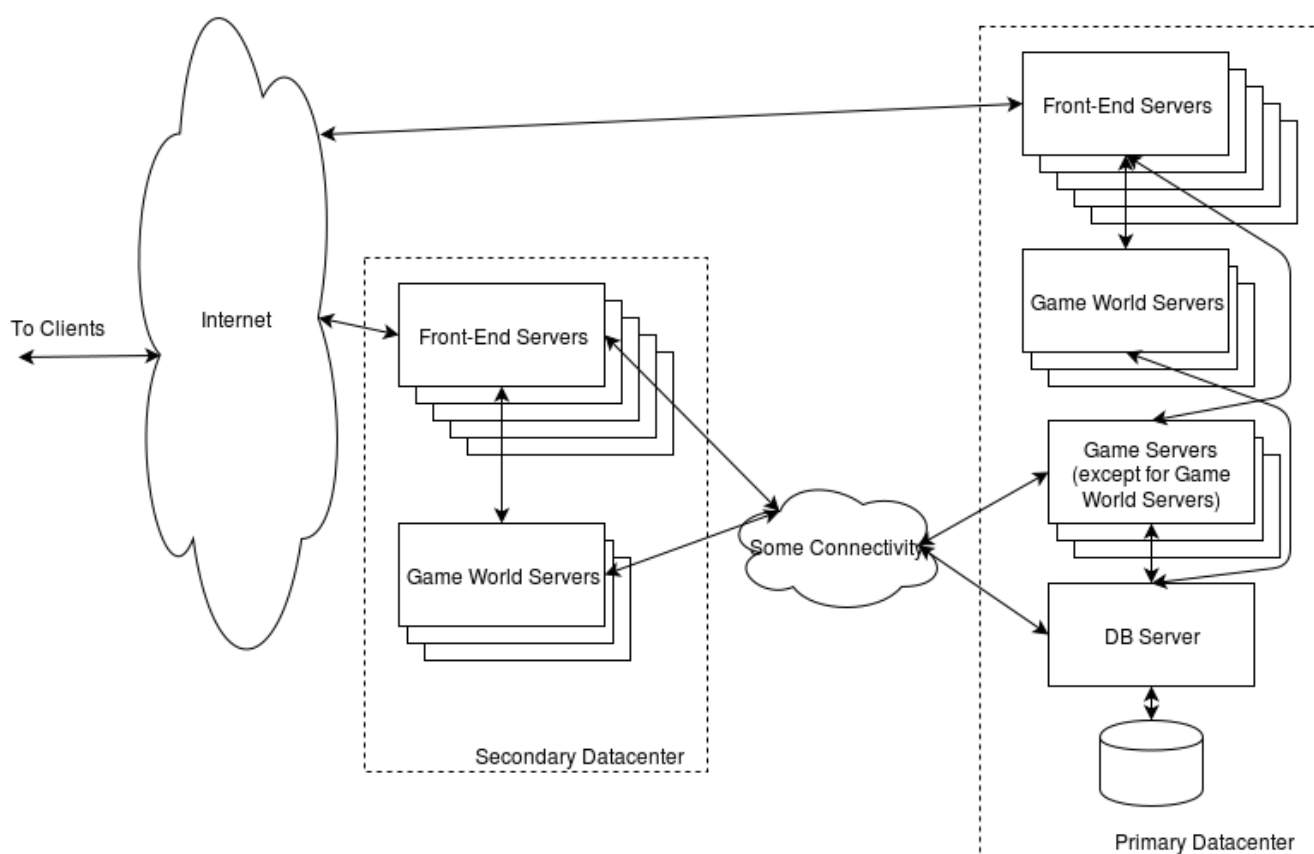
good connection between datacenters), which in turn may allow to level the field a bit with regards to latency. From my experience, it was hardly worth the trouble (because you cannot really improve MUCH in terms of latency, as the packets still need to go all the way to the Game Server and back), but keep the possibility in mind. For example, it may come handy in some really strange scenarios when you're legally required to keep your game servers in a strange location (hey casino guys!) where you simply don't have enough bandwidth to serve your clients directly.

**latency. From my experience, it was hardly worth the trouble**

## Front-End Servers + Game Servers as a kinda-CDN

On the other hand, if you're really concerned about latencies, it is usually much better to bring your Game World Servers closer to players (while leaving DB Server behind), as shown on Fig VI.10:

Fig VI.10



Here, we're moving the most time-critical stuff (which is usually your Game World Servers) towards the end-user, providing significantly better latencies to those players who're in the vicinity of corresponding datacenter. Maintaining such infrastructure is quite a Big Headache, but is doable, so if you're really concerned about latencies – you may want to deploy in such a manner. A word of caution – if going this way, you will end up with “regional servers”, which have their own share of troubles (you'll need to ensure that clients in the region go only to the relevant Front-End Servers, security on inter-datacenter connections becomes quite an

issue, etc., etc.); once again – it is doable, but go this way only if you *really* need it.

## On Affinity

In some cases, you may decide that you need to have a kind of “affinity” so that some specific players (usually those playing in a specific game world) are coming to specific Front-End Servers.



**“The things  
will go  
smoothly as  
long as the  
number of the  
game worlds  
which use  
affinity is  
small.**

Note when we’re speaking about our Front-End Servers, “affinity” is quite different from classical affinity (usually referred to as “persistence” or “stickiness”) used on load balancers for web servers. In the web world persistence/stickiness is about having the same client coming to the same server (to deal with sessions and per-client caches). For our Front-End Servers, however, affinity has a very different motivation, and is usually about Front-End-Server-to-game-world affinity (for players or for players+observers) rather than client-to-server affinity (see “Front-End Servers: Latencies and Inter-Player Latency Differences” section above for one reason where you MIGHT need such affinity).

Technically, implementing Front-End-Server-to-game-world-affinity is not *that* difficult, but the real problems will start *after* you deploy your affinity. In short – the things will go smoothly as long as the number of the game worlds which use affinity is small. On the other hand, as soon as you have a significant chunk of your players connected using the affinity rules, you will find that achieving reasonable load balance between different Front-End Servers becomes difficult 😞. When there is no affinity, the balance is near-perfect just because of the Law of Large Numbers; as you’re introducing the affinity rules, you’re starting to skew this near-perfectly-flat distribution, and the more players are affected by affinity, the more you’re deviating from the ideal distribution, so managing those rules while achieving load balance can become a Big Fat Challenge.

Bottom line: avoid affinity as long as possible (and most likely you will be able to get away without it).

## Front-End Servers: Implementation

Now let’s discuss ways how our Front-End Servers can be implemented. As mentioned above, the key property of our Front-End Servers is that they’re easily replaceable in case of failure. To achieve this behavior,

**you MUST ensure that there is NO original game-**

## world state on any of your Front-End Servers

**In other words, Front-End Servers should have only a replica of the original game-world state, with the original game-world state kept by Game Servers**

There is no need to worry too much about it if you're using a generic subscriber/publisher (or state replication) kind of stuff, but be extremely careful if you're introducing any custom logic to your Front-End Servers, because you may lose the all-important "easily replaceable" property above. See Chapter [[TODO]] for further discussion of this potential issue.

### Front-End Servers: QnFSM Implementation

One implementation of the Front-End Server implemented under pure Queues-and-FSMs architecture (see Chapter V for details on QnFSM, state machines, and queues) is shown on Fig VI.10:

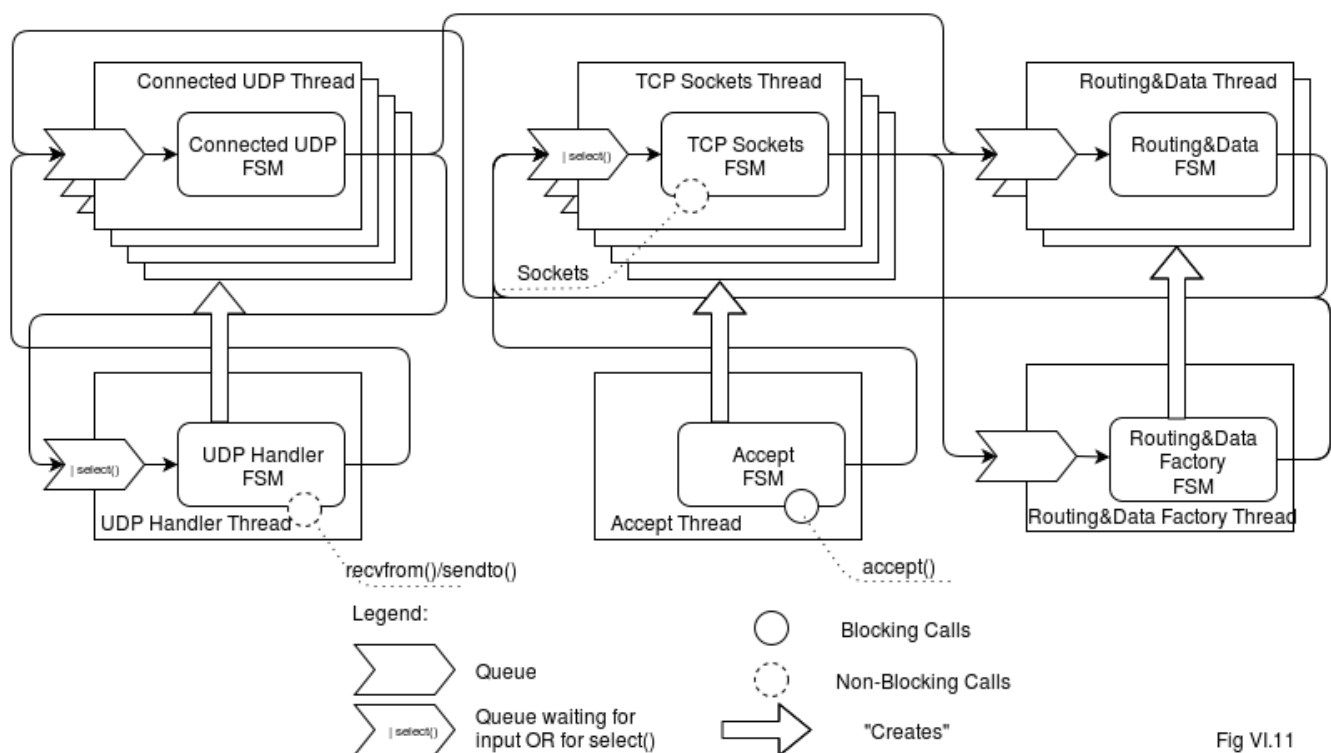


Fig VI.11

Here, we have TCP- and UDP-related threads similar to those described in "Implementing Game Servers under QnFSM architecture" section above with regards to Game Servers, and one or more of Routing&Data Threads (with at least one Routing&Data FSM each), which are responsible for routing of all the packets, and for caching the data (such as "game world" data). Let's discuss these routing-related FSMs in a bit more detail.

**Routing&Data FSMs.** Each of Routing&Data FSMs has its own data that it handles (and updates if applicable). For example, one such Routing&Data FSM may contain



a state of one game world. Other Routing&Data FSMs may handle routing of the point-to-point packets from players to (and from) one specific Game Server. Further details of the data types handled by Routing&Data FSMs will be discussed in Chapter [[TODO]], but generally there will be three different types of Routing&Data FSMs:

- generic connection handlers (to handle point-to-point communications including player input and server-to-server connections)
- generic publisher/subscriber handlers (to cache and handle generic but structured data such as a list of available games, if players are allowed to select the game)
- specific game world handlers (to cache and handle game world data if the required functionality doesn't fit into generic handler). In many cases you'll be able to live without specific game world handlers, but if you want to implement some kind of server-side filtering, like server-side fog-of-war to avoid sending data to those players who shouldn't see it (so no hack of the client can possibly lift fog-of-war) – specific game world handlers become a necessity.

It is possible (and often advisable) to have more than one Routing&Data FSM within single Routing&Data Thread to reduce unnecessary load due to an exceedingly high number of threads (and unnecessary thread context switches). How to combine those Routing&Data FSMs into specific threads – depends on your game significantly, but usually generic connection handlers are extremely fast and all of them can be combined in one thread. As for generic publisher/subscriber and specific game world handlers, their distribution into different threads should take into account typical load and allowed latencies. The rule of thumb is (as usual) the following: the more FSMs per thread – the more latency and the less thread-related overhead; unfortunately, the rest depends too much on specifics of your game to discuss it here.

**Routing&Data Factory Thread.** Routing&Data Factory Thread is responsible for creating Routing&Data Threads (and Routing&Data FSMs), according to requests coming from TCP/UDP threads. A typical life cycle of Routing&Data FSM may look as follows:

- One of TCP/UDP FSMs needs to route some message (or to provide synchronization to some state), and realizes that it has no data on Routing&Data FSM, which it needs to route the message to, in its own cache.
- TCP/UDP FSM sends a request to Routing&Data Factory FSM



**“It is possible (and often advisable) to have more than one Routing&Data FSM within single Routing&Data Thread**

- Factory FSM creates Routing&Data Thread (with an appropriate Routing&Data FSM)
- Factory FSM reports ID of the Queue, where the messages towards appropriate Routing&Data FSM should be sent, back to the requesting TCP/UDP Thread
  - TCP/UDP FSM (the one mentioned above) sends the message to the appropriate Queue (using ID rather than pointer to enable deterministic “recording”/”replay”, see Chapter V for details).
- Whenever the Routing&Data FSM is no longer necessary for its purposes, TCP/UDP FSM reports it to the Factory FSM
  - if it was the last TCP/UDP FSM which needs this Routing&Data FSM, Factory FSM may instruct appropriate Routing&Data Thread to destroy the Routing&Data FSM

## Routing&Data FSMs in Game Servers and Clients

I need to confess that personally I am *positively in love* these Routing&Data FSMs. I *love* them so much that I usually have not only on Front-End Servers, but also on Game Servers, and on Clients too; while they’re not strictly necessary there (and are not shown on appropriate diagrams to avoid unnecessary clutter), they did help me to simplify things quite a bit, making all the communications very uniform. Still, it is pretty much your choice if you want to have Routing&Data stuff on your Game Servers and/or Clients.

## Front-End Servers Summary

To summarize the section on Front-End Servers:

- As a rule of thumb, Front-End Servers are a Good Thing™. In particular:
  - they take the load off your Game Servers
    - which often makes the system cheaper (as Front-End Servers are cheap)
    - and also improves overall system reliability (as Front-End Servers are easily replaceable)
  - they facilitate single client connection (which is generally a good thing to have, see Chapter [TODO] for further discussion)
  - they facilitate client-side load balancing
  - they allow to handle 100’000+ observers for your Big Event easily (actually, the sky is the limit)
  - their drawbacks are pretty much limited to the additional latency, and this additional latency is firmly in sub-millisecond range



**“As a rule of thumb, Front-End Servers are a Good Thing™.**

- Client-side load balancing usually is the best one for games
  - one potential exception is Web-Based Deployment Architectures, where you MAY need server-side balancers
  - large-scale affinity is to be avoided
- CDN-like arrangements are possible, but not without caveats
- Front-End Servers can (and IMHO SHOULD) be implemented in QnFSM architecture, as described above

## [[To Be Continued...



This concludes beta Chapter VI(b) from the upcoming book “Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)”. Stay tuned for beta Chapter VI(c), “Modular Architecture: Server-Side. Eternal Windows-vs-Linux Debate.]]

## [–] References

[Larsen2007] Steen Larsen, “[Architectural Breakdown of End-to-End Latency in a TCP/IP Network](#)”

## Acknowledgement

Cartoons by Sergey Gordeev<sup>RL</sup> from [Gordeev Animation Graphics](#), Prague.

« ***[Chapter VI\(a\). Server-Side MMO Architecture. Naïve, Web-Ba...](#)***

***[MMOG Server-Side. Eternal Linux-vs-Windows Debate](#)*** »

*Filed Under:* [Distributed Systems](#), [Programming](#), [System Architecture](#)

*Tagged With:* [deployment](#), [game](#), [multi-player](#), [server](#)

Copyright © 2014-2016 ITHare.com