SmartFoxServer 2X Server Architecture White Paper

April 2012

Author

Marco Lapi
The gotoAndPlay() Team

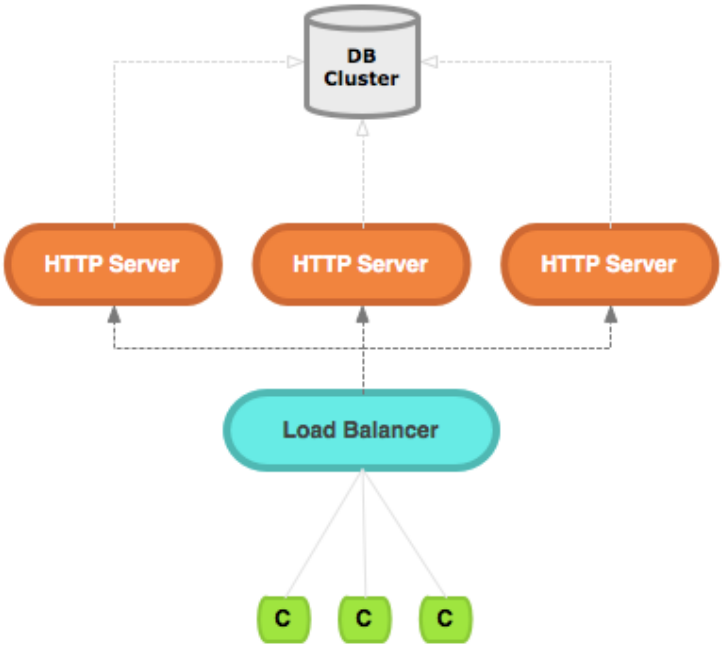
Introduction

In this document we will take an in-depth look at the best practices for deploying a multi-instance game server and the main strategies to simplify management and reduce costs. One of the crucial elements in creating a successful multiplayer on-line application is to avoid overly complicated server architectures which can become too complex to manage or too expensive to maintain.

» Why multiple instances?

The idea of running multiple instances of an application server is common in almost all network-based services. A recurring example is an http/web service that requires to handle a considerable amount of clients. All major online shops, portals, social networks, etc, are based on large server arrays that split the load efficiently, allowing hundreds of thousands of users to enjoy the web application without noticeable slowdowns.

The following diagram illustrates in a simplistic way how this is done: one ore more load balancers are exposed as the main entry point for the website which in turn will find a free web server inside the array and direct the user request to that machine. At this point the content is generated by the application server and delivered to the client.



Our diagram doesn't include some other important elements such as the details of the "DB Cluster" and the cacheing system which is usually found in these architectures, but this is beyond the scope of our paper.

The ability to choose a different HTTP server on each user request can be very effective in terms of load-balancing. Typically a user will request a page and then spend several seconds to few minutes browsing its content, reading descriptions, watching pictures, etc. When the client gets back with a new request (suppose he clicks on a product) the workload on the same HTTP server might have changed significantly. The load balancer will again check which machine is currently available and direct the user to that one in a completely transparent way.
This “transparency” effect is achieved by the fact that, most of the times, each HTTP request is opened, executed and closed allowing each new call to be run on a different application server.
Finally, web-apps usually keep a state for each client (e.g. the shopping cart) and this is handled by using a “session cookie” and by storing the user’s data in the central database. This way each HTTP server in the cluster is able to retrieve the data associated with each client and provide a seamless interaction even when the user is jumping from one machine to the other on each request.

» Sticky sessions

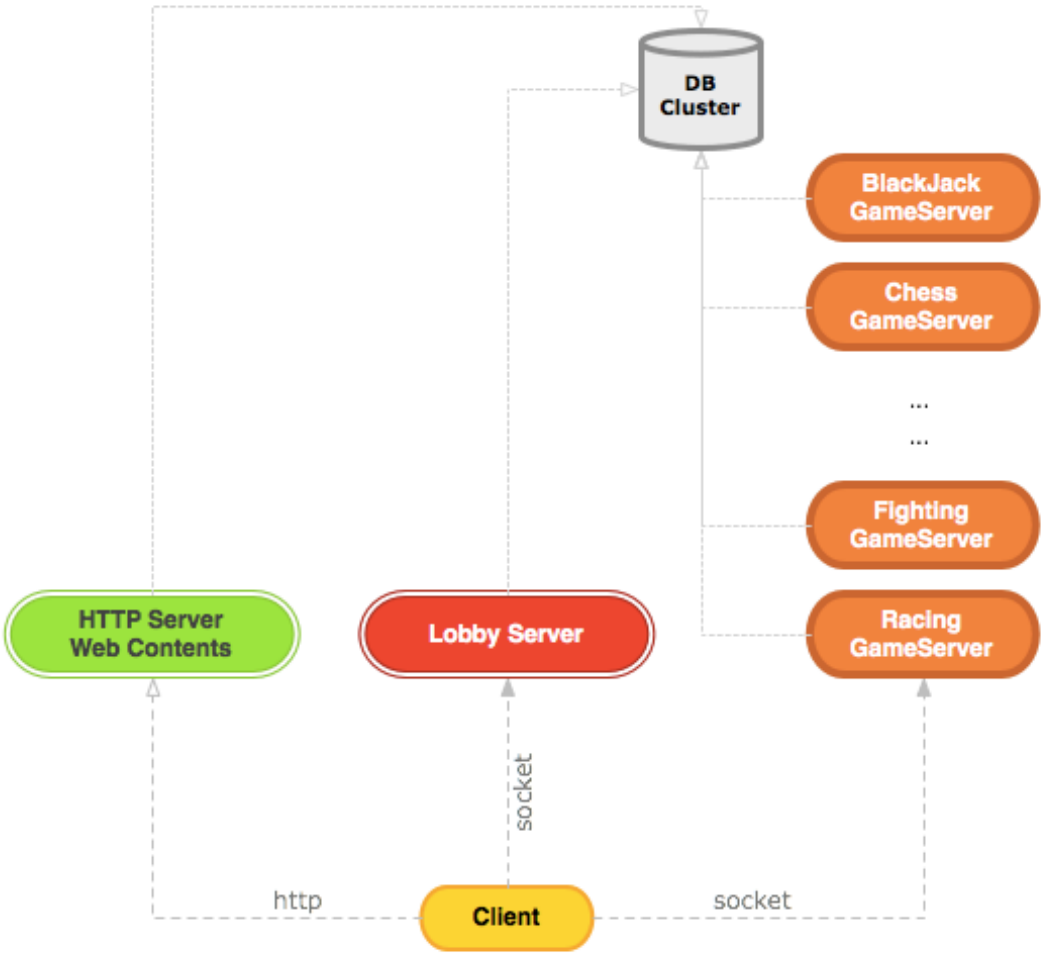
One fundamental difference between the HTTP/web world and multiplayer game servers such as SmartFoxServer, is the usage of persistent connections instead of open/close-type connections.

This technique provides a huge benefit in response times by minimizing the lag between client and server: in fact the cost of opening and closing the connection each time, as with HTTP protocol, would add too much delay and severely limit the game responsiveness. The approach of using a single persistent connection is also known as “sticky sessions” to indicate that the client’s connection is *stuck* to the server machine which it is connected to.

This small difference in the connection method involves a radical change in the approach to multiple-instances deployments because we loose the ability to dynamically redirect the client to any server in the array, so new strategies for load balancing must be employed.

Separation of concerns: Lobby and Games

The first solution we are going to explore offers the ability to create a multi-server architecture where the main lobby and each game run on separate dedicated servers. The system is particularly well suited for game portals providing several multiplayer games and with a continuos turnaround of games added every few weeks/months.



In this solution the load balancing is obtained by separation of concerns: each server in the system is responsible for one component of the whole gaming system. The client side application will provide the aggregation of the Lobby and the games by using two connections: one to the main Lobby server and one for the Game server, offering a seamless experience to the user.

The central database (or database cluster) will provide the main storage for user credentials, profiles, status, high score tables, rankings, etc. Also the web/HTTP part of the system will be able to access the data and publish it as dynamic HTML content to provide further integration.

Main **advantages** of this solution are:

- new games can be added easily without downtimes and with a relatively small expense (new virtual or physical server);
- ability to implement the system in the cloud with additional costs reduction and ease of deployment;
- in the cloud each Game/Lobby server can be resized to accommodate changes in the traffic;
- seamless integration between web contents and multiplayer side of the system;
- relatively simple management of the system and reduced maintenance costs.

Possible **improvements** could include:

- a second, passive, Lobby server used for high availability in case the main Lobby server should fail;
- multiple web/HTTP servers with load-balancing for high web traffic and improved reliability;
- use of a database/storage cluster for maximum storage availability and security.

In systems where the number of games is very high (suppose >= 100) a different approach should be employed to distribute the applications. Instead of using one dedicated server per game, games could be run in groups of 4-5 per instance making sure to aggregate those that are less demanding and leaving the most resource intensive ones on dedicated machines.

SmartFoxServer 2X provides great tools for fine-tuning the traffic in the Lobby server: each Zone and Room event can be configured separately, Rooms can be aggregated in separate groups to avoid massive Room lists and User count updates can be throttled to reduce excessive client network events.

In addition the SFS2X API provide high-level tools for sophisticated match-making, creating game challenges, sending invitations to multiple friends and creating custom permissions profiles for accessing different areas of your application.

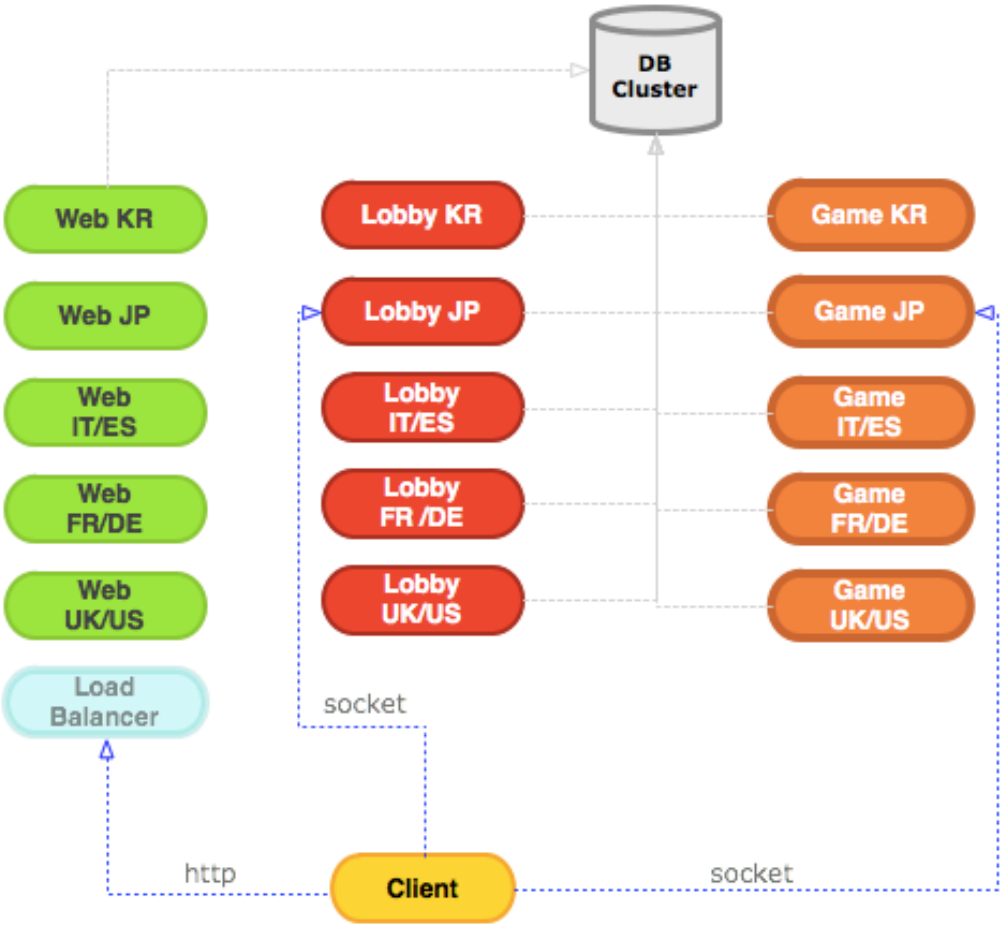
» Multiple lobby variant

A possible variation on the same structure can be applied to ultra large traffic systems where one machine for the Lobby is not be enough or simply not recommendable.

Suppose a world-wide, multi-language gaming portal targeted at users from Europe, America and Asia. In this situation it would be advisable to expand the Lobby component

to a multi-server architecture and implement a load balancing system based on language or region of origin.

Depending on the requirements users could be automatically directed to their language-specific Lobby or, in alternative, they could choose their destination from a selection of places. With the same approach very high traffic games could be further split on several machines in order to obtain the best load distribution



In this diagram we have aggregated some of the lobbies on the same machine depending on the amount of traffic generated by each country of origin.

As we mentioned previously, the load balancing in the system is obtained via a client-server interaction in which the user is recognized by its geographical location and automatically redirected to the correct servers.
In alternative the choice can be left to the players by providing a selection screen before joining the lobby and game servers.

Before opting for a solution like this you should make sure that at least these requirements are met:

- the need for a worldwide service supporting several different languages;
- vast traffic in the order of several hundreds of thousands CCU or even millions;
- large team (15+ people) with specialized personnel for server maintenance and management;
- ability to sustain a consistent initial investment ($250K-$500K at least);
- comprehensive business model including multiple premium account options, subscription fees, micro-transactions, merchandising, etc;
- deployment in the cloud for optimal scalability and ease of management (not mandatory but highly recommended).

In conclusion we would like to reiterate that this is just one example based on the client's country of origin, but there are plenty more possible solutions that can be adopted as rules for proper load distribution. In the end it all depends on the nature of the project and its main design characteristics.

What we are proposing in this document is that the architecture of the application and the related scalability issues should be analyzed side by side at design time to avoid excessive complexity and to simplify the implementation and maintenance where possible. In our experience, over the course of the last eight years, we have consistently noticed how the the most successful projects were exactly following these principles.

Virtual World topography

Massive multiplayer virtual worlds and communities provide an interesting challenge in terms of server architecture. Every MMO can have very different requirements, for example you can think of games such as Habbo Hotel, Disney's Club Penguin, Eve Online, Second Life. Each product has very unique features, very different demographics and technical prerequisites.

The challenge of building the back-end architecture for this kind of application requires that the world's design and the server structure are carefully planned in advance. Sadly the two things are not always analyzed and developed together. There are several "horror stories" in this field where problems are discovered in production because nobody took the time to evaluate the scalability issues.

To avoid these outcomes and many other obstacles we cannot stress enough how important is proper planning of the world, relative server architecture and appropriate testing. Also there are a few myths that we would like to dispel.

1 » The "Clustering solves all problems" myth
The idea that adding more servers in a cluster can solve scalability issues is often overrated. Besides the type of clustering technology or API in use, the main issue is that the data in local memory on one server must be kept in synch with all the other machines

in the cluster. If accessing the RAM is a pretty fast operation (few nanoseconds) the same cannot be said for data that is accessed over the network (few milliseconds). In other words **there can be five orders of magnitude** of difference when comparing network access vs memory access. This in turn can introduce huge bottlenecks and drastically limit the scalability of the system. Additionally these bottlenecks are very hard to predict and test in an asynchronous, multi-threaded environment and they usually show up in critical moments where lots of users are pounding on the system.

2 » The "All Users must see all" myth
Several customers have come to us asking how they could create a system that allowed each user to see and interact with all other clients in their MMO.
This feature can be very challenging to implement in a distributed environment because the state of Zones, Rooms and Users must be continuously synchronized between cluster nodes while the application runs. These data sets are complex tree-like structures with many levels of depth and multiple branches.
Cluster nodes synchronization issues apart, when the CCU is in the hundreds of thousands it is obviously impossible to handle that much information when received by a single client. Think about a list of thousands users continuously updated as they come and go, or thousands of avatars on the same map: interaction would be impossible! For this reason partitioning is always necessary and you should find a good strategy to divide the virtual world in several areas that can be mapped to physical servers.

3 » The "Linear scalability" myth
Not only adding more nodes in a cluster can magnify the data-replication problem we have just mentioned, but it can also degrade the current overall performance and response time of the system.
When this happens in a live environment it can lead to a downward spiral: in order to combat the performance degradation cacheing is introduced in the system adding a whole new layer of complexity. This in turn can lead to more management headaches, difficulties in debugging and isolating problems, etc.
The bottom line is that horizontal linear scalability is the holy grail of networking and **it can be achieved** by working with hardware and resource limitations as allied instead of enemies.

User can have a great experience if the virtual world structure provides a logical topology and easily navigable structure. Server-wide buddy lists can be employed to give the user a sense of "vastness" in their world while avoiding to overwhelm the client application with tons of useless data.
A central database is typically the core of the storage system, providing a unified repository for all game servers about player's online status, current location and other relevant runtime data. This in turn can be used to provide further sense of unification to the players logged in the MMO.

» Load balancing the virtual world

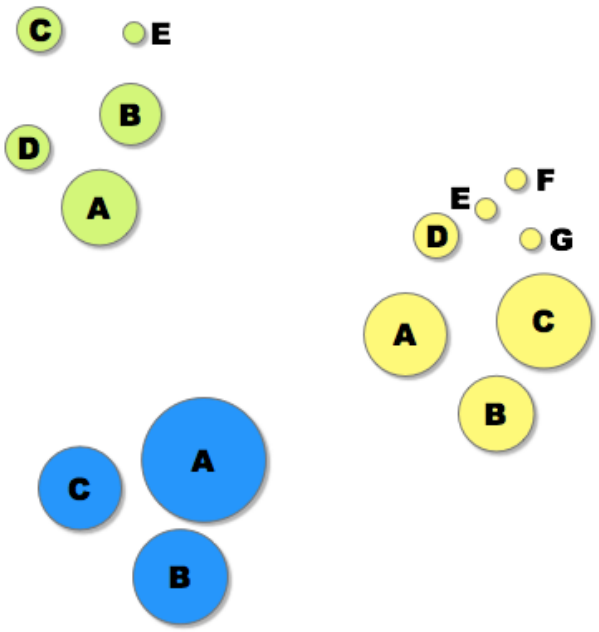
With an approach similar to the one described at the beginning we can create several ways of distributing the load in a massive multiplayer community.

In the first example we will take in consideration a virtual world based in space where many star systems are populated with planets offering many locations to visit.
Each location is a graphic avatar-chat like environment where users can interact, browse shopping catalogues, search for friends, customize their aspect, etc.
Also we can devote an entire planet to a specific game, so users can take a trip to planet X or Y in order to play with friends.

With this simple structure we can build a very complex virtual world, rich in features and games while maintaining a relatively painless architecture. Each *planet* will map to a Zone in the Server and each *location* can be represented by a Room. We can then proceed to group a number of planets into a *solar-system* which will be represented by one dedicated server (physical or virtual).

Again, the computing cloud may come to rescue with its ability to dynamically allocate resources: as we watch the virtual environment grow we will be able to size each *star-system* in accord with their popularity without the need to perform any physical configuration or switching hardware.

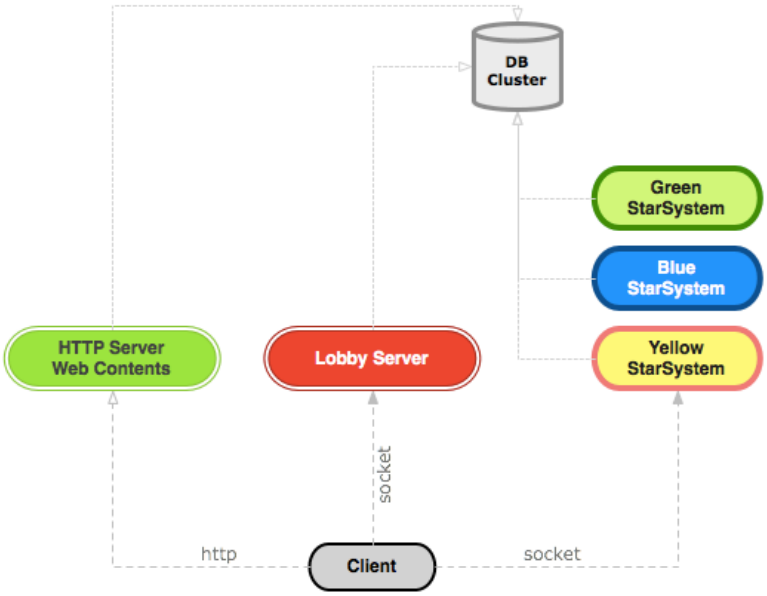
Finally this structure allows us to add several new *planets* to existing *star-systems* or even whole new *star-systems* with minimal to no impact on the game cluster.



This is an example of three *star-systems* with the relative planets.

- **Yellow**: each planet represents a completely different environment with several locations that can be visited with each user's avatar. Here clients will be able to chat, show off their avatars, buy and sell items and buy a place that can be customized with furniture and gadgets.

- **Green**: in this system we run a quest-oriented game where users must go around exploring each areas of each planet in search for clues and items to solve a number of mini-games.
- **Blue**: it is the gaming *star-system* with each planet running a different game.



Frequently asked questions

In closing we would like to address some common questions often asked by people approaching the multiplayer games development for the first time.

Q: We expect millions of concurrent players, will SmartFoxServer handle such traffic?

We are often asked how the server will behave under the most unreal loads.
It would be really great if a web-based multiplayer game could reach those peaks of popularity, however it is very important to make realistic estimates.

As of today (April 2012) most of the popular and trafficked gaming portals very rarely peak at 50 thousand simultaneous players. Large web-based MMO communities claim to have millions of active users but those numbers refers to the registered members returning to the website within a certain period of time. The amount of users all connected at once (concurrent users) rarely tops 100K even for the largest games and virtual worlds out there.
To give you an idea Second Life reached 75K+ simultaneous at peak times during 2008 with an average of 55K users between 2007 and 2008. Another very popular MMORPG, Eve Online, has recently broken a new user record at around 65K concurrent players (details here).

As we have explained in this white paper, if the virtual world architecture is planned carefully and ahead of time there will be very little problems with scaling the system to higher loads when necessary. At the same time there's no need to invest millions when starting up, a few server instances will be able to keep up for a long time as the user base grows. For more details on SmartFoxServer 2X's performance you can read the **SmartFoxServer 2X Performance and Scalability** white paper on this page.

Q: Our website has millions of registered users, how will the server scale?

Again, it's important not to confuse your website statistics with the number of real concurrent users at any one time. They are very different numbers. A very rough calculations of possible concurrent users would be: **website_unique_visits_per_day / 100 (or 200)**. So if you approximately receive 1.000.000 visits a day you could probably have a peak traffic of 5.000 to 10.000 concurrent players.

Q: Ok, but how is the server going to scale with my application?

It is impossible to provide an estimate without knowing what the application does, how complex it is, the type of hardware in use, the available resources, the quality of your code.

We provide the engine, on top of which you can build your car but the details of the final product are in your project. To expand this metaphor further we don't know about the materials you will use, the type of tyre, brakes, transmission you will employ, etc. All we can do is providing one of the best and scalable engine that will help you create your ideas quickly and reliably.

Again, for more details on SmartFoxServer 2X's performance you can read the **SmartFoxServer 2X Performance and Scalability** white paper on this page.

Q: What about Terracotta Clustering for SmartFoxServer 2X?

The integration between Terracotta 2.5 and SmartFoxServer PRO was done in 2008 and provided interesting results. Unfortunately Terracotta has taken a different direction with version 3.0, targeting large enterprises and offering their cacheing and "Big Memory" services while the DSO (Distributed Shared Object) was almost abandoned. In fact you can find very limited documentation about the DSO and the only book available, written by the Terracotta Team, is four years old and covers only version 2.5 which is not supported anymore.

Also, the usage of DSO is discouraged in several places in the Terracotta website:

"The threshold for successfully setting up a DSO cluster can be substantially higher than for a non-DSO cluster due to DSO's stricter code and configuration requirements. It is recommended that if possible you use the standard installation (also called express

installation) to set up a non-DSO cluster. Use the DSO installation only if your deployment requires the features of DSO."

And...

"Clusters based on the standard installation are much simpler and more flexible than those based on the DSO installation."

In part we understand their position about the DSO and the additional complexity that it introduces in terms of development especially with synchronization, class loading and memory management.
On the other hand the new Terracotta 3.x cacheing system based on EHCache isn't flexible enough to provide transparent clustering of the complex data structures that govern the SmartFoxServer runtime. Thus the decision not to support it anymore.