Clustering SmartFoxServer

Integrating the Terracotta technology with SmartFoxServer PRO to build highly available and scalable multiplayer services.

(c) 2008 gotoAndPlay() -- www.smartfoxserver.com - www.gotoandplay.it -- All rights reserved

Table of contents

Introduction 1

» What is clustering 1

» Objectives 1

» Prerequisites 1

The Study 2

» Sharing data across the cluster 3

» Introducing Terracotta 3

» Why using Terracotta 5

» Integrating SmartFoxServer and Terracotta 5

The implementation 8

» Login phase 10

» Lobby: the friend list 13

» Lobby: the chat 18

» Game servers 20

» Lobby: the game list 22

» Lobby: playing games 23

» Performance considerations 25

» Monitoring the cluster 25

» Handling Terracotta failures 26

» Conclusions 27

Introduction

In this white-paper we will take an in depth look at building highly available **SmartFoxServer** clusters, with no single point of failure, high horizontal scalability and failure resilience.

The study presented in this document is particularly suited for MMO communities, virtual worlds and large gaming systems, where a high server capacity and quick crash recovery are determinant for the success of the service.

» What is clustering

Generally speaking a server cluster is a group of computers working together to improve the capacity and availability of one ore more services. Each computer in the cluster is usually referred to as a "node", and nodes can have different roles in the cluster depending on its architecture. For example there can be generic server nodes, gateway nodes, cache nodes, backup nodes etc...

From a client side perspective, the cluster is usually seen as a single computer / entity.
If you want to learn more about clustering in general, we recommend checking this wikipedia page.

» Objectives

The objective of this study is to build the prototype for an on-line multiplayer gaming service based on **SmartFoxServer** that provides high availability, failure resilience and the ability to scale horizontally as the traffic and popularity of the application increases.

In the following chapters we will examine the architecture of the service, the technologies being used and we will analyze the server-side code used in the study.

» Prerequisites

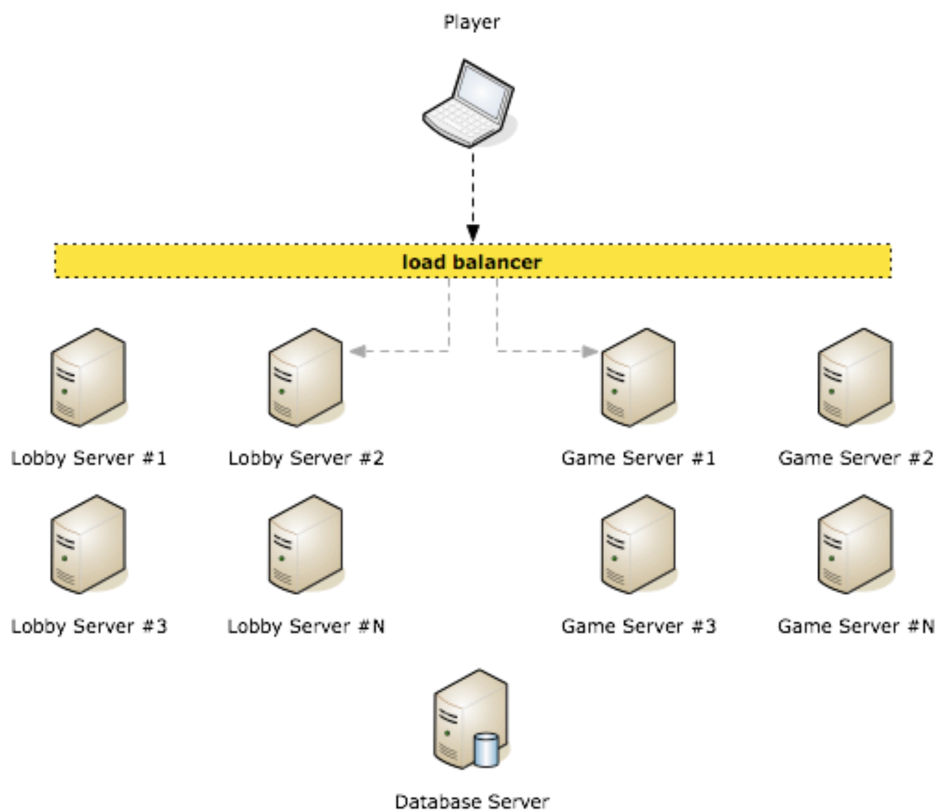
This white paper is structured into two main sections: the first one is an introduction of the concepts and techniques used to build a clustered service, while the second section is more technical and goes into the development details and relative code.

For the second part of this study it is required a good understanding of the **SmartFoxServer** technology and experience with Java programming and general networking concepts.

The Study

The multiplayer gaming service that we're going to discuss consists of a central Lobby, where all players are initially connected, and a number of multiplayer games that users can join.

The following diagram outlines the structure of the system:



The cluster will be made up of a number of server machines, dedicated to handling the Lobby traffic, and at least one server for each game available in the website.

All these computers will share some of the runtime data, to allow users connected to any lobby or game node to see each others as if they were connected to a single multiplayer server.

The ***load balancer*** outlined in the diagram will take care of transparently connecting users to the least trafficked node, ensuring and even distribution of the load across all machines.

Finally the ***database server*** will take care of persistent data such as player login credentials, high scores and rankings, buddy lists etc... The database server could be a cluster itself in order to cope with a high number of transactions.

» Sharing data across the cluster

One of the critical problems with clustering technologies, is keeping the application state replicated across all the nodes. From the previous diagram we can see that player can be connected to any lobby node, but we need them to "see" each others in order simulate a single big server.

This implies that cluster nodes communicate together, in order to allow users residing on one machine to talk with users from another one. Additionally, changes happening in one node that interest the global state of the application, will need to be replicated consistently across the cluster.

Synchronizing the state of a distributed application has been addressed in various ways depending on the type of service. Among the solutions to this issue there are:

1. continuously broadcasting state changes to all cluster nodes
2. using expensive database clusters for keeping the state in a central place
3. using one or more central cache servers (e.g. memcached) to keep the application state
4. registry-gateway solutions (a-la Flash Media Server edge/origin) where the application logic runs on a single machine surrounded by multiple gateways that handle the I/O work

While any of these solutions has its own advantages, each one also presents one or more downsides that could represent a show stopper. For example solution #1 will probably fail to scale linearly, solution #2 will probably cost too much money and #3 could suffer from the same problems of #1.

The 4th solution is not even a real distributed implementation, as it simply decouples the I/O workload from the application logic, which still remains confined to one single server. If the registry machine collapses the whole service becomes unavailable.

This last approach can be effective for media servers such as Adobe FMS (Flash Media Server), where the gateways also act as a cache for media content, but it's not recommended for an MMO gaming system.

» Introducing Terracotta

One of the most interesting approaches to clustering is the one provided by Terracotta (www.terracotta.org), one of the leading and most promising technologies for clustering and distributed computing for the Java runtime.

Terracotta provides clustering services at the JVM level, working behind the scenes and without interfering with the application code itself. In other words the distributed application does not depend on any new framework and there are no new dependencies or libraries to learn.

From the words of its creators, Terracotta "*provides network attached memory to the JVM*" allowing to distribute a single Java application over multiple virtual machines as if they were one.

To learn more about the many features and capabilities of Terracotta, we recommend the following online resources:

Terracotta on Wikipedia

Terracotta official website

Introduction to Terracotta

Figure 2 illustrates the concept of network-attached-memory: multiple copies of the same application run in different JVMs, all coordinated by the Terracotta server.

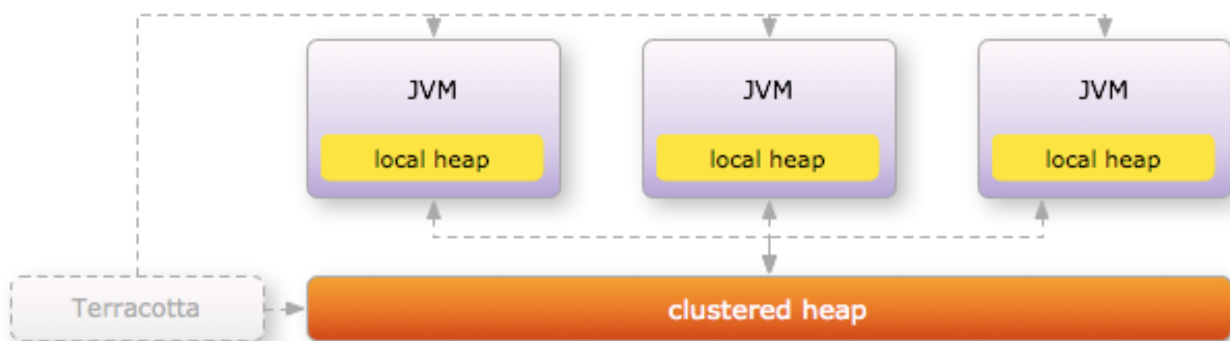


figure 2: shows the concept of network-attached-memory

The process for declaring which objects should be distributed across the cluster is very simple and effective: each Terracotta application comes with an xml configuration file, in which the developer can specify which objects should be seen cluster-wide. (these are referred to as “root-objects” or just “roots”).

Once the application is running, the Terracotta server will work behind the scenes keeping all nodes updated about changes in the shared objects, and performing all sorts of optimization to reduce the local network traffic.

The great advantage of using Terracotta, is the complete transparency for the developer, and the ability to use the “regular” Java semantics for data access and synchronization across the cluster. For example, a synchronized block on a shared object will acquire a cluster-wide lock (a Terracotta transaction) just as it would acquire a memory lock if it was running in a single JVM.

Finally it is important to mention the Terracotta Server, which acts as an orchestra conductor for all connected virtual machines in the cluster, managing the synchronization between nodes, fine-grained data replication, cluster-wide transactions, object identity, garbage collection and more...

The TC Server can also persist the distributed data on disk without impacting the overall performance, and it can be configured in “active-passive” mode to provide full failover support.

» Why using Terracotta

This is a brief summary of the reasons why we chose Terracotta to provide the clustering infrastructure for **SmartFoxServer**:

- **Transparent clustering**: Terracotta is not a framework and it doesn't add new libraries or dependencies to your code. It works behind the scenes, at byte-code level, providing distributed heap memory across multiple JVMs and allowing developers to use the standard Java semantics for synchronizing access to shared objects.

  Follow this link for more informations

- **Unprecedented scalability**: Terracotta allows impressive horizontal scalability, by simply adding more server nodes in the cluster and by eliminating the bottlenecks and inefficiencies of using big databases and caches, for managing the shared data.

  Follow this link for more informations

- **No single point of failure**: highly available systems need to be resilient to failures and be quick to recover from a node crash. Running multiple instances of the Terracotta Server in *active-passive* mode, guarantees that a failure won't compromise the cluster: as soon as the main server fails one of the passive instances immediately takes control, and starts from where the previous server has left.

- **Open source and cost effective**: Terracotta is an open source project that comes with a friendly licensing system based on the Mozilla Public License. The use of Terracotta doesn't provide any additional costs to your project. Finally creating a distributed system of enterprise level is not going to cost you millions!

  Follow this link for more informations

» Integrating SmartFoxServer and Terracotta

One of the most practical and effective ways of integrating the two technologies, is by using Terracotta to distribute the server extension data across multiple cluster nodes.

Since 95% of the application logic resides in the custom server extension, we can clearly start to see the benefits of being able to **distribute the extension state** across multiple SmartFoxServer nodes.

Before we discuss how exactly Terracotta fits in the mix, let's do a small step backward and see how you would "traditionally" approach a distributed system.
Let's imagine we have a multi-player Lobby where users can login and consult a list of running game rooms divided by game type: chess, blackjack, space-war, racing game etc... Also let's say we have four physical servers running this application.

What you would normally do is find a solution for sharing the state of all these game tables in a central place, so that each instance of the server can read and update the state of the Lobby and its game rooms.

The following diagram illustrates this scenario in detail:

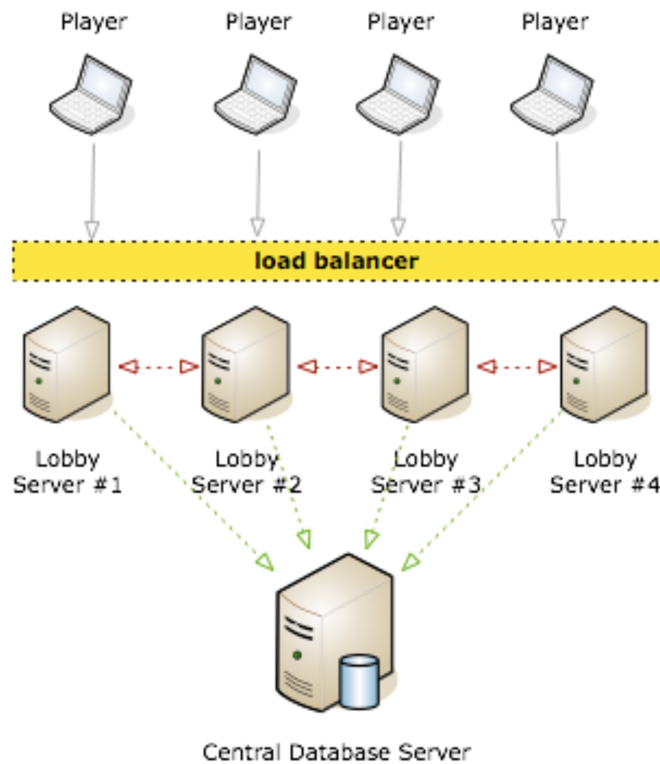


figure 3: shows a database-centric cluster architecture

The green connections from the Lobby servers show that each SmartFoxServer instance is talking to the central database to continuously retrieve and update the state of the games.

Additionally we also would like that users in the Lobby could communicate with each others as if they were all connected to a single big server. Of course this is not possible right out of the box, as each server instance is not aware of what is happening “outside” of it.

To solve this issue, we could use a messaging technology such as JMS, to allow each instance send and receive events from other servers (red connections).

This solution clearly brings us some problems:

- the central database could represent both a single-point-of-failure and a bottleneck in the whole system.
- the central database could cost a lot of money as it is required to deliver a very high transaction rate to keep up with the client requests.
- we need to introduce and learn at least one additional technology for the server-to-server messaging.

By moving from a database-centric architecture to Terracotta, we can solve all these issues at once and provide much better overall performance and scalability.

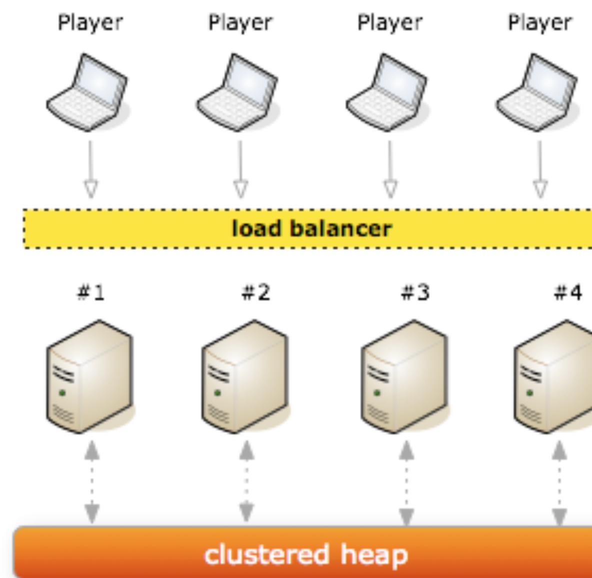


figure 4: shows a database-centric cluster architecture

The ability to transparently share data (regular Java objects, or POJOs) across multiple JVMs, provides the developer with new "super-powers" that no other clustering approaches can deliver. We can share entire collections of objects graphs and still retain outstanding performance, thanks to the fine-grained update capabilities of the Terracotta Server.

Also, we don't incur in the extra overhead of serializing / de-serializing objects during each cluster transaction, and only the fields that were changed are actually transferred.

The messaging system that we mentioned earlier can be implemented very easily by simply sharing a queue in the distributed memory. More specifically, all we have to do is sharing a Map of BlockingQueue objects, and run one or more threads in each JVM to handle the messages passed in the queue.

Now each server instance can send and receive messages by simply accessing the shared Map of queues. (We'll see this mechanism more in details in the later chapters)

It is also important to note that Terracotta allows fine-grained configuration of the shared objects, (enabling the developer to include/exclude fields from any object that is going to be distributed) and great control over cluster-wide object locking and synchronization.

The implementation

In this section we are taking a closer look at the application and its main features, starting from the user interface.The login screen will be the first dialogue window presented to the client. Behind the scenes the server has already routed the client application to one of the cluster's node, according to the load-balancing logic (more on this in a moment).

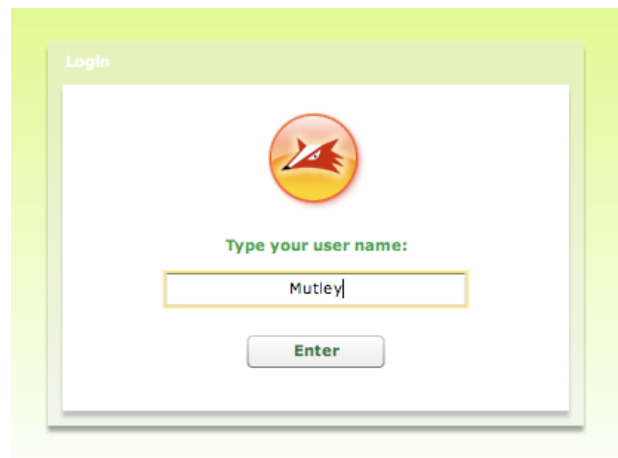


figure 5: the login screen

Upon successful, login the client is sent to the main lobby as shown in figure 6.
The lobby presents the following user controls:

- **Friend List**: this is were we keep track of our buddy-list and where we see who is online. By selecting one name from the list we can start a new one on one chat.
- **Private Chat Box**: this is the history of each private chat during the current session
- **Game List**: it shows the list of available games for the currently selected game type
- **Game Types**: the selector allows to see running games for all available game servers
- **Add Friend**: from here the user can add more friends to his buddy list
- **Send PM**: sends a private message to the currently selected friend
- **Debug infos**: shows the client/server messages (this is just for debugging purposes)
- **Create game**: from here the user can create or join a new game room for the currently selected game type

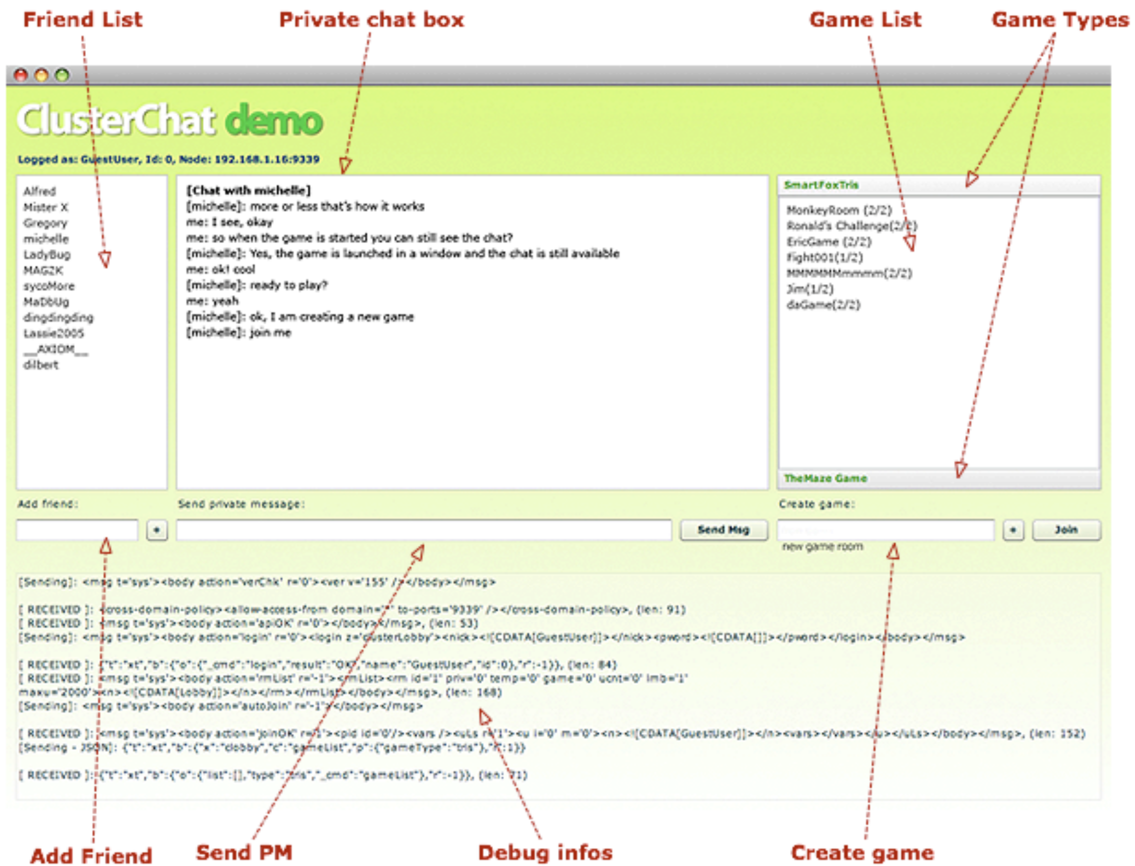
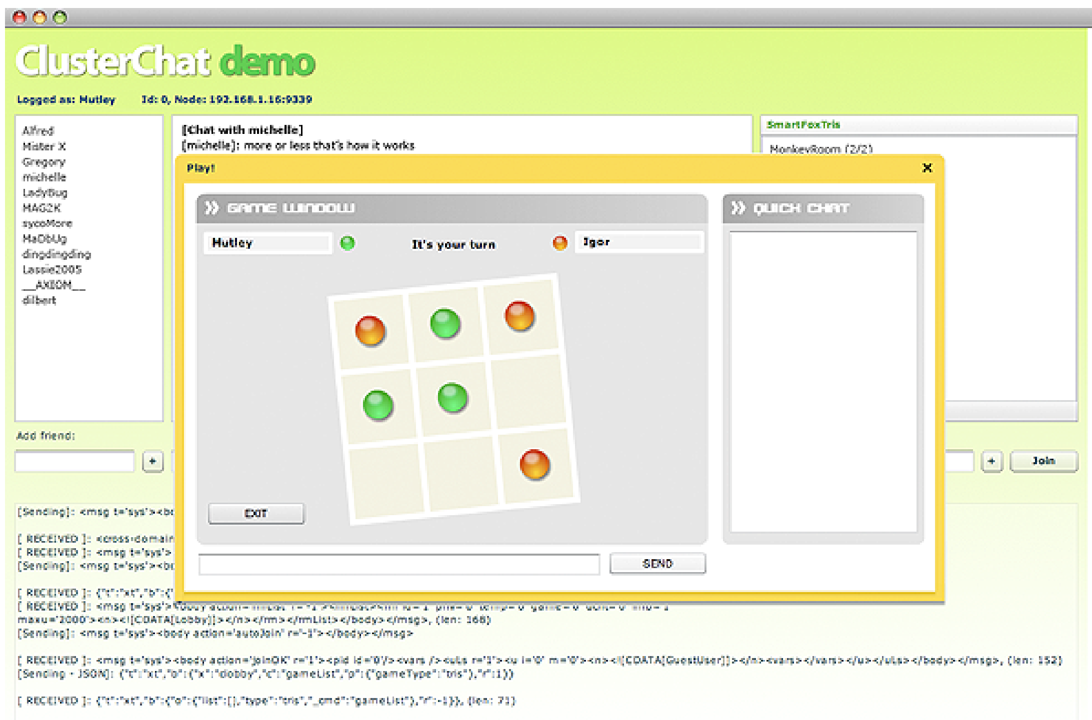


figure 6: the main lobby window

Finally when the user joins or creates a game, a new window is displayed inside the lobby allowing the client to play the game and interact in the Lobby at the same time.



» Login phase

We mentioned earlier that the login phase will transparently connect the user to the proper Lobby cluster node, ensuring an even distribution of clients in the system.

In order to keep track of the current load of each server we'll share the current number of users in each node in the cluster, and we'll keep it updated as new users come and go.
The simplest way to achieve this is by using a map (a key-value pair) of counters: in Java this could be done with a Map<String, AtomicInteger> object.

The key used in the map is the id of the node, which is constructed when the node starts by using it's IP address and port number. For example if the node runs on 192.168.0.100, port 9339 the unique string will be: "192.168.0.100:9339".

» How does this really work?

At this point you're probably wondering how the process of "sharing" data in the cluster actually works from a code standpoint. For this example we'll use a **DataStore** class that encapsulates all the collections and objects that we're going to share cluster-wide.

The **DataStore** class is implemented as a Singleton, ensuring that only one instance is ever created.

```
package sfs.cluster.lobby;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

public class DataStore
{
	private static DataStore _instance;

	//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
	// Cluster wide objects (roots)
	//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
	public Map<String, AtomicInteger> nodeUserCounts;

	// Private constructor
	private DataStore()
	{
		if (nodeUserCounts == null)
			nodeUserCounts = new ConcurrentHashMap<String, AtomicInteger>();
	}
```

```
		public static DataStore instance()
		{
			if (_instance == null)
				_instance = new DataStore();

			return _instance;
		}

	Map<String, AtomicInteger> getNodeUserCounts()
	{
		return _instance.nodeUserCounts;
	}
}
```

The second step for sharing this class across the cluster is to simply declare it in the Terracotta configuration file:

```
...
...
<roots>
	<root>
		<field-name>sfs.cluster.lobby.DataStore.nodeUserCounts</field-name>
	</root>
</roots>
...
...
```

The **<roots>** node allows us to specify which objects we want to share: in this case we have specified the fully qualified name of our Map. It should be now clearer why the instantiation of the Map happens only when its reference is null: we really just want to initialize this object once, but if we run the same application across multiple JVMs, we will end up recreating that object each time a new node is started. This would obviously delete all existing data.

Here is where Terracotta does the “magic”: once the first JVM has created the **nodeUserCounts** object, all the other nodes in the cluster will “see” it and skip the instantiation.

Now that we have a better idea of how the distributed objects work, we can go back to our load-balancing implementation. We will use the **SmartFoxServer embedded http-server** to handle an initial handshake between the client application and the cluster. The java servlet will choose the right node by checking all the available socket servers, and returning the IP address of the least trafficked machine in the system.

The client will then connect to the server returned by the cluster node.

In order to avoid that all user connect to the same server for the initial handshake, we will pass the client application a list of available server nodes (i.e. via an xml file).

The configuration file could look like this:

```
<cluster-config>

	<zoneName>clusterLobby</zoneName>
	<extensionName>clobby</extensionName>

	<gateways>
		<gateway port="8080">10.11.12.1</gateway>
		<gateway port="8080">10.11.12.2</gateway>
		<gateway port="8080">10.11.12.3</gateway>
		<gateway port="8080">10.11.12.4</gateway>
		<gateway port="8080">10.11.12.5</gateway>
		<gateway port="8080">10.11.12.6</gateway>
	</gateways>

	<games>
		<game id="tris">SmartFoxTris</game>
		<game id="maze">TheMaze Game</game>
	</games>

</cluster-config>
```

We have also added a few more informations like the name of the zone to connect to, and the id of the server side extension.

The **<gateways>** indicate valid SmartFoxServer nodes that also run the http-server for the initial load-balancer handshake.

The Flash client can simply choose one of these addresses randomly, and perform the handshake. In case something goes wrong we can try another address and finally fail in case none of the machines responds.

The login flow can be summarized with the diagram at figure 8.

In normal conditions we wouldn't expect that the the http-server doesn't respond, however we should always take into consideration that one node in the cluster might fail, so it is better to try a new machine in case the handshake is not successful.

Later in this white-paper we will also discuss how to monitor the state of the cluster.

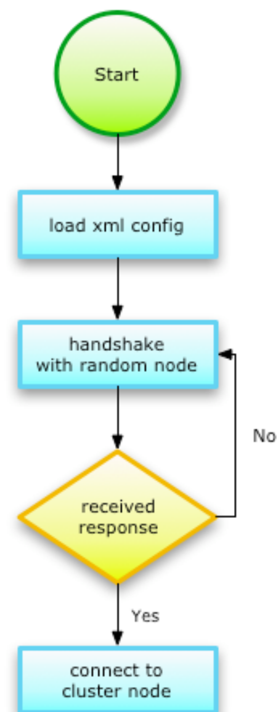


figure 7: the initial handshake

» Lobby: the friend list

The Lobby is designed to be a central place where all users are initially joined, so in a large cluster we could expect form tens to hundreds of thousands of connected players.

For this very reason, public messages are of course not allowed in this central room, which has been configure as a Limbo-type room.
Users will be able to communicate by adding their buddies to a friend list, start IM-like conversations, check the on-line status of their friends etc...

A search engine would be also necessary to allow clients to search for buddies using various criteria, such as location, age, gender, interests etc...

We're not going to analyze how this search system can be implemented, as it would out of the scope of this document, however it wouldn't be particularly difficult since we have already introduced a central database in our cluster architecture.

The buddy-list system has been rewritten from scratch, instead of using the SmartFoxServer built-in functionalities. This decision has been taken for a number of reasons:

- The embedded buddy-list system was not designed for clustering
- The persistence of the buddy-list system is not centralized (each server instance would only save the list of those users that are physically connected to the node)
- We thought it would be interesting to rewrite it for a clustered environment as this would emphasize some very important aspects of distributed programming (cluster-wide events etc...)

The new system will be referred to as "friend-list" to differentiate it from the embedded buddy-list system.

At the core of the friend-list we have a class called **FriendListManager** which encapsulates a Map<String, List<Friend>> for holding the global data.

The class exposes methods like **addFriend()**, **removeFriend()** and **getClientsToUpdate()**, which determines the clients to be updated when a user logs in or out of the Lobby.

Here are a few excerpts from this class:

```
public class FriendListManager
{
	Map<String, List<Friend>> friendLists;
	Zone zone;
	ClusterLobbyExtension parentExtension;

	public FriendListManager(Zone zone, AbstractExtension parentExtension)
	{
		if (friendLists == null)
			friendLists = new ConcurrentHashMap<String, List<Friend>>();

		this.zone = zone;
		this.parentExtension = (ClusterLobbyExtension) parentExtension;
	}
```

The **friendList** Map is the Terracotta “root” of our friend-list system, and we’ll share it in the cluster by adding a new declaration in the Terracotta xml configuration.

```
<root>
	<field-name>sfs.cluster.lobby.friendlist.FriendListManager.friendLists</field-name>
</root>
```

For the map implementation we are using a java.util.ConcurrentHashMap which will handle thread-safety for us behind the scenes. Please note that the List<Friend> will not be thread-safe so we will take care of its synchronization manually (an alternative would be to use a CopyOnWriteArrayList)

The **Friend** class is very simple:

```
package sfs.cluster.lobby.friendlist;

public class Friend
{
	String name;
	String nodeId;

	public Friend(String name, String nodeId)
	{
		this.name = name;
		this.nodeId = nodeId;
	}

	// getters/setters follow...
```

We store the name of the user as its unique id, and the name of the cluster node in which the user is connected.

In order to keep track of each user-status, we will also need to know the full list of users connected in the cluster. This data is not available out of the box, since the user lists live isolated in each SmartFoxServer node. We can easily create the full list of clients by sharing a collection in the distributed heap, and update it when a user logs-in or out of the service.

The collection can be added to the **DataStore** class, by declaring this:

```
public Map<String, ClusterUser> clusterUsersByName;
```

In order to keep this Map updated, we will listen for user connections / disconnections on the server side, and add/remove **clusterUser** objects as needed.
Additionally on every user log-in/log-out we need to fire an event to all cluster nodes, so that they update their clients accordingly.

In other words each time user "Mutley" logs in/out, we need that any other connected user, who has "Mutley" in its friend-list, gets an update about the status change.

This is the flow that we need to implement when "Mutley" logs in/out of the system:

1. handle the server side event (handleInternalEvent)
2. update the global user list (clusterUsersByName)
3. fire an event to all nodes notifying that "Mutley" has logged in the system
4. each node will look for users interested in this change and fire an update to those clients

Tasks 1 and 2 are very simple and we already know how to perform them, the third step however seems a little more obscure at the moment, while step 4 is already implemented in the **FriendListManager** class.

Let's now concentrate on step 3 as we need to solve one important issue of our distributed system: the ability to exchange messages between the cluster nodes.

If you remember we talked about the use of **BlockingQueues**, in the Terracotta introduction, for creating a clustered event system. Now it is time to expand on this concept and see how simple it is to create such a mechanism with the standard Java library.

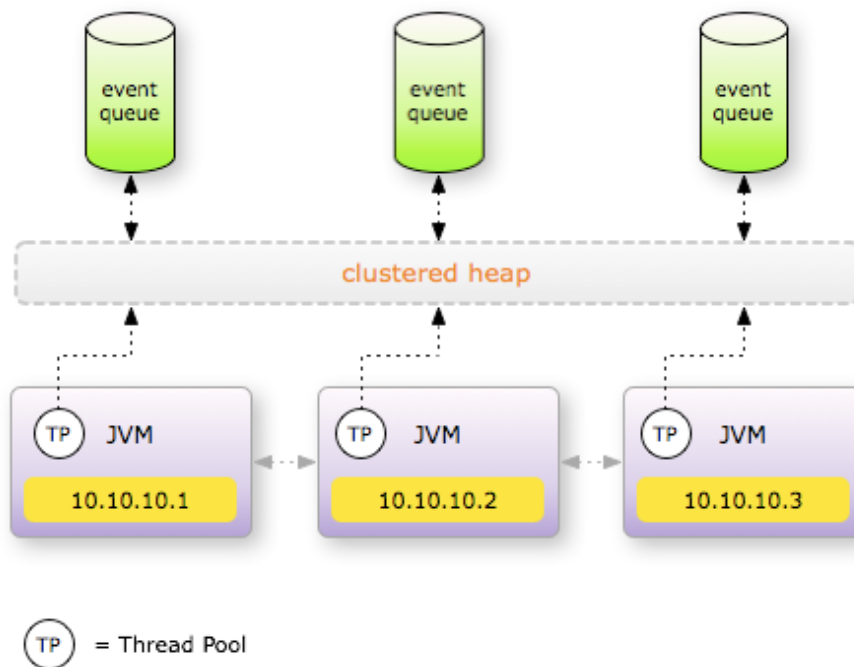


figure 8: the distributed event system, based on blocking queues

Figure 8 shows that each JVM shares a queue in the distributed heap, which is controlled by a number of threads in the local thread-pool. By using blocking-queues we can keep the threads asleep until an event becomes available.
Every time we need to fire an event to a specific node (or to all of them), we simply add a new object in the target queue, and the event will immediately wake up the local thread pool which will handle it.

In other words if JVM labeled **10.10.10.1** wants to send a message to JVM **10.10.10.3**, all it has to do is adding a new **Event object** in the target blocking queue. The rest of the "magic" happens behind the scenes thanks to Terracotta, which takes care of notifying the change in the queue and copying objects in the right places for us.

Notice that, from a developer standpoint, you don't have to learn any new framework or add dependencies to your project. Terracotta takes care of instrumenting your bytecode when the JVM starts up, and its Server will take care of monitoring and updating the object's states.

This is how the **Event** class looks like:

```
package sfs.cluster.lobby.data;

import java.util.HashMap;
import java.util.Map;

public class ClusterEvent
{
	private String eventName;
	private Map<String, Object> params;

	public ClusterEvent(String eventName)
	{
		this.eventName = eventName;
		this.params = new HashMap<String, Object>();
	}

	public String getEventName()
	{
		return eventName;
	}

	public Map<String, Object> getParams()
	{
		return params;
	}

	@Override
	public String toString()
	{
		return String.format("{ CLUSTER EVENT: %s -> %s }", eventName, params );
	}
}
```

The ClusterEvent object allows us to identify each event with a unique string, and pass an arbitrary number of parameters (as Objects) to the receiver.

Finally, we declare a Map<String, BlockingQueue<ClusterEvent>> in our **DataStore.** to collect the event queues and make them globally available in the cluster.

```
public Map<String, BlockingQueue<ClusterEvent>> postOffice;
...
...
if (postOffice == null)
		postOffice = new ConcurrentHashMap<String, BlockingQueue<ClusterEvent>>();
```

We thought of this map as a post office, where each node has its own mail-box and can drop messages in any of these boxes, hence the variable name.

» Lobby: the chat

Once our friend list is populated we will able to start a conversations with any of the online buddies. While this is probably one of the most simple feature to implement, we still have to take into account that users connected to a physical node won't be able to directly talk to a client on another machine. We still need to transfer the request to the target node and from there send the actual message through the receiver socket connection.

The flow for handling a cross-server message is the following:

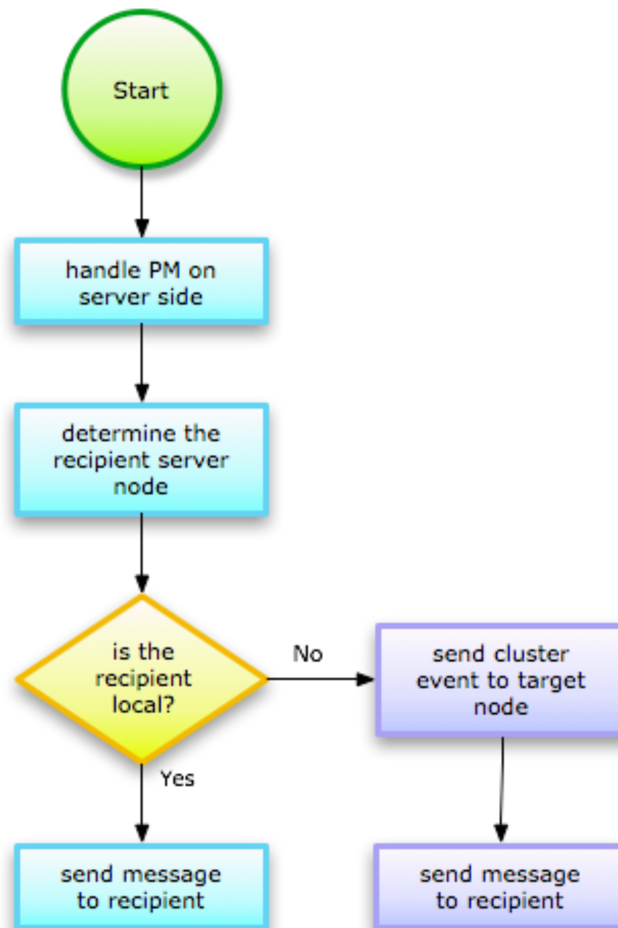


figure 9: the flow for sending a private message to any user connected in the cluster

Here's the code implementation:

```
public void handleInternalEvent(InternalEventObject ieo) throws Exception
      {
           String msg = ieo.getParam("msg");
           User sender = (User) ieo.getObject("sender");

           String[] params = msg.split(TOKEN);

           User recipient = parentExtension.getCurrentZone().getUserByName(params[0]);

           HashMap msgObj = new HashMap();
           msgObj.put("_cmd", "pm");
           msgObj.put("text", params[1]);
           msgObj.put("sender", sender.getName());

           // send to local user
           if (recipient != null)
           {
             JSONObject jso = new JSONObject(msgObj);
             LinkedList localRecipients = new LinkedList();
             localRecipients.add(recipient.getChannel());

             parentExtension.sendResponse(jso, -1, null, localRecipients);
           }

           // send to remote user
           else
           {
             ClusterUser remoteRecipient = parentExtension.getUserByName(params[0]);

             if (remoteRecipient != null)
             {
                   ClusterEvent event = new ClusterEvent(ClusterEvent.ON_PRIVATE_MESSAGE);
                   event.getParams().put("jso", msgObj);
                   event.getParams().put("recipient", params[0]);

                   DataStore.instance().dispatchClusterEvent(event,
remoteRecipient.getNodeId());
                   }
           }
      }
```

From the client side we send the private message by joining the recipient name with the actual chat message, using a separator. (referred in the code by the TOKEN constant)
The separator is an unprintable character, with ASCII code == 0x01.

First of all the received string is split into its two components, the name and the chat message. Then we check if the target user is connected to the same server of the sender: if so we immediately dispatch the message, otherwise we fire a cluster event to the node where the recipient is connected.

The event is then handled on that node and the message is finally sent to its destination.

» Game servers

Each game available in the system will run on its own dedicated server, and it will be part of the cluster in order to interact with the data store. Game servers will use a different extension from the Lobby since they don't need to know what is going on in that room.

We will run two different extensions in each game node:

1. **Zone-Level game-lobby extension**: this is responsible for logging players in the server and joining them in the game.

2. **Room-Level game extension**: this is the actual game code, where all the application logic will reside. Every game room will spawn its instance of this extension. Please note that, since we're using Java, we don't incur in the typical overhead of script-based extensions.

From the client perspective we will be using a second socket connection to the game server. The reason why we opted for this solution is to get the best performance possible in terms of lag and response times.

Distributing the game state over multiple servers is also an option, but it introduces an extra (unavoidable) lag in the system, due to the updates that keep the game state in synch.

To better understand these design choices we can take a look at the following diagram:

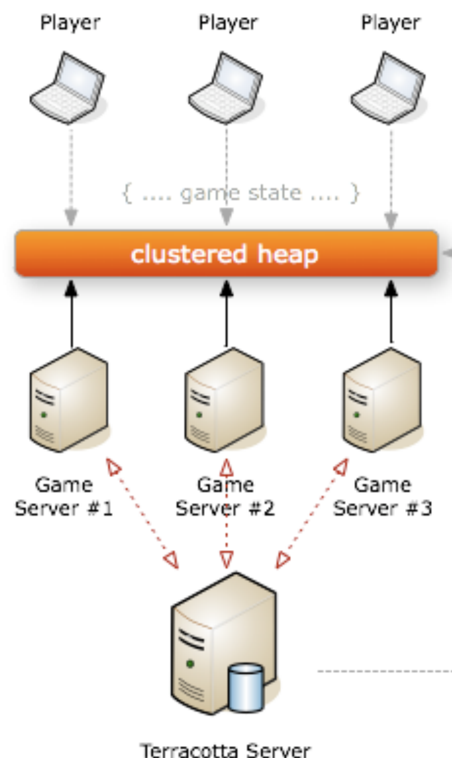


figure 10: sharing game state across multiple nodes

Figure 10 shows users connected to three different nodes playing the same game. This architecture has the great advantage of distributing the load over multiple machines.

Pros:

- can scale to the hundreds of thousands or even millions of concurrent players for a single game
- if one server goes down you just loose the players on that machine, not all

Cons:

- extra lag due to node synchronization could not be ideal for very fast real-time action games
- probably more expensive because you use more hardware for every game instance

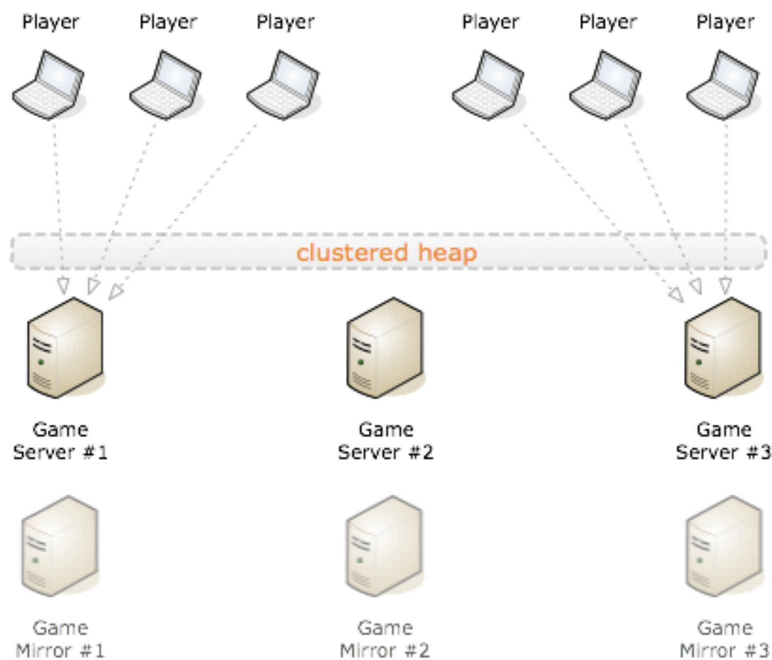


figure 11: game servers don't distribute the game state in the cluster

Figure 11 shows a different approach where every game is hosted on a dedicated server, and the game state is not shared across multiple nodes. Each machine is still part of the cluster, it can receive and fire cluster-wide events, and access the data store.

Pros:

- Great for all fast real-time multiplayer games
- Simplifies the architecture of the cluster when you don't need to handle enormous traffic on a single game

Cons:

- If the game server fails all players will be disconnected (even if they will still be connected to the central Lobby). They would be able to start playing again if you pair each game server with an idle instance that takes over when the master goes down.

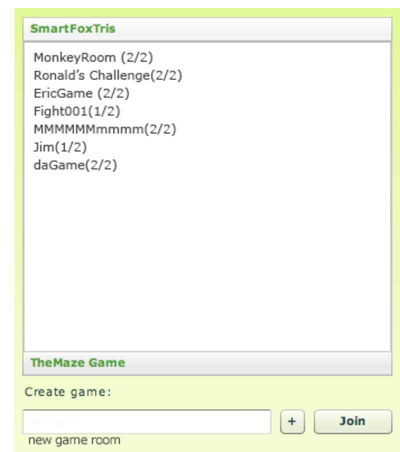
A third way is an extension of this last example where you use multiple server instances (no shared game state) to handle very high traffic loads (i.e. in the hundreds of thousands).

In conclusion we would like to point out that these three approaches **can be mixed together** in the cluster, providing endless possibilities to create highly available multiplayer games.

Also, from an economic standpoint, the ability to choose different types of architecture allows to start small and gradually invest more money in hardware and bandwidth, as the service becomes more popular and produces more incomes.

» Lobby: the game list

We can now proceed by examining how the game servers will share the status of their rooms in the Lobby. Each user should be able to choose one game from the Games Accordion component and see all the rooms available and relative number of players.



To enrich this section of the GUI we could also show more detailed informations about each game, when the user selects it. This would simply involve an extra request to the server to grab all the details about the selected game.

For the sake of simplicity we will just show the informations as depicted in figure 12.

In order to share the rooms from each game server we'll proceed in the same way of the friend list: we'll declare a new collection in the **DataStore** and share it with Terracotta by simply adding it to the list of roots in its configuration.

First of all we create a class for storing the relevant room informations:

```
class ClusterRoom
{
	private String name;
	private int localID;
	private String gameType;
	private int maxPlayers;
	private AtomicInteger playerCount;
	private boolean passwordProtected;

	public Map<String, Object> properties;

	public ClusterRoom(String name)
	{
		this.name = name;
		this.passwordProtected = false;
		this.playerCount = new AtomicInteger(0);
	}

	// getters and setters follow ...
```

Each ClusterRoom object is identified by a unique name, it has a localID (the id of the room in its server instance) and a gameType. Additionally we have added a flag for password protected rooms and a Map<String, Object> for storing custom properties.

In the DataStore we will add a collection of ClusterRoom objects where the String key is the name of the room.

```
public Map<String, ClusterRoom> clusterRoomsByName;
```

When a new room is created or removed in the system, the change is reflected in the **clusterRoomsByName** object, and a cluster event is fired. Each node will then broadcast an update to all its clients.

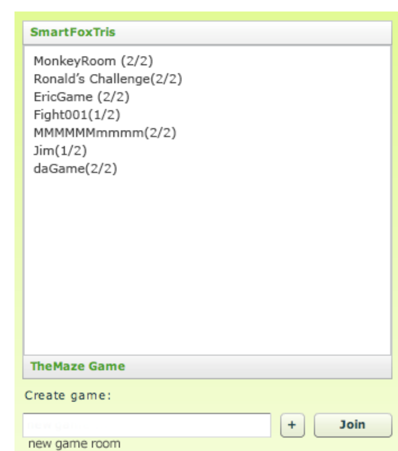
One interesting aspect of handling the room list, is how much traffic this will generate when thousands of rooms are available in the game server. Even by using a single dedicated machine for each game, we can definitely expect thousands of games running at the same time. In particular, the continuos update of the number of players inside each room would be very resource-intensive, if done in real-time.

In order to avoid wasting so much bandwidth, we have created a throttling system that aggregates changes for a certain amount of time and finally sends an update to the clients. This allows us to reduce the amount of message broadcasted, decreasing the network traffic significantly.

You can learn more about this throttling mechanism by checking the *sfs.cluster.lobby.GameListUpdater* class.

» Lobby: playing games

Users can browse the list of games by selecting a game title in the accordion: this will generate a request to the Lobby extension which in turn will respond with the list of available games. The player is then enabled to create new games or join an existing one from the same panel.



In the last section we have seen that users are connected to the appropriate game-server when they choose to play in a certain room. Now we are going to take a closer look at the mechanism that coordinates these actions.

The creation of a new game is pretty straightforward: we simply fire a cluster-event to the target game-server, which in turn will create the room (and dynamically attach the game extension to it). Once the new room is ready the player is automatically joined in it.

The join action is a little more complex, as we want to make sure that no other users enter that room in the meanwhile. In fact, starting a new connection to the game-server can take from less than a

second (on a fast network) to several seconds, so we have introduced a simple “reservation” system that allows players to ensure a slot in the room for a short interval of time (say 15 seconds).

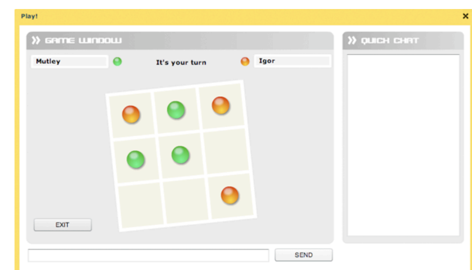
In short, this is the flow for joining a game room:

- User selects a room to join from the main GUI and presses the **Join** button
- A cluster-event is fired to the game-server
- If a slot is available in the room, it is reserved for 15 seconds to the requesting player
- The player is now connected to the game server
- Upon successful connection and login the game is launched on the client side
- The player is joined in the game

This is how the “reservation” class looks like:

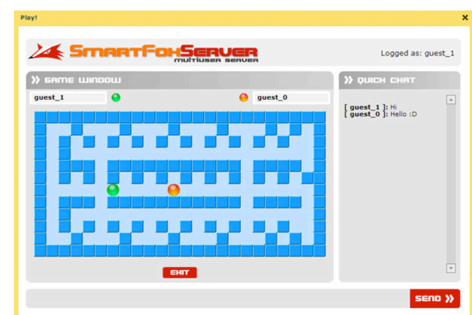
```
package sfs.cluster.lobby.data;

public class RoomReservation
{
	long creationTime;
	long expiryTime;
	String owner;
	int targetRoomId;

	public RoomReservation(String owner, int targetRoomId, int expiryTime)
	{
		this.owner = owner;
		this.targetRoomId = targetRoomId;
		this.creationTime = System.currentTimeMillis();
		this.expiryTime = this.creationTime + (expiryTime * 1000);
	}

	// getters / setters follow...
```

The **RoomReservation** object keeps track of the player’s name, the target room and the request time. Each of these objects are stored locally on the game-server and, upon user login, the server extension checks if the reservation exists.



This mechanism also prevents that any other user try to directly log into a game-server without an appropriate authorization.

On the client side we handle the socket connections separately: the main **SmartFoxClient** object, used for connecting to the Lobby, is instantiated in the main application class, while the games are stored in separate SWF files. Each game handles its connection to the game-server internally.



In our Flex Lobby prototype we used two games taken from the SmartFoxServer tutorials: SmartFoxTris (tic-tac-toe) and RealTimeMaze (real-time pacman like).

The advantage of this approach is that each game can be developed separately from the Lobby and doesn't depend on any specific resource of the parent container.

Additionally the games not necessarily need to be written in Flex, in fact the two example games were written in Flash CS3, tested separately (out of the cluster) and finally integrated in the system.

» Performance considerations

The idea of using new classes like **ClusterUser** and **ClusterRoom** for sharing user and room state may seem a little redundant, as each SmartFoxServer instance already exposes a User and Room class.

Actually, we chose this solution to minimize the amount of data being shared across the cluster, and just expose the essential fields we need to describe those entities. This reduces the internal traffic of the cluster and guarantees top performance and scalability as you add more machines in the system.

When designing applications that potentially could be hit by hundreds of thousands (or millions) of users, we have to carefully avoid every possible waste of resources in order for the system to scale.

In other words scalability just doesn't "happen" by simply organizing multiple servers into clusters. What we really need to do, is thinking and designing the application **to be scalable** from the very beginning and making sure we take into consideration all potential bottlenecks etc...

» Monitoring the cluster

In this section we're going to discuss what happens when one of the nodes in the cluster fails, and how we should handle the loss of the users connected to that server.

Terracotta provides a convenient method for notifying the connection and disconnection of a cluster node, via JMX events. This feature allows us to easily monitor the addition and removal of new nodes and react consequently.

For instance, if one Lobby node in our example cluster dies for whatever reason, we will need to clear all the relevant data in the **DataStore**, and update all the other clients in the system. This includes: updating the global user list, updating all the users' friend list, removing the user counts for that node and its "post office" mail box.

For this specific study we preferred to implement our own monitoring system, which is a simple Java application running in the cluster and monitoring all the connected nodes, at specific intervals. We opted for this solution in order to be able to provide a finer-grained level of monitoring:

1. check basic reach-ability of the node (ping)
2. check the availability of the SmartFoxServer instance
3. check / monitor specific functionalities of the server extension

With this solution the custom monitor agent (running in its own dedicated machine) can recognize server failures and also measure or log performance numbers, by directly talking to a custom extension as indicated at point 3.

Note: the monitoring and performance logging topic would involve a long dissertation, which is not in the scope of this study. In this section we are just mentioning a few of the many possible options.

When a failure is detected, the agent immediately starts the clean-up process and fires a cluster event to all other nodes. The client side update is optimized by preparing a single extension message, containing all the friend-list changes. This is sent in one shot instead of having each update being fired separately.

The failure of a game server is very similar to this scenario. In this case we will need to remove all the server's rooms from the data store and send an update the other clients.
The approach used for the update is the same: we group all the room details in one single message, and broadcast it to all interested users.

One optimization we have added to the mix, is that we keep track of each user's current game list. This allows us to send the update only to those clients that are listening for changes in the game server that died.

» Handling Terracotta failures

The Terracotta server is actually the most important component in our cluster, as it provides all the behind-the-curtain services that we have discussed in this chapter. Additionally it is responsible for keeping all the runtime data alive so... what is going to happen when the TC server dies?

Terracotta provides an "active-passive" mode where one server (**active**) runs together several other instances (**passives**). As soon as the active machine dies, one of the passives takes control starting exactly from where the other server left.

There are two ways to configure the Terracotta "active-passive" mode:

- **Active-Passive using shared disk**: in this configuration the server uses a shared disk between the active and passives to replicate the cluster state.
- **Active-Passive using network**: in this mode the server replicates the cluster state across the network.

If you want to learn more about all the details of the Terracotta configuration we recommend to visit this web page.

» Conclusions

The integration of **SmartFoxServer** with the Terracotta infrastructure, opens an unlimited amount of possibilities for creating the next-generation of highly scalable back-ends for massive web-based multiplayer games and applications.

The great flexibility provided by this combination allows anyone, from small startups to larger enterprises, to benefit of the clustering services at a fraction of the cost of traditional solutions, and with best overall performance.

Additionally the non-intrusive nature of Terracotta allows the developers to concentrate on their application logic, without the need of introducing and learning new dependencies.

Finally the future releases of **SmartFoxServer** will move towards a tighter integration with Terracotta, to provide even more distributed services out of the box, and further simplify the development.