SmartFoxServer 2X
Hazelcast Integration white paper

June 2016

Author

Marco Lapi
The gotoAndPlay() Team

Introduction

In this document we are going to discuss how to build fault-tolerant multiplayer games without single point of failure, using SmartFoxServer 2X and a distributed grid/data store tool such as Hazelcast.

Some of the most recurring questions on the subject of high availability are “How do we deal with server faults?” or “How do we make our game fault tolerant?”

In these pages we’ll attempt to answer these questions by proposing a relatively easy and generic architecture that works especially well with turn-based games, and suggesting ways of integrating other tools that help solving the problem.

It should be noted that a generic solution covering all possible use cases is not possible, instead we will pursue a more reasonable objective of proposing a simple, yet flexible design that can cover a large percentage of use cases.

What is Hazelcast

Hazelcast is an open source, in-memory data grid used for distributed computing.

It may sound like a mouthful but in essence Hazelcast is a convenient tool for managing distributed data in a cluster, allowing for replication and high availability (i.e. fault tolerant).

The main key words here are:

- **in memory**: meaning that data is read and written from (fast) RAM as opposed to (slow) disk storage
- **highly available:** meaning that our data is kept in multiple copies across several nodes in the cluster so that it can be resilient to failures

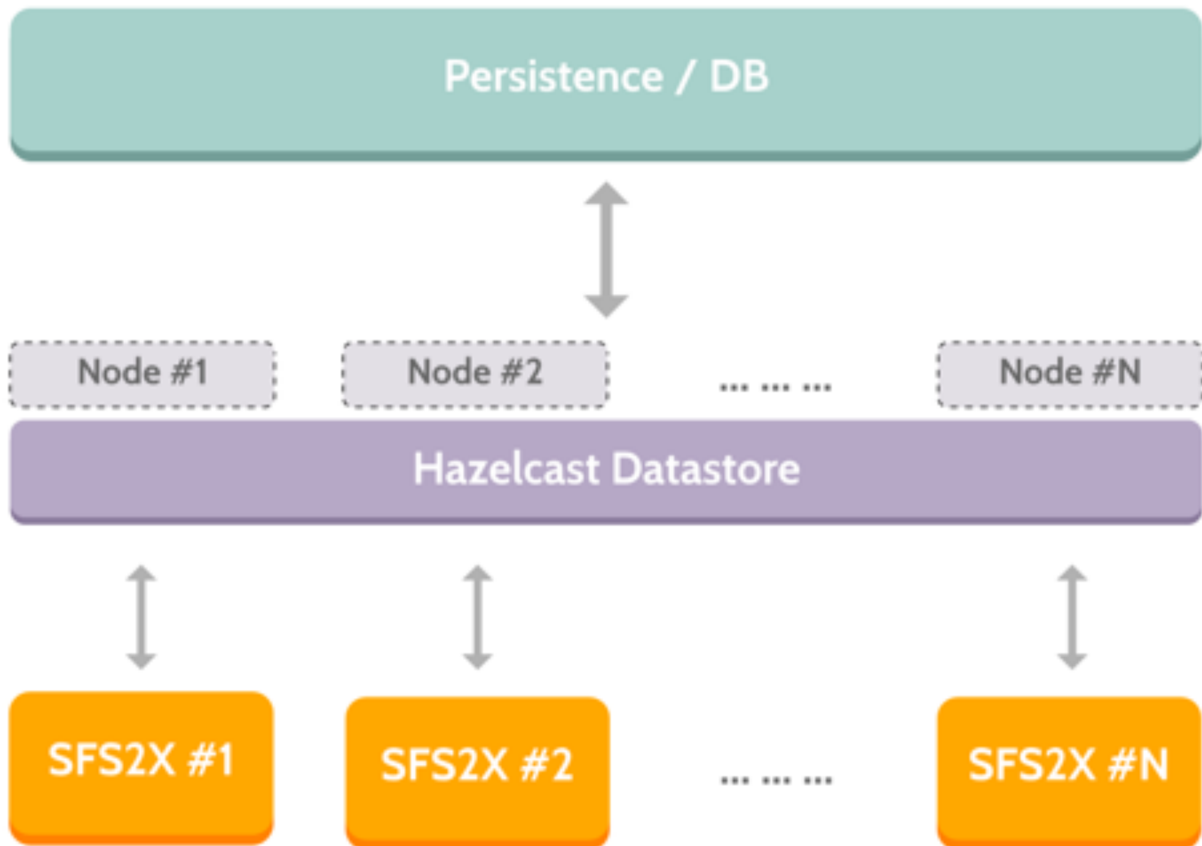
Also it is important to note that when we refer to “data” we are talking about any POJO (Plain Old Java Object) really, which makes it very convenient to use as opposed to standard RDBMS.

Why is this relevant, you may ask: if you’re familiar with our “SmartFoxServer Architecture” white paper published a few years ago, we usually suggest to implement multi-server architectures by running a central database (or database cluster) as the main repository for data shared between game servers.

Hazelcast can come in handy as a substitute for the central database providing higher scalability (faster read/write access), fault tolerance and offloading the database of potentially heavy traffic.

This doesn’t mean that our database needs to be ditched. Persistent storage is still a useful and needed requirement in most cases, but it can be relied upon less heavily and in exchange for more speed.

The following diagram should clarify what the proposed architecture should look like from a bird's eye perspective:



Multiple SmartFoxServer 2X instances can all be connected to Hazelcast to store and retrieve distributed data, for instance the state of a specific user or game.

From the "outside" the Hazelcast datastore looks like a single entity to every SFS instance and it can be scaled horizontally by simply adding more machines, when necessary.

Finally we have our persistence layer which, as usual, can be represented by one or more database engines.

» How does it work

Objects in Hazelcast are stored in simple, well-known data structures such as Map, List or Queue. The interesting part is that every entry in one of these structures is distributed in the cluster and backed up a number of times, providing extra reliability.

Nodes in the Hazelcast grid are connected in a peer-to-peer (P2P) network where more nodes can be added or removed at runtime without worries. The system automatically redistributes the data across new servers and manages the back ups transparently.

» An example

The following is a basic code snippet showing how a distributed Map works with Hazelcast:

```
Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance();

Map<String, Integer> scoresMap =
instance.getMap("ScoresByName");
scoresMap.put("Kermit", 1000);
scoresMap.put("Gonzo", 950);
```

Everything should look pretty familiar, Hazelcast provides us with data structures that implement familiar Java interfaces such as *Map, ConcurrentMap, List, Queue etc…* That all developers are familiar and comfortable with.

Behind the scenes the data is published in the grid and backups are managed transparently.

Another node in the cluster could obtain a reference to the same Map and read the data as expected:

```
Config cfg = new Config();
HazelcastInstance instance = Hazelcast.newHazelcastInstance();

Map<String, Integer> scoresMap =
instance.getMap("ScoresByName");
int kermitScore = scoresMap.get("Kermit");
int gonzoScore = scoresMap.get("Gonzo");
```

Additionally most of the distributed data structures in Hazelcast allow many advanced features such as:

- Entry listeners: provide events when an entry in the structure is added/removed/changed
- Synchronous / Asynchronous operations
- Locking
- Persistent storage
- Configurable backups and cacheing
- … much more

Case study: highly available Tris game

It is time to put some of these concepts to work and show how SmartFoxServer 2X and Hazelcast can be integrated in a multi-server environment.

In our use case we will take our simple Tris game, which is provided as an example for multiple platforms, and demonstrate how we could turn it into an highly available, multi server application which could scale in the hundreds of thousands and even millions of CCU.

Keep in mind that the Tris game could be easily swapped with any other turn based game while still using the same concepts that we're going to illustrate.

The main objectives of our game architecture are:

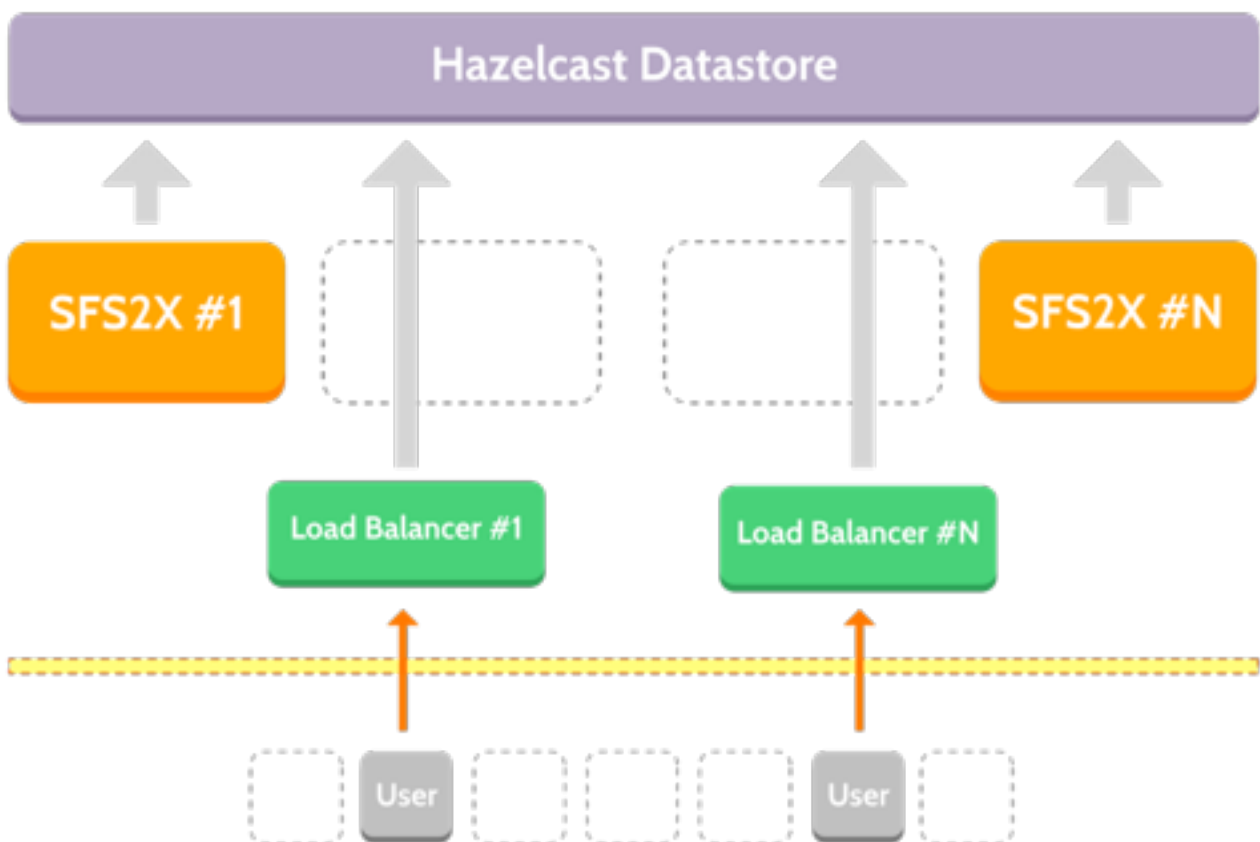
- Scaling the game horizontally allowing the system to grow by adding more SFS2X instances, on demand.
- Ensuring that games are fault tolerant and that game state is not lost if one server dies.
- Allowing players to restart a match that had previously crashed or was interrupted.
- Managing persistence.

» Scalability and load balancing

We want to build a system that enables any number of users to start a 2 players game of Tris. To achieve this goal we need clients to be able to connect to the main website and from there to be taken to the game transparently.

Behind the scenes we will use a Load Balancing Algorithm (LBA) that takes care of directing the player to one of the active game servers. We also need to keep in mind that SmartFoxServer, just like any game server, uses persistent TCP connections, so when a server is selected, the client will stick with it for the whole game session.

The load balancing will be handled by dedicated servers, at least two of them to avoid the dreaded single point of failure, using standard HTTP requests and talking to the Hazelcast data store.



There are several ways in which we can implement this. To keep things simple, we will start by employing a basic “round robin” LBA where a user is assigned to the next available node until we finish the list of nodes and restart from the top.

Before we proceed with the implementation we would also like to mention an alternative approach to load balancing that takes in consideration a “load state value” for each node.

This is an outline of the main flow:

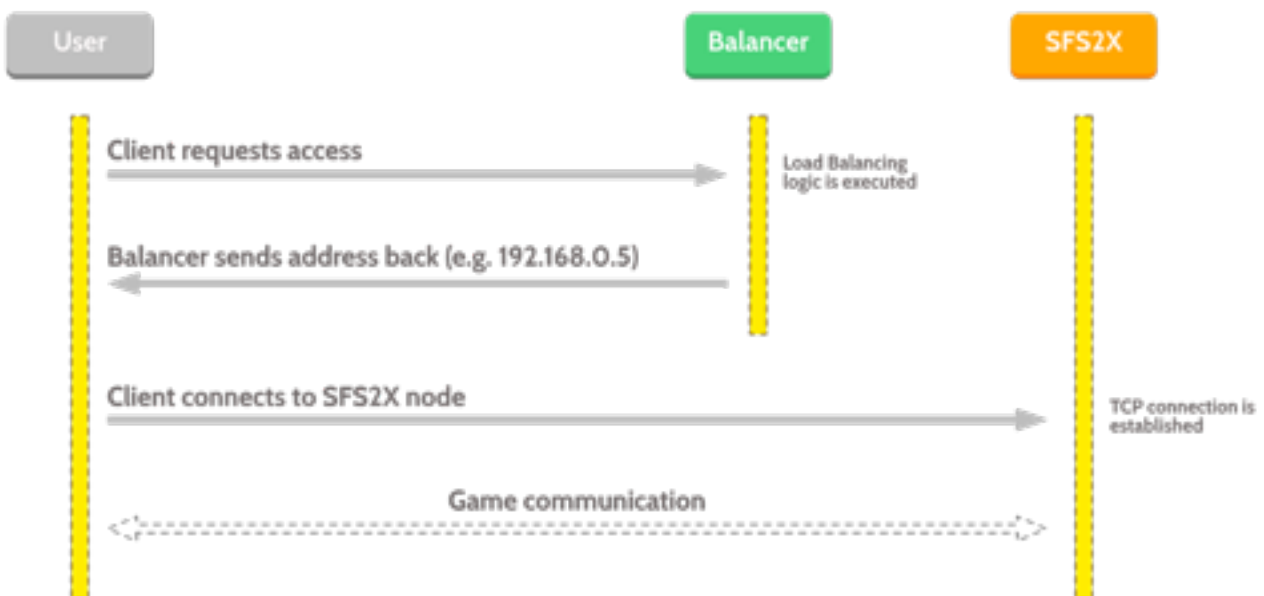
- Each SFS2X node updates its load state in the Datastore every few seconds. Let's say every 10-30 seconds.
- The client asks the load balancer which node it must connect to.
- The load balancer runs the load balancing algorithm (reading the current load state from the data store) and replies with the address of an available SFS2X node.

The load state could be defined by many different parameters or any combinations of them, such as:

- Number of concurrent users (CCU)
- Number of active Rooms / Games
- CPU Load
- RAM Load
- Number of SFS2X nodes
- etc…

This approach should prove quite flexible as it can be customized to work in many different case scenarios and allows for further modifications or improvements even after the system is deployed.

Back to our round robin, this is what the initial load balancing exchange looks like:



Basically we are going to precede the connection to the SFS2X node with an HTTP call to one of the load balancers in order to obtain the address of the target game server.

To maintain fault tolerance we will also provide the client with at least two possible load balancer urls so that if one them fails to respond we can at least try a second one. This could be done via an xml/json configuration file that is retrieved remotely by the client from the game's website, with the advantage of being able to control future updates of the file.

The implementation of the LBA is relatively simple. This is how each SmartFoxServer instance can join the Hazelcast cluster:

```
HazelcastInstance instance = Hazelcast.newHazelcastInstance();

Member mbr = instance.getCluster().getLocalMember();
mbr.setStringAttribute("domain", "alpha.tris.game.com");
```

The first line creates the main Hazelcast instance, joining the cluster. The following line takes a reference to the local member, that is the object that represents the local node in the grid.

Here we can attach custom attributes that will be accessible also from other nodes. This is useful to store global properties that can be retrieved from other nodes such as the load balancer instances.

We're attaching a “domain” attribute to each SmartFoxServer node so that the LBA can later select one node and send the client the url for the connection.

Finally this block of code could be added to our Extension's **init()** method so that the attribute is ready when the server starts up.

The LBA nodes will join the cluster as clients and interact with all the members and data structures shared in the distributed grid:

```
private int rrLastIndex = 0;

HazelcastClient client = HazelcastClient.newHazelcastClient();

public synchronized Member getNextNode()
{
   Set<Member> memSet = client.getCluster().getMembers();
   Member[] members = memSet.toArray(new
Member[memSet.size()]);

   if (rrLastIndex >= members.size())
        rrLastIndex = 0;

   return members[rrLastIndex++];
}
```

Similar to the previous example we create an instance of the Hazelcast object, only this time we use the **HazelcastClient** class. The difference between a standard instance and a client instance is that the latter is a more light-weight process that can still access any of the data in the grid but doesn't participate in the partitioning of the data.

For our "round robin" style LBA we need an index that keeps track of the last node selected and proceeds incrementing by one at every request.

The **getNextNode()** method obtains the list of members, which is a Set, and converts it to an array so that we can access it by index. Then we make sure that the current index isn't overflowing and we return the Member instance, while incrementing the round robin index.

Finally we could then read the "domain" attribute (which we are setting at the server start up) and send it back to the client as part of the LBA HTTP response.

NOTE: the implementation of the HTTP-based Load Balancer is not restricted to a Java-only solution. The Hazelcast Client is available also for languages such as .Net C#, C++, Scala, Python and Node.js, allowing to integrate clients written with these platforms with the main Hazelcast grid.

For more details about the supported platforms we recommend checking the Hazelcast download page.

» Making games fault tolerant

Now that we have laid down the foundation of our architecture and defined a mechanism for load balancing, we can move towards the core of our implementation: making the games resilient to failures.

What we mean by this is that even in the pretty bad event that an entire game server becomes unresponsive (or crashes) we'll be able to restore such game at a later time from any other SFS2X instance in the system.

Our goal is to be able to handle from a single user crash (e.g. the client application dies) to a massive server crash, where one game server instance fails and all its clients loose connection.

To accomplish this we need to store a copy of the game state in the Hazelcast grid, while also maintaining a local copy in our Extension code, since local data access is orders of magnitude faster than a remote access.

> **NOTE:** To put things in perspective accessing local RAM typically is done in the order of nanoseconds while accessing data on a remote machine, even in a very fast network, takes at best 100s of microseconds or, more likely, milliseconds.
>
> The difference in access times between local and remote can be anywhere between **5 to 7 orders of magnitude** (i.e. 100K or 10M times in local RAM as opposed to remote server). (source)

Essentially what we're doing is avoiding to hit a potentially slow database on every move (of every running game) and instead leveraging the performance of in-memory data storage, via a distributed grid such as Hazelcast.

Once a game finishes we no longer need to keep it in the distributed memory: this data is only useful if a game gets interrupted, so when the match is complete we can delete the relevant data and make room for new games.

Modeling the game data

Working with Hazelcast is pretty straightforward. We can publish any of our objects in a distributed collection provided that the object is serializable.

In the classic Tris example we used three main classes:

- TrisGameBoard (class)
- Tile (enum)
- GameState (enum)

where the TrisGameBoard class models the current game state.

```
public final class TrisGameBoard implements Serializable
{
		private static final int BOARD_SIZE = 4;
		private final Tile[][] board;
		private int winner = 0;

		…
		…
		// getters / setters
```

```
public enum Tile implements Serializable
{
		EMPTY(0),
		GREEN(1),
		RED(2);

		…
		…
}
```

```
public enum GameState implements Serializable
{
		RUNNING,
		END_WITH_WINNER,
		END_WITH_TIE;
}
```

We have now modified the previous classes by simply adding an “*implements Serializable*” which allows Hazelcast to use the Java standard serialization to port the objects to any of the nodes in the grid.

Instead of using **TrisGameBoard** as our top class defining a game state we will create a new object for this purpose that contains a bit more information about the game.

```
public class TrisGame implements Serializable
{
        private String gameId;
        private TrisGameBoard game;
        private String player1;
        private String player2;
        private long startTime;
        private boolean active;

        int currPlayerTurn = 0;

        public TrisGame() { }

        public TrisGame(String p1, String p2)
        {
             game = new TrisGameBoard();
             player1 = p1;
             player2 = p2;

             playerTurn = 1;
        }
```

The TrisGame class wraps the board object adding references to each player's names and the the current player turn. This is all the information we need to know to restart an interrupted game. Also we have added a **gameId** property which we will use as unique ID for each distributed game, generated via the Java UUID class.

Let's now see the code we are going to use in our Extension's **init()** method to setup the Hazelcast cluster.

```
    private final String TRIS_MAP_NAME = "TrisMap";
    private final HazelcastInstance instance;
    private Map<String, TrisGame> games;
    private TrisGame localGame;
```

First we define three new fields. The TRIS_MAP_NAME constants will be used to keep a reference to our specific distributed map in the Hazelcast cluster.

Next we proceed starting up Hazelcast:

```
@Override
public void init()
{
        MapConfig mapCfg = new MapConfig();
        mapCfg.setAsyncBackupCount(1);
        mapCfg.setName(TRIS_MAP_NAME);

        Config hazelcastCfg = new Config();
        hazelcastCfg.addMapConfig(mapCfg);

        instance = Hazelcast.newHazelcastInstance();

        games = instance.getMap(TRIS_MAP_NAME);
}
```

The new code added here creates a configuration object for the distributed Map we're going to use and passes it to the main Hazelcast object.

While regular Map implementations in Java don't need any setup, Hazelcast Maps provide many parameters to configure advanced features, such as the number of backups of each Map.

Here we are specifying that we want 1 extra backup for each entry in the cluster. This allows Hazelcast to distribute copies of the Map entries and avoid that one node failure causes the loss of game data.

NOTE: when multiple servers access the same Map, via its name, they will be effectively referencing the same Map instance and thus be able to interact with the same data contained in it.

Also, the Map implementation provided by Hazelcast fully supports concurrency,so it can be safely handled/modified by multiple threads.

Now that we've seen how to modify the game model for serialization and how to setup a distributed Map in our extension we can move to analyzing the logic that manages the game state and how to manage interrupted games.

» Managing the distributed game state

The following steps outline the life cycle of a game of Tris with its distributed state:

1. A game Room is created using the standard API calls in SFS2X (either client or server side)
2. The relative Room Extension is initialized, which in turn references the distributed Map.
3. When the game is ready to start (2 players are joined) a new state object is created (TrisGame) added to the distributed "games" Map.
4. On every move sent by each player the TrisGame instance is updated locally and stored in Hazelcast, by updating the entry in the distributed Map.
5. When the game is finally complete one of the two player is declared winner and the game state can be removed from Hazelcast.

If anything bad happens between points 3 to 5, e.g. a server crash or a player drop, we'll be able to restart the game from the last turn stored in the Hazelcast's distributed Map.

» Handling interrupted games

There are many different aspects that can be taken in consideration when a game is interrupted before its end. A player could have left the game because he was loosing or got bored and he's not interested in retrieving the game from where it was left.

Other times the client application might have frozen, lost connection or crashed forcing the client to leave the game.

It is quite difficult to differentiate these cases so what we can do is maintain these game states for certain amount of time (e.g. a number of days or weeks), allowing the players to return and restart the game, or maybe fill a refund request if the game involves money transactions etc…

For our Tris game we can leave interrupted matches in our datastore for 2 days (48 hours) after which we can remove it.

Players can return to the server and restart the game if they want to finish the interrupted match, knowing that 48 hours later the suspended game will be gone for good.

» Restarting a game

Since every Tris game has a "creator" (he who started the game in the first place) we will allow game owners to start a game that was previous interrupted.

This is pretty simple to implement: after the user has logged in on a server node we can send an Extension call and ask the server a list of games that can be restarted by the user. This in turn involves querying the Hazelcast data store which looks like this:

```
EntryObject eo = new PredicateBuilder().getEntryObject();

Predicate<String, TrisGame> predicate =
eo.isNot("active").and(eo.get("player1").equals(userName));

Set<TrisGame> res = (Set<TrisGame>) games.values(predicate);
```

The predicate object contains the conditions we want to check, i.e. that the game is not active and that the **player1** field matches the user who sent the request.

NOTE: we know player1 is the game's owner because the creator is always joined first in the Room.

This is then applied to our distributed map which returns a Set of matching objects. If the Set is not empty we can go ahead and present a list of games to the user who can then choose to restart one or skip past it.

Once the game is restarted we will need to alert the players that were participating in those games that a previous match is available and that they could join it.

This can be a little tricky because our LBA currently uses a "round-robin" logic and users might end up in different servers, while we would need them connected on the same machine in order to play in the same Room.

NOTE: distributing game Rooms across multiple server is not a good idea, if you're wondering about it. This involves having players scattered all over the place and having to distribute the entire Room and game state across multiple instances. This in turn forces the system to continuously access the network to reach what should be local data, and create lots of performance issues.

In the end it would be too hard to scale and too easy to cause bottlenecks and slow response times for players. Not to speak about the added complexity of programming a game where the game state is distributed on multiple machines.

To solve this issue we will need a new distributed Map that keeps track of the games each player has played. The map will be a Map<String, List<String>>, where the key is the player name (always unique) and the List of string will contain all UUIDs of each started game.

We're using a List<String> to keep track of potentially multiple interrupted games, even if our Tris Game technically allows players to be in one game at a time.

This is how it should work:

- When a user joins a game we'll add the Game UUID to the List of started games
- When a user finishes a game we'll remove the UUID from the same list
- All UUIDs that are left in the list refer to games that were interrupted

The removal of UUIDs from the list could also be triggered by the player manually leaving the game Room or other similar events that are triggered by user itself. To keep things relatively simple here we'll just remove the UUID when a game is complete.

The map can be added and initialized in the Extension's init() method:

```
private final String GAMES_BY_USER = "GamesByUser";
private Map<String, List<String>> gamesByUser;

@Override
public void init()
{
        . . .
        . . .

        MapConfig mapCfg = new MapConfig();
        mapCfg.setAsyncBackupCount(1);
        mapCfg.setName(GAMES_BY_USER);

        Config hazelcastCfg = new Config();
        hazelcastCfg.addMapConfig(mapCfg);

        gamesByUser = instance.getMap(GAMES_BY_USER);
}
```

When we want to add new items we need to check that the list element is initialized:

```
List<String> gameIds = gamesByUser.get("userName");
if (gameIds == null)
   gameIds = new ArrayList<String>();

gameIds.add(targetGame.getGameId());
gamesByUser.put("userName", gameIds);
```

This will get the existing list from the data store, modify it and put it back into Hazelcast.

We will use the same approach to remove one item from the user's game list:

```
List<String> gameIds = gamesByUser.get("userName");
if (gameIds != null)
{
   gameIds.remove(targetGame.getGameId());
   gamesByUser.put("userName", gameIds);
}
```

We don't have to worry about concurrency issues and atomicity here as the Hazelcast map is already and implementation of the ConcurrentMap interface. Additionally each key represents a different user so we don't expect cases in which players contend the same object at the same time.

» Searching for a restarted game

So far we have discussed how a game owner can restart an interrupted game, and we have laid down the basic code to keep track of the each player's suspended games.

Now it's time to put it all together and see how we can notify players (that are not game owners) that a previously interrupted game is available.

Let's consider this scenario:

- User Kermit creates a game and user Gonzo joins, the game is started but Gonzo's computer crashes in the middle of the game.
- An hour later Kermit logs back in the game, and restarts the previous game (we have already showed how to do this).
- Gonzo also logs back in the system but he's connected to a different SmartFox instance.
- The system notifies Gonzo that a previous game is available with user Kermit waiting to play.
- User Gonzo is disconnected from the current instance and connected to the same server where Kermit is.
- The two players are now on the same server, in the same game able to continue from where they left.

We now need to implement the part that notifies the user that a suspended game has been restarted.

To do so we can obtain Gonzo's list of interrupted games and see if they match any of the active games. We can then send the list of matches to the client so that he can choose if and which he wants to join.

This is the pseudo-code we can use to query the data store:

```
List<String> gameIds = gamesByUser.get(userName);

if (gameIds != null)
{
   StringBuilder sb = new StringBuilder();

   for (String gameId : gameIds)
   {
        if (sb.length() > 0)
             sb.append(" OR ");

        sb.append("gameId='").append(gameId).append("'");
   }

   SqlPredicate pred = new SqlPredicate(sb.toString());
   Collection<TrisGame> res = games.values(pred);

   if (res.size() > 0)
   {
        for (TrisGame game : res)
        {
             // ... send game list to user
             if (!game.isActive())
                  continue;
        }
   }
```

After obtaining the list of unfinished games we build a SQL like query by stringing together all game ids with an “OR” operator.

If the resulting list of ids contained this: [“0001”, “0005”, “0007”] the resulting query string would look like this:

gameId=‘0001’ OR gameId=‘0005’ OR gameId=‘00007’

This is a valid query that we can submit to the Hazelcast data store to search for all TrisGame objects with those ids.

If the returned collection is not empty we can cycle through all items and prepare a list of games that the user can join.

Notice how we're skipping the non active games, as those are just waiting to be restarted but they are not currently running.

To make this work correctly we will need that each **TrisGame** instance also contains the IP address of the server which it is running on. This is because the IP is necessary for the client so that it can check if it's already connected in the right server. If not, it will switch connection to the server where the Room is hosted.

This is the final flow of communication at the start up of our Tris client, including what we have described so far:

Persistence

Our final step is to add a layer of physical persistence to the system which will secure all our data in the Hazelcast data store.

In terms of persistence Hazelcast provides a set of calls delegating the actual I/O to the developer, and thus remaining fully agnostic to the modality of storing and retrieving the data.

We could use standard JDBC, the DatabaseManager API in SFS2X, or a more advanced ORM such as JPA which is essentially the official way to do object-relational mapping in Java.

With Hazelcast we can activate the persistence events on a per-data-structure basis, so each distributed Map, List or Queue that we use can employ a different strategy and run with its own persistence logic.

Here's how we can configure our game Map that we have initially defined to support persistence:

```
@Override
public void init()
{
        MapConfig mapCfg = new MapConfig();
        mapCfg.setAsyncBackupCount(1);
        mapCfg.setName(TRIS_MAP_NAME);

        MapStoreConfig storeCfg = new MapStoreConfig();
        storeCfg.setEnabled(true);
        storeCfg.setClassName("my.package.PersistenceClass");

        mapCfg.setMapStoreConfig(storeCfg);

        Config hazelcastCfg = new Config();
        hazelcastCfg.addMapConfig(mapCfg);

        instance = Hazelcast.newHazelcastInstance();

        games = instance.getMap(TRIS_MAP_NAME);
}
```

(The dimly colored code is what we already had earlier, while the emphasized lines are the new ones). We only need to create a **MapStoreConfig** object, enable it and specify the fully qualified name of the class that will receive the persistence events.

This is the body of our custom class:

```
class PersistenceClass implements MapStore<String, TrisGame>
{
   public TrisGame load(String gameId) { return null; }

   public Map<String, TrisGame> loadAll(Collection<String> gameId)
{ return null; }

   public Set<String> loadAllKeys() { return null; }

   public void delete(String gameId) { }

   public void deleteAll(Collection<String> gameId) { }

   public void store(String gameId, TrisGame trisGame) { }

   public void storeAll(Map<String, TrisGame> games) { }
}
```

The MapStore interface supports three operations (load, save and delete) and six relative methods.

Each of these parent methods need to be implemented to properly reflect any changes in the distributed collections to our storage engine.

For the purpose of this tutorial we will not go into the details of hooking up a database and implement MapStore interface. This should anyways pretty straightforward as you can instantiate any database connection directly from the persistence class and fill in the details of each method.

Conclusions and further readings

We have deconstructed the complexity of running a multi-instance SmartFoxServer deployment with a centralized, in-memory data store as a centralized database for coordinating our game.

We have introduced Hazelcast as an open source data grid to integrate distributed and highly available data structures across multiple JVMs with SmartFoxServer. Also, we have discussed the advantages of in-memory storage for faster interaction among servers in the same network, eliminating the slower disk-related I/O waits.

Then we have seen how to implement a generic load balancing system to distribute the load across multiple game servers, and suggested ways to integrate different variables in the balancing logic.

Finally we have dived deep into the management of interrupted games and how to coordinate players so they can either restart or re-enter matches that were previously stopped.

In closing we have touched on how to integrate physical storage into the system to keep backups of the live data store.

» What is next?

We highly recommend to take a look at the official Hazelcast website, maybe download the product and start browsing the documentation.

- Hazelcast website
- Documentation

The integration with SmartFoxServer is very easy as the library comes with a single jar file that can be added to any Extension project and deployed in the SFS2X/extension/__lib__/ folder.

» What are the alternatives?

In closing we'd like to mention other alternatives to Hazelcast, which is not the only Java solution available for this task. If you are interested in evaluating potential alternatives before choosing which library to integrate in your multi-server system, here are several links to similar open source, in-memory key/value stores:

- Apache Ignite
- Infinispan
- Tayz Grid
- Redis

The main reason why we chose Hazelcast is because it is relatively simple to get started and has a balanced amount of features that cover all the needs we have described in this paper and many more. Additionally we have stress tested it and found no particular problems and good overall performance, especially after release 3.5.x (at the moment of writing 3.6.x is available).

The only true limitation with Hazelcast is that the companion Management Tool supports only 2 nodes in the open source edition. This doesn't limit the number of nodes that can be deployed but only the usage of the Management Tool which is limited to 2 nodes. (In other words a commercial license needs to be acquired to use the tool with more than 2 nodes).

» Further discussion

If there's anything else you would like to ask or discuss about this paper you can send us your thoughts and questions via the SmartFoxServer's support forum.