IT 11 025
Examensarbete 30 hp
Maj 2011

Research and implementation of Lobby System in Erlang

Yury Dorofeev
Wilson Tuladhar
Yeli Zhu

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Research and Implementation of Lobby System in Erlang

Yury Dorofeev & Wilson Tuladhar & Yeli Zhu

Nowadays, a number of games which are played online has increased dramatically. According to the statistics, only for the last 3 years giant game corporations and tiny groups of amateurs have produced and emitted 2.1 times more games than all the previous years together. It is a paradoxical situation that such a huge game world lacks the community Lobby systems that can combine games and provide users with convenient opportunities to get access to all of them with only one login.

Handledare: Christian Lönnholm
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 11 025
Tryckt av: Reprocentralen ITC

Contents

List of Figures

Individual contribution

This thesis project required lot of team work such as planning, discussions and implementation. Meanwhile, each of us took part in some individual work activities. There there is a list of them.

Wilson Tuladhar was responsible for the researching of database and Web framework [31]. In design phase he planned the system core and database API. The idea of module structure as well belongs to him. After the design phase was done Wilson developed database API, made the Unity [2] Web client work with the Lobby system and finalize the Unity API, made the system core. The current web design is his merit. The system test such as Eunit [36] and Tsung [39] was done by him.

Yeli Zhu was responsible for researching of database and Web framework together with Wilson. She put much efforts to the Mnesia [26] evaluation. The next step was to evaluate instance messenger Ejabberd [13] and integrate it to our system. Yeli implemented one of the most important system modules Client. The functionality of this module has to be extended in the future.

Yury Dorofeev was responsible for the researching of existing Lobby systems and Unity tool. He designed and implemented the Game Node service, designed database structure and implemented Instance manager service. The test coverage for the Game Node such as Eunit and manual was done by him.

Acknowledgments

First of all, we would like to thank our supervisor, Justin Pearson for his support and suggestions for lending us a hand during the thesis and also during the report writing.

We are also indebt to the guys from Pikkotekk, especially Christian Lönnholm and Björn Dahlman without whom this thesis would not have been possible.

We would also like to thank Mats Lundin and Karolina Malm Holmgren for their help in providing us with the technical equipments and support for our networking problems.

We are grateful to Dick Elfström for providing us a place to stay during our course of thesis work.

1 Introduction

1.1 Project Overview

This thesis is initiated by Pikkotekk [1] with the purpose of implementation of a "Lobby system" for the Unity [2] gaming environment.

Game developers nowadays make games which are played amongst hundreds of players. Some of them may desire to play in large group while others prefer to play in a closed environment with friends or with some other specific people in the game. This has given rise to multiplayer games where players can create their own instances of games, setup their own rules and invite players of their own choosings. Along with this, there is a need for messaging between players and ranking systems. These are some of the defining features for a "Lobby system" within games.

This project was done by a group of 3 students Yeli Zhu, Wilson Tuladhar and Yury Dorofeev. All team members took part in researching, designing and implementation of various parts of the system. Meanwhile, each member was also responsible for some particular part of the system. Pikkotekk played an active role in the system design and development. They gave us useful advice and valuable tips regarding Unity technologies, made the system design revision, provide the group with some tools and did not disturb us from the job.

Currently, there is no central Lobby system for the Unity [2] game environment. Instead, each game uses its own local Lobby System developed by the same team of developers. This fact complicates the process of game development dramatically forcing developers to spend their valuable resources to implement functionality such as user login, user registration, chatting etc. that does not improve their product characteristics. Instead, game developers should put all their strength to increase the game competitive ability and provide final players with amazing online games.

As from the players perspective also, there are lots of disadvantages. They meet the problem of multiple game registration and authentication. The scale of problem rises dramatically and becomes critical for super active game players. Below there is an example which is followed by numbers and a simple calculation.

Task: to register a user

Initial conditions: one player A going to be registered to 10 games

Minimum number of user operations for one registration: 8 (type username, password, real name, real surname, e-mail, address, telephone, personnummer)

In order to be register to all these games user A has to make 10x8 operations + two operations for log-in procedure (type username and password) x 10 = 100 operations.

Totally: 100 operations.

The result is significant. It is obvious that nobody wants to spend so much time to be registered only. To play multiple games, users need to be registered only once. Besides providing simplicity for the users/players, the Lobby system helps game developers to promote their product as players will be able to view and play the games as soon as they are registered to the system. The Lobby system should provide reliable, configurable and simple mechanism for game developers to plug their games in into the system.

The lobby system was designed and implemented in such a way that it could accommodate any type of games with minimal configuration. Since the Lobby system is made general for multiple games, the load on the system will be high as the number of games, game instances and players increase. It means that the system needs to be scalable and highly robust. We have used Erlang [3] as the programming language due to the fact that Erlang applications are highly known for its distributability, robustness and fault tolerance.

Current project includes different activities

- Requirements elicitation
- Study and analyzing the current existing Lobby systems
- System design
- System implementation
- Testing and bug-fixing
- Releasing the System

The first section includes fundamental and deep analysis of both existing Lobby systems and principles of how online games work. None of the project team members were highly experienced game developers which made some impediments at the beginning. After evaluating several Lobby systems we found that their design does not fit our system requirements and made the process of running new games quite complicated.

Design phase took 3 weeks. The main problem there was how to make the system general for any Unity games. An appropriate conception was found. One of the advantages of the system is that it does not care about any particular game and the internal process inside. The lobby server does not even need to care where games are hosted i.e it does not need to be hosted where the Lobby system resides. All the simple Lobby-Game communications are established through the API which were developed by the Lobby team and which could even be easily extended .

After the system design was completed and approved by our supervisor and the reviewer, we moved on to the implementation phase. That part required strong Erlang/OTP experience and knowledge of the Unix OS[4]. Testing was done concurrently during the implementation by using special tools and also manually.

1.2 Thesis Specification

Since the focus of the thesis was to develop the system which can serve multiple number of different games without any dependencies with any games whatsoever, so from the game developer's point of view, the Lobby system is a black box, to which they plug their game server and client to and the system will manage the traffic between players and games. What happens inside the Lobby system? Generally speaking, the Lobby system consists of the following features:

- Adding the developed games into the system without having to do code-level changes
- Game list: A list of existing games
- Game instance list: A list of existing game instances of a specific game
- Creating game instance: Create game instance button that allows players to create a new instance of an existing game.
- Joining existing game instance: A join game instance button allows players to join an existing game instance of a specific game
- User list:
  - A list of registered users in one game instance
  - A list of registered users in all game instances
- Chat: players are able to chat in game play
- Single Sign-On Feature

We believed that it is going to be a fascinating project to try and explore more features in Erlang.

1.3 Outline of Thesis

This section outlines the structure of the thesis.

Section 1 give the introduction about the general description of what the system is, what is the motivation for doing it and the amount of work needed to be done to complete the thesis.

Section 2 describes about the gaming concepts, where and how they are built and the conventional implementation of the lobby system in the game itself and also some reasons in what perspective our system is suitable over them.

Section 3 describes the working environment for the successful implementation of the project. It includes the description of the tools we have used for implementation and some comparison on why we chose the tools that we did.

Section 4 describes the overall and detailed design of the system.

Section 5 describes the actual implementation of the system. Here, you will find the detailed description of the system and reasons behind implementing in such a way.

Section 6 describes the various tests that we conducted to check the stability and robustness of our system.

Section 7 describes different types of problems and issues that we faced during the research, design and implementation phase and how we overcame those issues and problems.

Section 8 describes some of the features that could be useful in the system but have not yet been implemented yet in the system.

Section 9 gives the conclusion for the thesis

2 Related Work

One of the most famous existing Lobby system that we found was "SpringLobby" [7]. This is an open source cross-platform framework developed for supporting RTS Engine-based [8] games. The system consists of two parts Server and Client. The client is represented by number of stand alone applications written in different languages. Different clients provide different functionalities and features. They allow users to play games online or off-line as a single player. Currently there are three working open source clients available of which two of them are still in development phase. The list of Lobby clients:

- SpringLobby is cross-platform client written in C++ [19]
- Zero-K is Windows-based client written in C# [20]
- Alphalobby is Windows-based client written in C [49]
- TASClient is Windows-based client written in Delphi [50]
- QtLobby is cross-platform client written in C++, unmaintained

In the online game world there are two types of Lobby system:

- thick-client and thin-server
- thin-client and thick-server

The server is centralized in both cases. According to the statistics provided by International Computer Games Association (ICGA) [12], the number of Lobby systems of the first type in 2010 was 80% against 20% of the system of the second type respectively.

Thick-client Lobby domination might be explained by several factors. First one is the server simplicity. Here, the Lobby server provides the routing functionality only. It means that the server does not contain any logic and plays the role of a connector center for all the clients. The second factor is ability to host games. In order to start a new game (or battle), a player has to download and install this game on his/her machine. Then the actual player can either become a game host and choose to allow other players to join the game. Meanwhile, this type of system suffer for the quick game joining. It means that if a player wants to join en existing game (battle) it is inevitable that he/she has to download the game client as well.

Lobby client supports two regimes: online and offline respectively. In online regime, client permanently connects to the Lobby Server. Then the player is able to see other players and join their battles and play as a team or as opponents. Depending on the design and implementation, Lobby client can provide voice chat, video chat, conferencing, etc. In offline regime, a player can play only games which are installed on his/her machine. However, regardless of the regimes, the game client has to be installed locally.

The central Lobby server is responsible for providing player-player, player-game communication. It also might provide some functionality for keeping track of users joining and quiting games and available battles.

"SpringLobby" system allows a player to host its own game or a battle. In that case, Lobby server might be initial game provider or the game host himself. It means that in the first case all the available games are registered and placed on the Lobby server themselves. When players decides to start their own game, they downloads it from the Lobby server directly. The second alternative means that the Lobby server does not store any games at all. Instead, it redirect players to the responsible game source server and a player get the game from there.

After the analysis part of our project was finished, we came up with a decision to not follow the module/system described above. Instead, we developed our own unique solution which gives Lobby, game developers and players simple and rapid mechanism of management, hosting and playing computer online games. More information about system design and architecture specifics can be found in chapter Design 4. And for more details regarding the implementation, see chapter Implementation 5.

3 Working Environment

This section first gives the brief overview of the company we are developing this product for and then later the tools that we have used for building the system and why chose those tools that we did with some nice comparison with its nearest competitors.

3.1 Erlang

Since we were building a web-based Lobby system, it could have been developed in various of web languages such as PHP [51], Java Server Pages (JSP) [52], ASP.NET [53] but we chose to do in it Erlang/OTP [3]. So why did we choose the Erlang to build our system over the rest?

Erlang is a programming language developed by Ericsson [54] during 1980s with the main focus on distributability, high availability and fault tolerance for the telecommunication products. Erlang was built for the concurrent network environment which could handle huge number of transaction requests without crashing and even in an event of crash the ability to restart itself and function as if nothing ever happened. The Erlang nodes could still function even if any other node it is supposed to send the request is down or updating, waiting and collecting the request for the other node to come up and then function again.

The first version of Erlang was published in 1986 by Joe Armstrong and his team from the Ericsson Computer Labs and since then a lot of noticeable features have been added to it with Open Telecoms Platform (OTP) being released on 1996. OTP contained a huge set of libraries to solve telecommunication as well as network related problems, many error handling mechanisms in building a system which not only provided robust system but also organized the way the code was to be written in Erlang. These features attracted a lots of developers to use Erlang, so it was made publicly available as "Open Source" in 1998. Since then a lots of companies have used Erlang for various applications, some of them being the world renowned companies like Amazon [55], Facebook [56] and Yahoo [57].

Due to the distinctive features such as concurrency, light-weight processes, high distributability and robustness nature of Erlang, its use was not confined to telecom applications only. Web-servers such as Yaws [6], Inets [58], Nginx [59], Mochiweb [60] and web-frameworks such as Erlang Web [9], Nitrogen [5], Zotonic [61] which were also built in Erlang which made it possible to develop web-applications written entirely in Erlang with support of HTML [62] templates, CSS [63] and Java scripts [64]. This made perfect sense as the world of web also require high concurrency, throughput and availability.

Erlang is fast because it does not rely on the OS (operation system) threads i.e each process in Erlang is not an OS thread as in other languages but rather they are created and managed by the Erlang VM regardless of the OS thread.

Below we can see some distinctive comparisons in the process creation and message passing between Erlang, Java [68] and C# [20] and also a comparison of the

throughput between the Erlang built Yaws server and Apache server [10].

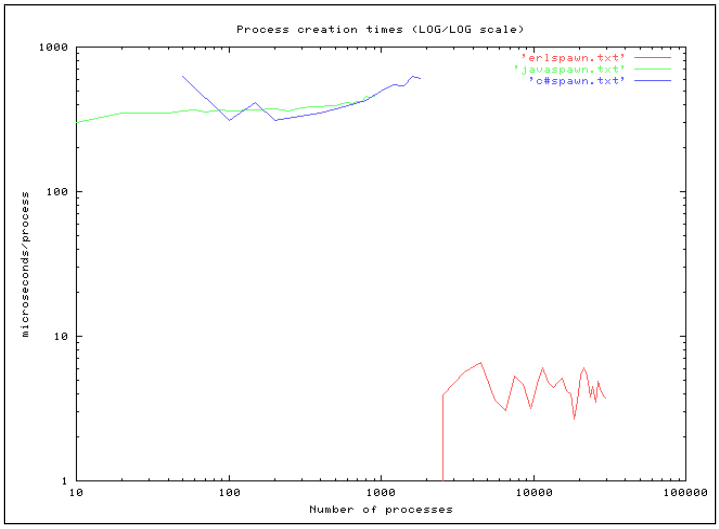

Figure 1: Process Creation

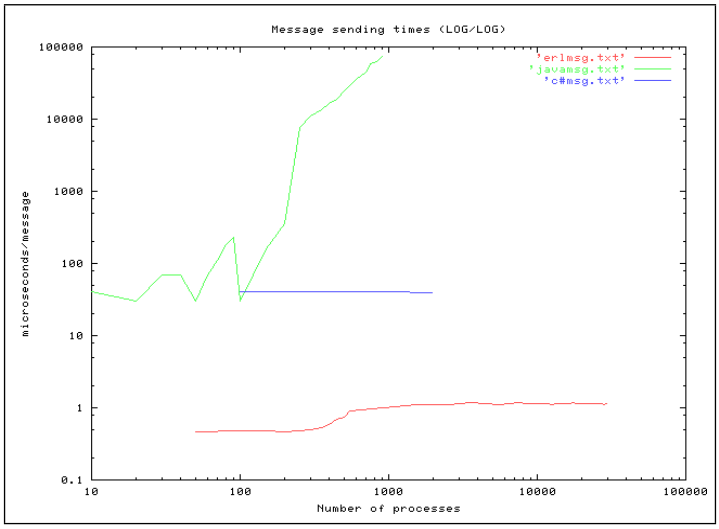

Figure 2: Message Passing between Processes

As seen from the figures Figure 1 and Figure 2, the time taken for creating a process in Erlang is way less than in Java and C# and also the time taken to send a message from one process to the other in Erlang is more or less constant as there is rise in the number of process. As C# also has a constant time for message passing but it takes longer time than in Erlang and in Java even more with increase in time as the processes keep on increasing.

Also as seen from the Figure 3, the capability to handle the request of an Erlang built web server, Yaws is much greater than the Apache server. The Apache server could only handle around 4000 requests while Yaws server could handle around 80000 requests, an increase of 200% more than that of the Apache web server.

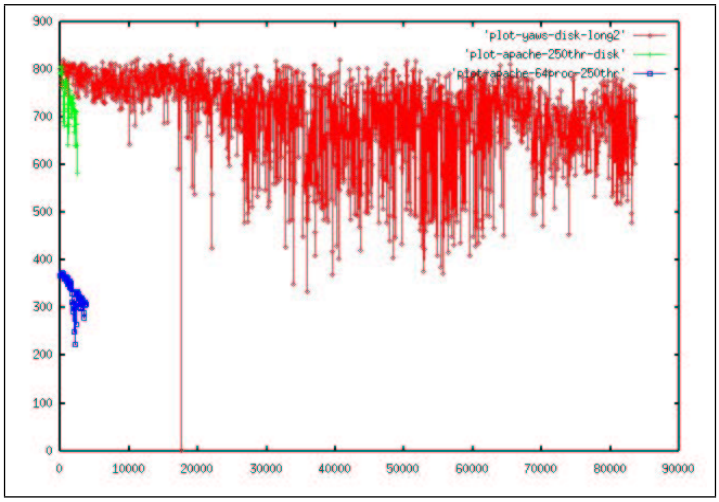

Figure 3: Comparison of Apache and Yaws Web server throughput

3.2 Database

A database [21] is one of the major components for any application development. It is one of the component where all the information concerned with the system are persistently stored and which gets queried the most either for storing a new data or updating the existing data or retrieving the stored data or may be deleting them completely.

During the initial research phase of the thesis work, we had to find a database which would fit with our system requirements. Since we were building a system in Erlang [3] which is in itself distributable in nature, so our main focus was on using database which was also distributable in nature.

Many databases nowadays claim to be distributable but in real situation they are replicated databases. In order to understand the difference between them, we will give a brief explanation about what distributable and replicated databases are. A database is distributable when it assign different part of its data to different database nodes. It means that node A does not contain the data which belongs to node B and vice versa. A database is replicated when all its nodes contain similar data. It means that node A contains all the database data as well as node B contains the same data and hence the data replica.

The second important factor for us was the language in which database is written on. It was critical for us because one of the specific features of Erlang based applications was to be able to generate and handle lots of concurrent requests. Thus, the chosen database should be able to handle high load and still be fault tolerant.

Following these two requirements, the conventional relational databases like SQL Server [22], MySQL [23], Oracle [24] and PostgresQL [25] were omitted from the list of contenders as we were looking for non-SQL DB. After bit of search, we found three possible databases to choose from, which were CouchDB [11], Riak [15] and Mnesia [26]. The great thing about all these databases were they are all built on Erlang. So in this section we have described each of them briefly, compared one to another and

at last made our decision of which database is fit for us to use.

3.2.1 CouchDB

CouchDB [11] is an open-souce, document-store database written in Erlang [3], which was built by the Software Foundation Apache [27]. The obvious question about CouchDB is what is document store and how is it different from the SQL [28] databases. Document store entails that all the data are stored in a field-value fashion inside a document in a database. A document can be regarded as a single row of entry in a normal SQL database table. The database in CouchDB corresponds to a table in the SQL terms, not the actual database.

CouchDB is a schema-less database which means the users do not have to define any specific schema prior to inserting the data. The developer only has to create a database and keep inserting any field-value pairs. By doing so, the users can now search the required documents through the unique document id. But if one wishes to search the documents through the related fields in the document, some special mechanism required. In CouchDB this mechanism is implemented as view. A view is in some senses similar to the indexing but here the entire document is stored again with the view_id (the search field). A view is useful when one needs to get a number of documents from one database with only one query. The shortcoming of view mechanism is that it is not possible to link two or more databases in one view to get the combined result.

3.2.2 Mnesia

Mnesia [26] is an open-source, record based, and distributed DBMS written in Erlang [3]. It is an Erlang application with the combination of ETS (Erlang Term Storage [33]) and Dets (Disk Erlang Term Storage [34]) tables. Mnesia is designed for Erlang based telecommunication applications. It is intended to satisfy the following requirements (from [35]):

1. Fast real-time key/value lookup.
2. Complicated non real-time queries, mainly for operation and maintenance.
3. Distributed data due to distributed applications.
4. High fault tolerance.
5. Dynamic reconfiguration.
6. Complex objects.

Before creating tables in Mnesia, a schema needs to be created that contains the definitions of all tables. A Mnesia schema can be created over distributed Erlang nodes, each of which will contains a directory. Tables (or data) are replicated on all the nodes on which they are created.

In Mnesia, the data is stored as tables of Erlang records. For example, a PERSON record is defined as follows:

```
-record(person, {name,
                 address,
                 phone}).
```

The record name matches the table name. A table is a collection of records of the same type (e.g., a PERSON table contains PERSON records). Each row in the table contains one record of that type (e.g., one PERSON). The record entries (from previous example: name, address, phone) act as the rows in a SQL database.

When inserting data, many concurrent processes may access and manipulate the same objects simultaneously, which in turn causes the database to be inconsistent. To avoid such situation, Mnesia transactions are introduced to guarantee that the distributed Mnesia database remains consistent. Besides transactions, Mnesia provides another operations known as dirty operations for reading, writing and deleting data. With dirty operations, there are no locks required and therefore dirty operations are much faster than transactions.

Mnesia uses ETS and Dets tables for storing memory- and disk-data. The following table shows the upper limit of a Mnesia table for storing different storage types.

Table 1: The Upper Limit of Mnesia for Storing Different Storage Types

Storage Types	Upper Limit
disc-only	2 GB
32-bit systems	4 GB
64-bit systems	16 exabytes

A Mnesia table can store data on disk up to 2 GB due to the storage limitations of Dets tables. What we considered about Mnesia is that it is not scalable enough for the system requirements. It is not the right choice for increasing amounts of data that could be generated during game play.

3.2.3 Riak

Riak [15] is a highly distributable database which was influenced by Amazon's Dynamo Paper [16] and Dr. Eric Brewer's CAP Theorem [17]. It is based on the results of the Sales Force Automation business [29] by Basho Technologies [30] which later focused on data store technology instead. Riak is written mainly in Erlang [3] and C [49] with some part of the code also written Java script [64]. Riak is also an open-source database developed by Rusty Klophaus. However, there is also an Enterprise version of Riak which includes extra features such as SNMP [69] suport, inter-data center replication, web-based administration interface and a top-tier support.

It is a key-value store database which means the data are stored such that a unique key identifies a list of values. Riak is built in such a way that it can be scaled to any point easily and efficient without any single point of failure. In Riak, we have "Buckets" where all the data are stored. So, while comparing Riak in terms of an SQL

database, one could say that the "Bucket" is a table, "Key" serves as a primary key, which is mandatory in Riak and "Values" are again a key-value pair comprising the rest of the column-value pair for that particular row of the table in the SQL database. Riak may be written in Erlang and C but it support a wide range of programming languages like Java [68], PHP [51], Python [70], Ruby [71] and also Java script. There are client-libraries that have been built to support these libraries, so the developers can choose any one platform and use Riak database according to ones needs. Theses libraries could be used to store, retrieve and update the data in the Riak DB.

After storing the data, if the user knows the key of the stored data, then retrieving is a straightforward process. However, if one wishes to retrieve the data by searching and matching through the values of the bucket then in such case Riak provides the possibility to index the bucket. However, in such circumstances, Riak is no longer a schema less database as one has to define a schema of what needs to be indexed prior to inserting the data. There is a certain format for creating where one specifies what keys from the values section needs to be indexed and how they should be indexed. Some of the available options are Whitespace Analyzer Factory, which tokenizes the field by splitting the text according to whitespace (spaces, tabs, newlines, carriage returns, etc.). Standard Analyzer Factory, is an useful for full-text searches across documents is written in English, Integer Analyzer Factory, which tokenizes a field by finding any integers within the filed and so on. After we define the schema in Riak, now any operations such as insert or update on that Bucket will result in the values also being indexed. There is still one defect left in the Riak index which is if due to some reasons one needs to expand the index field, then all the data that have been stored previously need to be re-indexed again. Now, after the data have been successfully inserted and index, we can use the Lucene syntax [72] for searching the data.

Redis

Redis[40] is also an open source key-value store which is often is used with Riak and other databases for various operations. Since the keys in Redis can contain any data structure such as strings, lists, sets, hashes and sorted, it is also referred to as a data structure structure. Redis server is one of the node and it has support for different clients which includes Action Script [73], C [49], C# [20], C++ [19], Erlang [3], Go [74], Haskell [75], Java [68], Node.js [76], Perl [77], PHP [51], Python [70] and many more. It is a replication based key-value store and is written in C.

3.2.4 Comparison between CouchDB, Mnesia and Riak

	CouchDB	Mnesia	Riak
Storage Type	document-store	record-based storage	key-value store
Pre-defined Schema	no	yes	no and yes(in case of Riak-search)
Distributability	yes(replication based)	yes(replication based)	yes
Memory Copies	no	yes	no
Storage Limit	no	yes (Table 1)	no
Support for views	yes	no	no
Indexing values	no	no	yes
Fault Tolerant	yes	yes	yes
RESTful	yes	no	yes
Client libraries	Erlang, Java, Python, PHP	Erlang Only	Erlang, Java, Java script, PHP, Python, Ruby

3.2.5 Our Choice

After looking into all three databases, we chose to use Riak[15] as our database. We chose Riak not because the other two databases are not good but because of the fact that Riak's features seemed to fit well to our requirements. Since the games are added by the users dynamically and the information to store varies between one another and we need to create the storage place on the fly, it seemed quite easy in Riak. To be able to create a schema in Riak we just take a predefined template and fill it with the bucket's fields we want to be indexed. CouchDB [11] for instance does not support this mechanism but uses views instead. Moreover, views mechanism has many restrictions.

One other advantage of Riak over CouchDB and Mnesia [26] is it is truly distributable with no replicated data over the nodes. We could have chosen Mnesia if it did not have the storage size limit. And the final reason was also that Pikkotekk had encouraged us from the beginning to use Riak over the rest.

3.3 Web Framework

Since we were building a web application with all the components built in Erlang [3], so we needed a web server and web framework to host and build our site that used Erlang as well. After bit of searching we found three frameworks that were suitable for our project namely Erlang Web [9], Nitrogen [5] and Zotonic [61]. So in this section we have described each of them briefly, compared one to another and at last made our decision of which framework to use.

3.3.1 Erlang Web

Erlang Web [9], is an open source web framework developed by Francesco Cesarini and is being maintained by Erlang Solutions [78]. Erlang Web [9], at first glance, seemed pretty nice with it supporting the very popular MVC (Model, View, Controller) [42] model of web applications. The HTML [62] templates are kept separately from the rest of the system and there is a controller which is responsible for receiving the requests and after the request has been processed, the template is loaded into the controller and appropriate data is fitted into the template and then served for display. Considering the fact that it is open-source, the documentation was quite good with a full detailed Wiki pages of almost all the important modules in it. The support for various web servers like Inets [58] and Yaws [6] also make Erlang Web a framework to look at and further more it has and in-built authentication module with support for Erlang's Mnesia [26] database as well as CouchDB [11], a document-oriented database. Erlang Web also supports different HTML template rendering engines with default being its own Wparts [79] engine and possible supports for ErlyDTL [80] and DJango [1] templatting language. It also provides session handling mechanism in nice and easy fashion.

3.3.2 Nitrogen

Nitrogen [5], an another open source web framework, is also built in Erlang [3] and is developed and maintained by Rusty Klophaus. Unlike Erlang Web [9], Nitrogen uses an event-driven architecture, with a lot more support for Java script [64], Ajax [82], and Templatting, which make it more easier and faster for developers to build a fancy web pages. Nitrogen works with three best known Erlang Web servers: Inets [58], Yaws [6], and Mochiweb [60]. Nitrogen treats all the web pages as Erlang modules which are later rendered to HTML [62] by the templating engine. All the HTML components are treated as Erlang records with fairly similar names. Nitrogen also provides mechanisms for handling the user session and keeping track of user's roles, if specified. It has no inbuilt support of the databases but its very simple for a developer to make his own database API and use with Nitrogen.

3.3.3 Zotonic

Zotonic [61] is not only an open source web framework but also it regards itself to be a CMS (Content Management System) [83] developed by bunch of developers namely Marc Worrel (Lead architect of Zotonic), Arjan Scherpenisse, Maas-Maarten Zeeman and Atilla Erdodi (Core developers), Tim Benniks and Peet Sneekes. Zotonic is also written in Erlang [3] with the support to running only with PostgreSQL database [25].

It also has a lot of features regarding making dynamic web pages, session handling and templating engines.

3.3.4 Comparison between Erlang Web, Nitrogen and Zotonic

	Erlang-web	Nitrogen	Zotonic
Architecture	MVC based	Event Driven	Event Driven
Servers	Inets, Yaws	Inets, Yaws, Mochiweb, Misultin	Mochiweb
Container	EWGI	SimpleBridge	Webmachine
Templating Language	wparts, ErlyDTL	Record Based	ErlyDTL
Databases	Mnesia, CouchDB	N/A	PostgreSQL
Inbuilt Authentication	yes (e_auth)	no	yes(Oauth)
Java script Form Validation	no	yes	yes
Built-in Comet	no	yes	yes
Error logging	yes	yes	yes

3.3.5 Our Choice

After looking into Zotonic [61] and realizing that it does not use and Erlang based non-SQL database and that we cannot remove the database PostgreSQL [25] from it, so we decided to drop it. So we had to choose between Erlang Web [9] and Nitrogen [5]. The main advantage of Erlang Web over Nitrogen was its use of MVC model [42] in default, but it was not luring enough for us to use it as we could have implemented that structure in Nitrogen quite easily as well and we did as well. But the Ajax [82] features, Java script [64] validations and simplicity to build the pages gave Nitrogen a slight edge over the Erlang Web and hence we decided to use Nitrogen as the web framework to render the web pages.

3.4 Instant Messaging System

3.4.1 Ejabberd

Ejabberd [13] is an open source, distributed, and fault-tolerant instant messaging server written in Erlang [3]. It implements the eXtensible Messaging and Presence Protocol (XMPP) [43], which is an open-standard and XML-based [84] communication protocol formalized by the Internet Engineering Task Force [44].

In Ejabberd, internal modules are Erlang modules beginning with the word "mod". Each internal module implements the "gen _mod" behaviour and provides the following API [14]:

start(Host, Opts) ->ok
stop(Host) ->ok

Some features that included in Ejabberd are listed here:

Table 2: Ejabberd Supported technologies

Supported technologies	Yes/No
XMPP Protocol	Yes
Erlang/OTP	Yes
Mnesia	Yes
ODBC	Yes
MySQL	Yes
PostgreSQL	Yes
CouchDB	No
Riak	No
LDAPS	Yes
OpenSSL	Yes
HTTP-Bind (BOSH)	Yes
Administrative Console	Yes

3.4.2 iJab

iJab [45] is an open source, Ajax-based [82] instant messaging client development in Java script [64] and HTML [62]. It is based on XMPP/Jabber instant messaging protocol [88] and it works with any dedicated XMPP/Jabber server such as Ejabberd [13] or Openfire [65]. There are many other popular XMPP chat clients, such as Pidgin [54], Gajim [55] and so on. These are desktop console client differ from iJab that is a web console client.

Some of the features that included in iJab are listed here:

- iJabBar.
- Sound Support. It can be easily enabled or disabled.
- Group Chat Support.
- HTTP-Bind Support.
- Localization.

iJab provides a Facebook [56] style chat bar that runs inside a web browser without downloading additional software. It is a cross-browser application that works with different web browsers, such as Firefox, Chrome and so on. iJab is now available in English and Simplified Chinese: "en" for English and "zh" for Simplified Chinese. To set which language you want to use, you can use the following code:

```
<meta name="gwt:property" content="locale=en">
```

iJab is compatible with other chat networks, such as AIM [86], ICQ [87] and so on. It supports many features as listed below:

- Sound Support.
- Group Chat Support (with Multi-User Chat (MUC) protocol [88]).
- Invitation.
- User Search.
- HTTP-Bind Support.
- User Management.
- Roster Management.
- Emotions & Avatars.
- Notification.
- Blacklist Support.
- Chat History Support.
- Transport Registration Support.

3.5 Unity

Unity [2] is a game development environment which allows the developers to focus simply on creating game by hiding the other irrelevant facts from the developers. It is fairly easy to create a game in quite less amount of time using Unity. Unity has all the functionalities required for networking with little learning and even less effort. One can create a single player game to MMO [89] games without much difference. It can be run on web, mobile or in native console terminal.

Some of the features that Unity provides for developing games are:

- Rendering
- Lighting
- Terrains
- Physics
- Audio
- Programming
- Networking

For creating the game client-server module, they provide a network framework called "ulink" which helps in the making communication links between the players with the game servers.

4 Design

The previous section described what tools we used to build our system. Now we will go into more detailed information about our system design and explain how we have used those tools to achieve our goals.

4.1 System overview

The architecture seen in the figure 4 provides the general structure of the system Lobby-Game-Player. This is a classical triangle architecture which does not depend on either the type of the game or the Lobby itself. All the existing Lobby systems obey this architecture. In the next chapters we will give you detailed explanation about the Lobby part of this schema. The current Lobby system is designed to be a stand-alone application from both the game and the client sides. Since we are not concerned with the development of the game, the design part will not give you any information about game architecture and how they are developed. The Lobby system does not put any restriction on the game design and implementation. The only thing it does is it provides some APIs for Game-Lobby communication. These APIs are used to connect from the game client to the game server and also for the games to send data to the Lobby system. The great feature of the system design is that the Lobby will still function correctly without crashing even if those APIs are not used by the game developers. The only problem is that users will be able to see that game in the Lobby's list of available games but will not be able to play it. Thus, the basic principle of components independence is reached. Each component of the system (Game, Lobby, Client) works (does not crash) even if all other components are not there.

As seen from the figure 4, which is the basic skeleton of our entire system, there are three major parts that needed to be implemented. The "Lobby" is the actual Lobby server which is the central body of our system. All the requests are forwarded to it, all the responses are generated from it and data are stored in its database service.

For now, let's assume that it does all these stuffs and we will come into more details of how it is done later. Then, we have the "Game Client" which are the actual end users which generates almost all the requests to the system. Last element is the "Game Server" which is where the actual game is hosted. The "Game Server" also contains various parts but for now it is a black box which is started by the Lobby system, sends requests to the Lobby system and allows the users to play the game.

After the user has been authenticated in the Lobby server, it is no longer required to authenticate when joining the game. When a player has joined the game it is connected directly to the actual game server. However, the connection to the Lobby system is not lost since a player plays game inside the Lobby system.

A game is able to send messages to the Lobby server. There might be many different type of messages predefined in Game-Lobby API but there are numbers of them which are mandatory for being used by game developer. Using these APIs guarantee that the system Lobby-Game-Client will work correctly with stability. See

the list of API's and description in Appendix A.

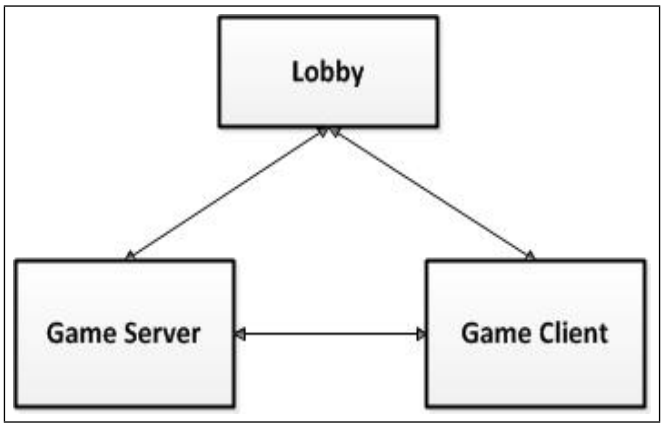

Figure 4: General Lobby-Game-Player structure

4.2 Lobby System

The architecture shown in figure 5 describes the general structure of the Lobby system. Meanwhile, the detailed information about module-module communication and message passing is hidden there.

As seen from the figure, the Lobby system is separated into different modules with each module responsible for certain tasks, thus making it easy to distribute and each module is in itself distributable in nature as well. As we are making the system which is accessible online through the web-browser, so we have tried to follow the MVC model [42] of the software architecture when handling the requests in the webframework. The MVC model divides the processing of the web applications into three components namely Model, View and Controller. Model is the one which works as the backend and manages the data and behaviour of the application domain. It accepts the requests from the view and responds if there have been any change in its state. View is responsible for rendering the model and presenting it in a nice form to the end user. The controller is the central part which connects the model and the view. It receives all the requests from the view and decides where to make calls on the model and then the response from the model is sent to the calling viewport as the end result to be presented to the user.

We have chosen Nitrogen as our webframework which acts as a viewport in the MVC model. The core consists of one controller and few other modules which helps in making the system more abstract and will be described later as we go along and the rest of the system will act as the base model of the MVC model.

Since we wanted our system to be as distributable as possible, we have separated the system into several modules where each module can be run on a separate node (on the same or different machine) independent of other nodes. We have the Authentication module which is responsible for user authentication operations such as adding a new user, deleting the existing user, authenticating the user's credentials for logging

into the system, changing password and updating the logged in user's profile.

Next, we have the Admin module which is responsible for all the administrative operations such as adding the new game into the system, adding numerous instances of the existing games, dynamically configuring the system to be able to handle the game data and index them when the game sends the statistics of some game instances as the players start to play them.

The Client module is responsible for handling all the requests from the users other than the administrative ones. These requests include features like starting a new instance of a game, joining the currently running game, querying about the running games, querying about other players and about their statistics, filtering out the results and so on.

The Database module is the module which directly communicates with the Lobby database. This module provides the layer of abstraction to the system as if one wishes to swap out the actual database then the only change that needs to be done so that the new database will be used instead of the current one is here. This module will do the generic calls to the actual database and returns the result to its caller.

Last but not the least is the Game node, which is a part Lobby system but resides outside the system. The Game node is run on all the machines which hosts the actual game. The Game node is responsible for managing the game instances in the host and the communication between the Lobby server and the games. For more details look to the chapter Design/Game node 4.5 and Implementation/Game node 5.7.

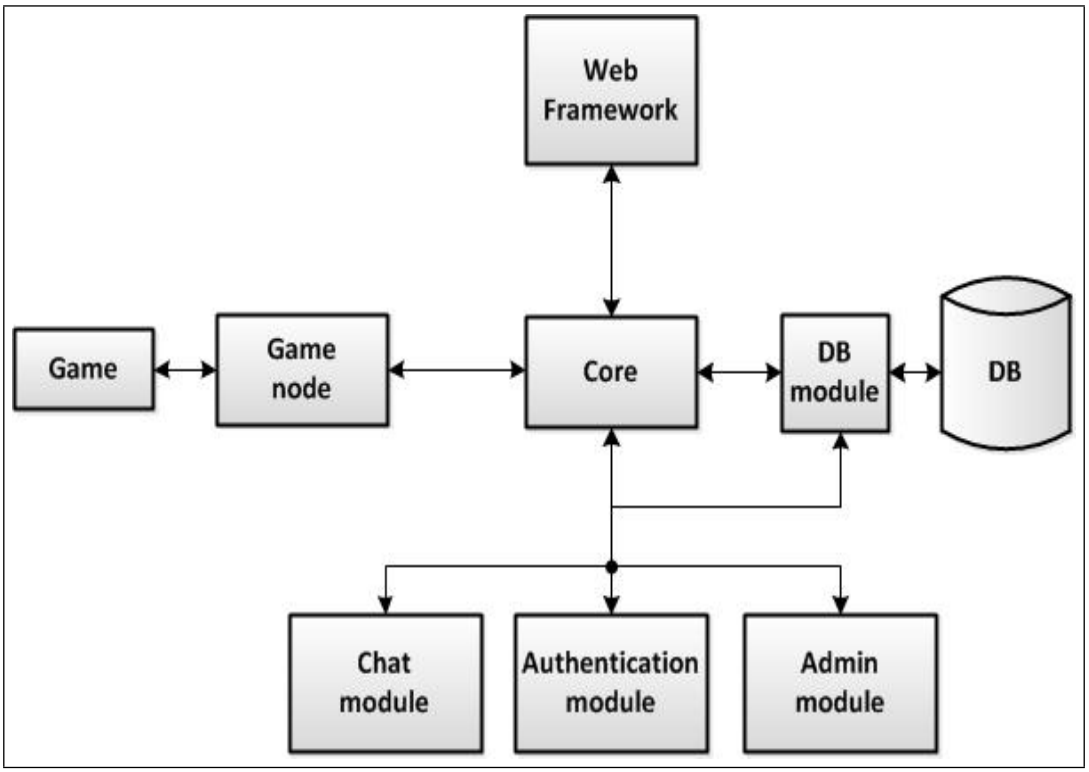

Figure 5: Lobby structure

4.3 Module design

We have tried to make all our modules to have a similar behaviour with the 3-level structure. The general module design is depicted in the figure 6. This has worked out in most of the modules except few ones such as web-framework and game server. Generally speaking, the module consists of an API which is publicly available to all other modules to make calls into that specific module. For instance, for authenticating a user, the Web Framework will call the functions which are publicly available through API of the authentication module and apart from that it does not care what happens inside it but only expects a valid response. The API will then make a function call to its dedicated server (gen_server). Going deeper to the logic we can say that the API does not even know that it makes a call to the gen_server, where the function again does a generic server call(synchronous call). Instead, API calls the exported function of some related module. The call is handled in the server where a process is spawned for each request with the request and the caller ID as the arguments in the callback module for executing the request and then the reply is sent directly to the caller of the API. After the request has been executed and the response has been sent to the caller, the spawned process dies. This structure has helped in handling the bad requests from the caller. Since the server is just spawning a process for processing the request, any bad request will result in termination of that process only, thus keeping the server alive for eternity. We have also implemented the supervisor for each module. In case the server dies due to some unknown errors, the supervisor is responsible for restarting the server.

Each module has a supervisor behavior. It makes it stable to any outside and inside influences. For more information see the chapter Supervisor behavior 5.9.

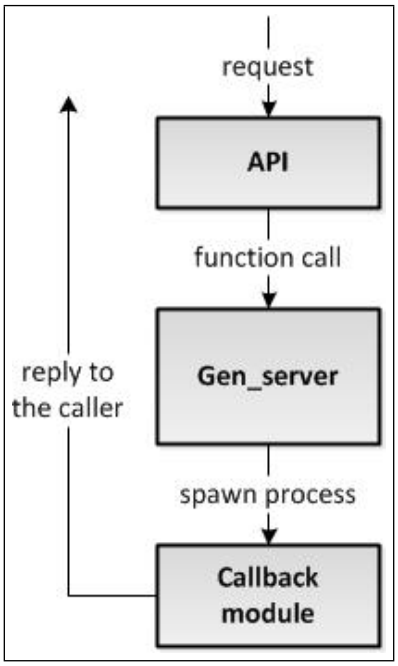

Figure 6: Module structure

4.4 System Core

The design of the Core module is quite different than the rest of the modules in the system. One of the reasons for doing this is that we wanted to follow the MVC [42] structure. This is depicted in figure 7.

The Controller is the heart of the system and all the requests into the system from the end users or external modules such as game node in our case, must go through this controller. The controller is responsible for deciding which operation should be initialized for any particular request. At the same time, the Controller itself does minimum work and just passes along the requests to the appropriate modules in the system. Controller is always free. Following this strategy, when the Controller only transfers the requests and not processing them, we make the Controller fault tolerant and reliable.

We have assigned Instance manager module to process the operations closely related to game instances. Some of these operations are:

- Starting a new game instance
- Recording game statistics
- Closing the running game
- Keeping track of game instances

Start new game instance operation sends the request to start the game application with unique id hosted on the remote node, no more than one time simultaneously. It means that the game instance application can not be started again while the previous application is not closed. Instance cooperates with 3 modules: User processes, Game nodes and Database. The idea behind the designing of Instance manager in such a way were:

- to isolate game instance operations
- to gather the database calls in one place
- to make the Game absolutely independent from the Lobby system

Record game statistics operation is used for storing information that is sent from the game side into the Lobby database.

Close Game operation intends for releasing the locked game instance. After the game instance has being started, it is locked and is unavailable for starting again until the request for its closing is received from the Game node.

Keeping track of the game instances is not an operation rather its a mechanism. When the Instance manager initializes, it reads all the relevant information of the game and its instances from the Lobby database and keeps them in its state as "available" and "unavailable" lists. So, whenever there is a request for the game starting or releasing, the instance manager is responsible for checking its availability or unavailability, blocking it if necessary and then maintaining the updated state in both

its internal state as well as in the DB so that in an event of crash, the state can be loaded from the database. Hence, the stability of the module is achieved.

All the requests other than which are related to the game instances belonging to that particular user are forwarded to the Client manager for processing.

Like in other modules, here also we have implemented the supervisor behavior for the system which monitors the application. All the components of the Core application are automatically restarted in case of abnormal termination.

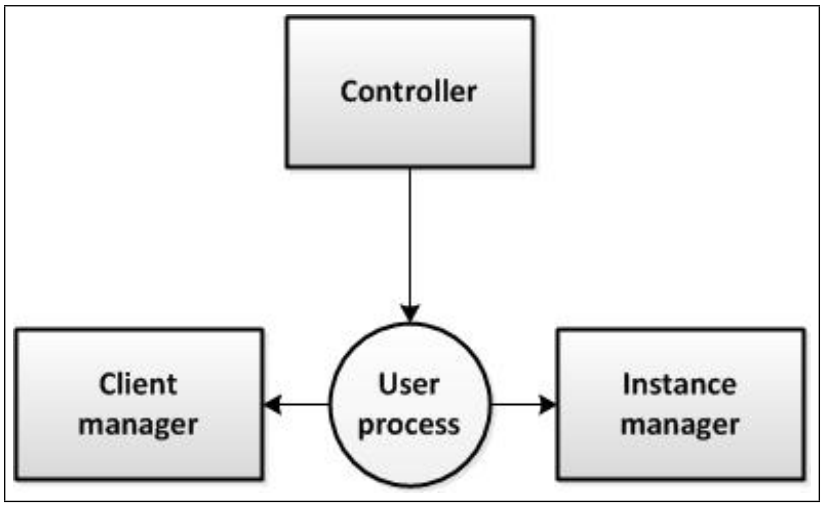

Figure 7: System core structure

4.5 Game Node

The Game Node is a part of the Lobby system. It id placed at the game server side and has two functions:

- to start game instance
- provide Game-Lobby communication

The Game Node structure is shown on figure 8.
In order to communicate with the Lobby server the Lobby API provided by the Lobby developers is used. When the Game Node receives the request to start a new game instance, it starts the additional process which starts the game, sends the notification to the Lobby server and terminates. If a new game can not be started because of some technical reasons the report is stored to the log file but the notification is not sent to the Lobby system. The notification is sent only if the game instance starts successfully! If the lobby system does not get any reply during specified time, the call times out signifying failure to start the game instance and an appropriate message is send to the user. The next element of the node is UDP listener. It always waits for the message from a particular port. If a message received, then the new process is spawned and message is sent to that process. Thus, the UDP listener's only job is to listen to that port and forward the incoming message for further processing. The last element in this figure is the Configuration file. It is used by the Game Node when a

new game is started or new request from the game is received.

The Game Node-Game communication is done through the Socket by using UDP [18] protocol. We have chosen this protocol for several reasons. First one is that the UDP is fast and simple to use. To establish the communication the sender just need to send the data packet to the particular port (or socket). As far as data is sent to the socket, the receiver is ready to get the packet. No confirmation packets are sent back to the sender. So the connection is established immediately. The second reason is simplicity. In Erlang [3], C++ [19], C#[20] and other languages which provide API for UDP connection, the functionality to use it is extremely simple. However, UDP protocol lacks reliability, so the sender is never sure about if the packet has been delivered or not. Meanwhile, such a situation when sent packet is not delivered is ruled out because both Sender and Receiver are run on the same machine. Thus, if a game provider decide to host game instances on different machines then it has to put the Game Node on each of its machines! Before using, the Game Node should be accurately configured. How to configure the Game Node see Appendix A.

In the Game Node-Game communication, Game Node is always the listener or the receiver. It does not send any data to a game through the socket rather than waiting for the incoming packets. Game is always the sender. It does not receive any messages from the Game Node. Thus, the Game Node-Game communication currently is only one directional. It is still possible to change the logic and make this communication bi-directional. How to do that see chapter Future work 8.

In the Lobby server-Game Node communication, Lobby server does not know where the Game Nodes are hosted. They are not registered in database or their IPs are not stored in the configuration file. Instead, each Game Node knows only about the Lobby server. This setting is also configurable. Where the Lobby IP is stored and how to change it see Appendix A.

The developers of the Lobby system does not put any restrictions on the overall number of Game Nodes run in the system. The only rule is that only one Game Node can be run on one machine.

The Game is able to send messages to the Lobby system by using Lobby-Game API. The game developer does not have to care about how to establish communication between Game and Lobby and how to send the messages to the Lobby. Everything is done inside the API. How to use it see Appendix A.

4.6 Lobby API for Unity

We have provided two API for the actual Unity [2] game developers of which one of them will be used in the Server side of the game development and the other one is to be used with the client part of the gaming. The client API sits infront of the actual gaming client such that this API will contain the function to pass the user's identity into the game. There is one another API provided for the Server which is resonsible for sending the messages to the Lobby system. There is no direct communication

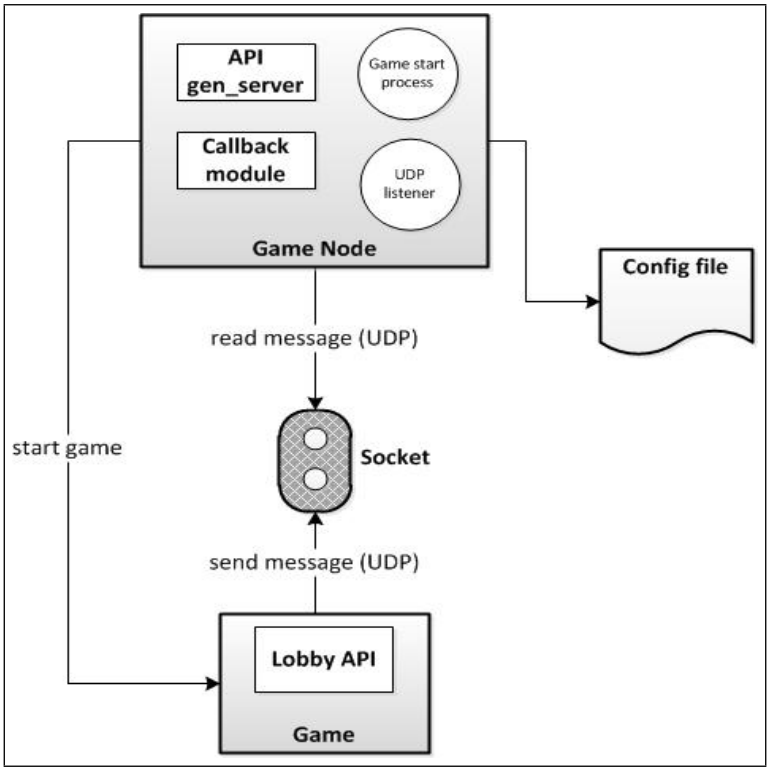

Figure 8: Game node structure

between the API and the game lobby or the game node but the API only knows about an port to which it is supposed to send all the requests to. This port on the other side is being listened to by the game node as explained in section 4.5. These two APIs are the only things that makes communication possible between the Lobby system and the actual game.

5 Implementation

In the following sections we will describe how we have actually implemented the design that described in the design section. Here, one can find the technical details of what, how, when and where the request are passed and processed and the response is sent back.

5.1 General Module

As seen from the design from fig 6, the general module structure consists of three layers. The first layer is the publicly available API which is called by the other modules in the system to send the request to that particular service. For example, if the module is the authentication module, the the publicly available API will be "auth_api". Since the API is the gateway to that particular service, it knows only about one module running behind that API and a function to call. The API will then do a function call to the actual "gen_server". For authentication module, this will be called "auth_srv". The function will internally convert the function call to the synchronous call to the gen_server. The call is handled in the server where it spawns a new process to process the request. The process is spawned with the general structure of the callback module, in authentication's case "auth_mod". The callback module is then responsible for matching the request and handling it accordingly. Also here the Id of the caller of the API is sent to the spawned process which will be later used for reply. The spawned process after processing the request will use the gen_server reply function to give the response back. Here, the response is directly sent back to the caller of the API, not the gen_server. After the response has been sent back, the process dies.

5.2 System Core

The Lobby Core is a bunch of processes which are running on the same node. The module follows a bit different structure that the previously defined general module structure. The Core contains a part of the MVC model [42], namely Controller. All the request from the framework (Nitrogen [5]) go through the Controller. The Controller is the only component which knows about all the other services in the system and how to call them.

The Lobby Core mainly contains four components namely:

- the Controller
- the User node
- the Instance manager
- the Client callback module

Each components mentioned above are responsible for different operations depending upon what the request is and from whom the request is. For instance all the processing when the user is not logged in is done by the controller itself but otherwise it will just relay the incoming message to different services depending upon what the

request is.

Some of the features that are handled by the controller itself are

- Register
- Login
- Logout
- Check status
- View games

The registration of the users is a straightforward process as we already have the service to do that, which is the authentication service. So, the controller will just call the external API of the authentication service to add the user into the database where the service will check if the user already exists or not and if not then that particular user is inserted into the database and the response is received by the controller and sent back to the framework to display the result of the registration operation. If the registration process fails, in that case also error message is sent to the framework for display.

Another operation handled by the controller is the login process. The login process includes three steps. First one is authenticating the user's identity. This is also a job for the authentication service, so it will use the login function provided by the authentication API to check if the user is who he/she claims to be. If the user is not authentic then an error message is sent back to the framework instantly. But if the user is a valid user, then the controller will call the supervisor to start a child process which is the User_node module. This is done so that from now onwards if any request from that user comes, then it is this process that will handle the request and give the reply back to the framework. Also if there is any malformed request from that particular user then only that user will perish not the entire users that are connected to the system since only that user's process will crash and terminate. Still the user process is being monitored by the top supervisor which will restart the user node in case of any abnormal failure. We also keep track of in how many places a single user has logged in from. So, user can have more than one session but in either case, only one process is created for that user's entire request from all the sessions.

In a typical web application system, the Controller does not need to be bothered about the user being logged out but since in our system there is a user process for each user, the process has to be terminated when the user logs out. So, for this reason the request is sent from the framework to the controller to kill the process and then the session in the framework is cleared. Since we allow multiple sessions for each user and the controller is keeping track of the logged in sessions, so it should check how many connected session the user has before terminating its process. Say, if the user has two connected session and the user logs out from one of them, then in such case the process should not be terminated as it is still required to process for the other connected session. So, all of these condition are checked thoroughly in the controller and the decision is taken depending upon the situation.

The Controller is also able to check if the user process is still alive or not. This is typically useful in the cases where the user process dies unexpectedly and even after multiple tries, the supervisor is not able to restart the it. In such cases, the framework will check for the status and if it is false then the user will get redirected to the login page to start over again from the login process.

The Controller is implemented as gen_server and follows the OTP standard [3]. When the message is sent the Controller does not wait for the reply, so it is always free.

Last but not the least, there is view games. View games is the part of the administrative service. But because we want the users to be able to see the list of games so that they can choose to register if they find something interesting to play. So the users don't have to be logged in to actually see the list of all the registered games in the system but to be able to play them, they have to be registered. We have already an implemented a version of view games in the administrative service, so here we will just call the same as anonymous user and then reply the result to the framework for display.

The User node is a separate process inside the web framework node which is spawned after successful user login and registered with that username. So each user will have a spawned process of the user node module which is of the gen_server behavior. This process is responsible for handling administrative and well as the client user requests. It is responsible for the administrative requests like add game, add game instances, view game instances, starting the registered game instances, updating the instance manager with new game instances and so on. All the request except updating the instance manager and starting a new game instances are sent to the administrative service which will process the request and send the response back and then from here onwards to the framework. The rest of requests are sent to the instance manager for further processing which will be explained after this. The user node is also in supervision of the web framework supervisor, so in case of abnormal termination it will be restarted and the user process will die only when the logged in user has logged out from all of this connected sessions.

The Instance manager is a separate process as well which runs on the same node as the Controller and the User node. It is implemented as gen_server and follows OTP standards. It maintains the state of all the games and the status of all of its available and unavailable instances. Instance manager implements the following operations:

- Starting new game instance
- Recording game statistics
- Closing the game
- Keeping track of game instances

Start new game instance operation is implemented as call back function. It does not spawn a separate process and is done within a number of steps. First one is to find the available game instance for the requested game. To do that we have to take into account that find the available instance in the database by sending the select request is not a good idea at all. The problem here is in the database or more specifically in

the ability to support locking mechanism for such operations as reading, writing and data modifications. If the database does not support locking mechanism and get two or more simultaneous requests for reading the available game instance then one game instance will be started more then one time. This situation should not be allowed according to the system design 4.4 and basic logic of game implementation. Thus, to be isolated from the database internal logic we implemented our own locking mechanism in Instance manager. The key point here is Instance manager initialization process. When the Lobby system core application starts (see the Appendix A how to start the system) it reads all the games and game instances from database. When the Instance manager receives request to start a new game instance and the available instance exists then it blocks this instance immediately. Thus, if more than one requests are received at the same time they will never get access to the same game instance. The game instance will be assigned to one process only. If the available game instances is successfully found then the Instance manager broadcasts the request to all the Game nodes (see the example 5.10.2). The request contains game instance id which is unique for each instance registered in the Lobby database. The broadcasting mechanism is used because Game nodes (see sections 4.5 and 5.7) are not registered at the Lobby system. It means the that Lobby server or the Instance manager in particular does not know about where this or that game instance is hosted. So, the game host are free to place their games where they want without notifying the Lobby system. Mainly this technology allows the game host to be absolutely independent from the Lobby system. What happens if there are no available game instances? The request will not be sent further.

Similarly, the operations such as recording of game data and closing of game instance are also handled by storing the incoming data from game node into the database and as for closing the game, the status for that instance is made available in both the state of the instance manager and also an update message is sent to the database regarding the same.

Client callback module is a number of functions implements the logic does not related to the game instances.

5.3 Authentication

We have implemented the authentication module as a separate service which is run separately on a node as a standalone process which follows the general module structure. So, whenever the other services need to interact with authentication, they should use the publicly available API that is provided by the authentication service. The API will then call the only exported function in the server module where it is converted into an asynchronous call to the gen_server. The call is handled in the server by spawning a new process from the callback module of the service to process the request and then the response is sent back to the caller of the API. Since we spawn a new process for each request, the spawned process dies after it has processed the request or it will just crash and terminate in case of any malformed request. In either case, the server which is handling all the requests never care about the response since the reply is directly sent to the caller of the API, so eventually this server will never crash or terminate. But still as a precautionary measure, se have implemented a supervisor

which is responsible for restarting the server in case it fails.

Our current implementation of the authentication module include following features:

- User Registration, which includes encryption of the plain password using MD5 hash function before inserting to the DB. Before applying the hash function, a random 4-byte random number is generated, "Salt" and appended to the end of the plain password
- Deleting a user
- Verifying user's authenticity, where the password provided by the user is first hashed using the salt generated during the registration process and then checked for equality since MD5 is a one-way hash function.
- Changing user password, where the user is first checked for validity using the old password and in case of success validation only the new password is accepted and stored after the encryption with again a new Salt.
- Updating user profile, the profile of the logged user will be updated with the latest changes.

The authentication module has to store all the users information in the database. It will store all the information in the "User" bucket (in terms of Riak [15]). The authentication service does not care about which database is being used but since we are using Riak [15] and Riak requires some pre-configuration to store the data. The bucket need not be present before the insertion, but even though Riak is considered a schemaless database, a schema for indexing the values and fast searching must be pre-configured before inserting or updating any thing. We have made script for this which is available in the "schemas" directory and it needs to be run once before using this service. This does not mean that authentication service is totally dependent upon database and if the database is changed then the authentication service also needs to be changed. The above step is done depending upon which database one uses and rest is just dependent upon the API provided by the database. If for instance, Riak is swept with say, CouchDB [11] then depending upon CouchDB the pre-requisites needs to be configures such as creating the "User" database and then creating the views for indexing and then the only change needed will be in the database API not here in the authentication service. Hence, we say that this is a standalone service independent of other services.

5.4 Database

Similar to the authentication module, the database module is also a standalone service which follows the general module structure and runs on the same node where the database (Riak [15]) is running. So, the only requirement here is that the "db" node must run on the same machine as the running database whichever that may be. Our database API is the gateway for our internal system to the actual database. The database service provides an external API publicly available to all the components in the system. Like in authentication, the API does the function call to the server

which converts it into synchronous call which is handled by spawning a process of the database callback module to process the request. Now, unlike the authentication where the callback module is the end processing unit, here the callback module will be responsible for accessing the actual database and getting response from the database and then converting them into suitable message structure to reply to the request. All the functionalities regarding the database are kept in the callback module. Here also since we spawn the processes for each request our implemented db server will never crash but still a supervisor is implemented which will ensure that the server is restarted in case it dies.

Our current implementation of the database service includes the general requirements from any database which are as listed as follows:

- insert for inserting the key and its corresponding view into the bucket
- select for retrieving the value by searching through the key
- update for updating the values which matches the key
- delete for deleting the key-value pair which matches the key
- search for retrieving the key by searching through the values
- store and install schema

Since we have used Riak [15] as our database, the implementation of the callback module is dependent with Riak. If one wishes to change the database to something else like Mnesia [26] or CouchDB [11] or even SQL databases like MySQL [23] or Oracle [24], then the only change required is in the callback module but one has to make sure that the return values of the function match with the existing one.

Riak is a key-value store and it stored all those pairs in a what is called as a "Bucket". A bucket can store virtually infinite number of unique key-value pairs. Values are the lists of more key-value pairs which signifies the key. All the key and values are stored as binaries in Riak so the callback module is responsible for converting the string to binaries before performing the insert, select, update or delete operation. So in general, the insert operation will take the Bucket name, a key and a proplists as the values for insertion. This is not always true since in some cases we need the keys to be dynamically generated and not pre-specified. For instance, in our case, in the "User" bucket the key was the username which is always unique and it is pre-specified by the user during the registration process. But, in case of "Games" bucket, where the user has no control over the key, so they have to be generated sequentially from the system. For such case, we have an insert function which takes only the values and the key is generated on the fly and inserted into the database. So, here a question arise how do we generate the key and how do we ensure that we will not have any duplicate keys? Since we spawn multiple processes for each request, an normal counter mechanism would fail since two processes may read the same value before either one of them has increased it. We need some locking mechanism or an auto-increment feature which is provided by most of the SQL databases. Since there is no way to achieve this through Riak, so we had to search for an alternative. We found one another small key-value store named "Redis" [40] which included the ACID

property required by the database. For our system , we have currently used Redis for three buckets “Games”, “Game_instances” and “Game_instances_run”. While inserting a game, a connection is established with the Redis server and request is sent for the latest “game_id”. The assurance of the uniqueness and non-duplicate ids is left for the redis.

Similarly, the select function takes the bucket name and the Key to search for. Here also we have implemented two versions of the select. One of them take empty list as an argument and returns all the key-value pairs in the “Values” that matches the “Key” and the other one takes a list of keys which are then matched against the key of the key-value pairs in the “Values” that matches the “Key” and only those key-value pairs are returned in the order of the specified list.

The update function also has some special properties depending upon how the information in stored in the values section. Normally, the values section would contain a key-value pair where both are strings but there might be cases where the values in the key-value pair is not a string but a list of string. So in such cases, the update will have to add/remove an item from the list not replace it. So we have considered these options also while implementing the update functionality.

The delete function will take Key and the bucket name and then removes the Key and its associated values from the bucket.

The search function has a pre-requisite that the bucket has to be index before the above mentioned operations so as for it to work. If the bucket where data is stored is not indexed this operation will not crash however it will always return an empty list ([]). For indexing a bucket, a schema must be written (which is shown how to write in the Riak’s official wiki site) and since we are using Erlang as an client, a pre-commit hook or post-commit hook must be installed with the bucket. After the bucket has been successfully installed, we are ready to use the search function. The search function will take the bucket name where to search the data for and a search criteria to match against. The Riak search uses the Lucene search syntax [72] as the search criteria.

```
Lucene Search Syntax:
Key:Value_to_search
```

So if we store username as the key and [“fname”,“Myname”] as values and we have it indexed for the search syntax will be “fname:Myname” as a string as argument to the search function.

It is to be noted that every mentioned operation works in the binary mode and the conversion is done internally in all the functions so that it is compliance with the Riak database.

The last function that we have is the function which takes a list of arguments and generated a schema files, stores it as a file in the system, then sets the schema to the index in the Riak and then installs the KV hook to the bucket. This function is helpful when we have do generate a bucket dynamically when a new game is added

to the system and it is upto the game which values they wish to index, hence giving them more flexibility while creating the actual game.

5.5 Admin

The admin service is also a standalone service which follows the general module structure and is responsible for handling all the requests that are generated by the admin user. Like authentication service, it also has the external API, server and a callback module. The server is responsible for handling each request by spawning a process which dies after processing the request and sending the response to the caller of the API. Likewise, a supervisor is implemented for any fail-overs which will restart the server if such a case arises.

Our current implementation of the admin service includes the following features:

- Adding new game
- Deleting an existing game
- Adding new game instance
- Deleting an existing game instance
- Viewing games
- Viewing game instances

Since the admin module is responsible for game and game instances creation and deletion, it has to communicate with the database, but like in authentication and db module, there is no prerequisites required for the admin module. In admin everything regarding the database is done dynamically.

As we already know that in Riak [15] we can only do the key based search if the bucket is not indexed. So a schema must be previously written and stored beforehand. But this cannot be done for adding the games as it would be impossible to know which are the fields that need to be indexed as each game would have their own set of different fields to be indexed, for instance for a football game, the number of goals scored, number of yellow cards or red cards received would be the possible fields to be indexed while for the shooting game, number of bullets fired, number of kills and so on will be indexed fields. As the system is built with no restriction to the games so predefining these fields would not be good. So, that's why we create the schema depending upon the game requirement during the game insertion and store and install it in the Riak server. The "add game" has two variations one where the fields to be indexed are specified and the schema is generated and the one without the indexing such that it is search-able only through the key. This is totally dependent upon the game developer. The thus created schema is also locally stored for cases when the riak server crashes or when riak nodes needs to be increased and all they have to do is run the schema on the new nodes and they will all be able to index the games.

In Unity [2] gaming, the each game will have many instances that the users can create and each instance can have multiple players. But the instances for a particular

game has to be predefined that the users can create. The total number of instances will be hidden from the user however, the players will only see that there is a game and they can start a new game or they can join the existing running game. So, the admin interface in the system gives the game developer the opportunity to add any number of game instances for a particular game.

There is also possibility to delete the created game and game instances from the system if one wishes to. Also, two functions for viewing the added games and game instances are available in the system. The instances are not instantly available for playing as soon as they are added. There are some configuration needed to be done in the game node which will be explained later in section 5.7. So, the admin will have to make it available through the web interface from the view game instances by checking the available checkbox.

5.6 Nitrogen

Nitrogen [5] is the web framework which also runs on a separate node and it is used for rendering the HTML pages which are displayed to the end users. Here, we have employed to follow the famous MVC model [42] of building the web applications. All the pages in nitrogen are the Erlang modules so each module corresponds to a single page being rendered into the HTML for display. Also, the directory structure of the Nitrogen is bit different with all the pages or modules kept in the "src" directory, all the static contents like JavaScript, CSS and images kept in the "static" directory within their own folders and the base templates for the page, if any, in the "templates" directory. The compiled version of the modules are kept in the "ebin" directory. One more thing to note is that the filenames that we provide for the modules become the part of the "URL" for accessing that module through the web browser. Say, if we have a module called "index.erl", then it can be accessed through the URL "http://localhost:port/index" and if it is "user_login.erl", it is accessible through URL "http://localhost:port/user/login".

Since we are using the MVC model [42], Nitrogen is only responsible for displaying and getting the postback data from the web pages, other than that it will not do any processing of data. We have some predefined templates with all the header and footer and the menu bars and all the module depending upon the status of the user will load one of the templates and the contents of the page in inserted into the template before displaying them to the end user.

5.7 Game Node

The Game Node shown on the figure 4.3 consists of API, Callback module and UDP listener. Apart from the general Lobby module structure 4.3 where the API and Gen_server are two independent modules here they are combined and represented as one unit. The Callback module represented as a bunch of functions realized the logic of Game Node. Another component of the Game Node is UDP listener. It is implemented as a simple non-OTP behavior process. The process on the figure which is responsible for the starting new game instance also implemented as simple non-OTP process.

As mentioned above the Game Node structure is different from the general Lobby module design. This change in design was done because of implementation peculiarity of Lobby-Game module communication. In details, as described in chapter 4.5 the Lobby server never sends requests to the specific Game node directly. Instead, it broadcasts the message to all the Game nodes in the global net. The broadcasting implemented as function call rpc:multicall(Nodes, gn_srv, start_game_instance, [InstanceId], 3000) where:

- Nodes is a list of nodes where Game node application is run
- gn_srv is the name of the main Game node nodule
- start_game_instance is an API
- InstanceId is a unique game instance id
- 3000 is a time in milliseconds. If the Lobby server does not get any reply during that period then the game instance releases

The receiver of such a call must be a process or a gen_server or fsm_machine. Thus we could not catch this call in flat stand-alone API and merged it with gen_server.

Not only the Lobby system broadcasts the message to the Game Node. The Game Node as well uses this Erlang tool to communicate with the Lobby system. The reason for that here differs from that explained above. The Game Node is designed as stand alone application which does not have any dependencies with the Lobby system. It delivers to the game developer without the Lobby server. According to Erlang, we are not allowed to make a direct function call to the module which is not mentioned at our application (simply it will not work). The only way to establish communication with the Lobby server is to use rpc:multicall mechanism.

In order to make Game Node stable, the implementation was done by using Application behavior. All the processes and modules are started in the supervision tree. Whenever a process crashes, it is automatically restarted by its Supervisor. See the chapter Supervisor behavior 5.9.

5.8 Lobby API for Unity

The lobby API is the only API that establishes connection between the Lobby system and the game. As described in the design section, it communicates with the Lobby system through the predefined port/socket. The current implementation of API has only one prespecified port to which it will send request to. It does not matter how many games are hosted on that machine but they will all send the message to the same port. It is desirable to have different ports for different games for efficient and faster processing but the current solution also works quite well. As mentioned earlier, there are two APIs, one at the server side and one at the client side. Both of the APIs are written in C#.

Let's see first how the client API works. Since we are using the web interface as the client, the client will load the game client provided by the game developers to

the lobby system and as soon as the client loads that game client (Web client) into their browser, the client's username, the server IP and port will be passed onto that API through Javascript by the Lobby system and then the game developer can use these information to connect that client to the specified game server and hence that particular game instance is able to keep track of the players connected to it.

There is one another API provided for the Server which is responsible for sending the messages to the Lobby system. The current messages that it can send is that the game server has closed; it doesn't matter normally or abnormally but when the server is closed it will send the message to the Lobby system and that game instance has to be made free in the Lobby system so that it can be used again for playing by other players. The other information that it sends is regarding the statistics of the game for that particular instance. The Lobby system needs to store the statistics of each instance of the game that has been played. So, when should the game send the data/statistics to the Lobby? It is totally dependent upon the game developers. We have provided an function in the API, which the game developers can call and it takes a string dictionary of data. The data from dictionary is then extracted and converted into key-value tuple pair, which is understandable by Erlang and is sent to the specific port as binary. Since the port is being listened by the game node and from there how things work is explained in the section 5.10.3

5.9 Supervisor behavior

In order to make any application stable it should have supervisor behavior. The supervisor behavior task is to monitor its children, based on some rules and take action when they terminates [3]. The children should be either gen_server, gen_fsm or gen_event. So the supervision behavior can work only within the OTP framework. The figure 9 illustrates the basic supervision tree. Whenever a child terminates abnormally the supervisor will immediately restart it. In order to prevent the deadlock (the situation when the child will always terminates) the supervision implementation allows to set up the max number of restarts in a row. If that limit is reached, then the supervisor automatically stops the restarting procedure.

To make the Lobby system simple to start we implemented each of its component as Application. Application is started, stopped and loaded as one union and keeps track of all the relations between its modules. Application gives developers additional level of abstraction and hides the process relations.

How to start the system see Appendix A.

5.10 Module integration

This chapter describes the relations of the modules while proceeding different operations. In particular, three operations are described in details: user log-in, start new game and game-lobby message passing.

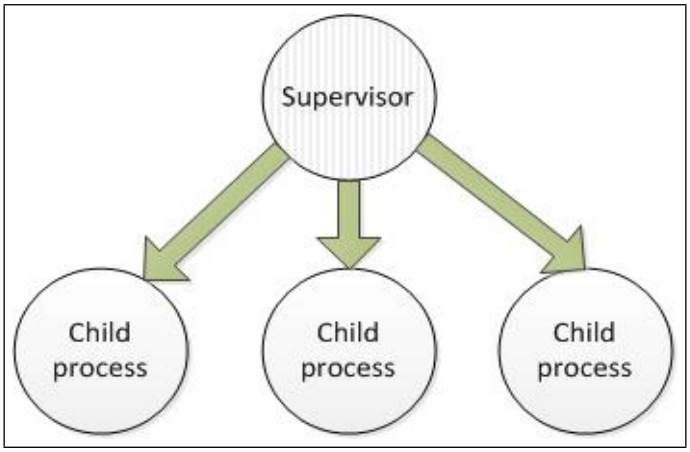

Figure 9: Supervisor tree

5.10.1 User log-in

The first operation which user initiates after coming to the Lobby web page is log-in. The process of user login is visually demonstrated in the figure 10. The request is handled by the Web Framework [31] and generates an appropriate request for the login process which is passed on to the Controller. The Controller identifies the request and a decision is made to send the request to Authentication module. Authentication module talks to the database through the database module and get the current user password stored in the database. The reply from the database includes an encrypted password and a 4-digit salt. The authentication module will now encrypt the plain password after appending the salt to it with md5 hash function and a comparison is made between the two passwords. The comparison result is sent back to the Controller. Controller evaluates the reply and in case of positive reply either starts a new fresh process or increments the counter or sends an error message back in case of negative reply. Before spawning a new process, the Controller checks if the user has been already logged-in in any of the previous sessions. If so, instead of spawning a new process, the counter for that user which is maintained by the controller is incremented by 1. After that, all the requests initiated by that logged-in user will be redirected to the spawned process.

5.10.2 Start new game

After a player has logged-in to the Lobby system, it is granted permission to start its own games, if available. All the players can see the list of available games in the Lobby main web page. Each game in this list is accompanied with a start button. If the game exists in the Lobby game list, it means that it has at least one available game instance hosted on the game developer side. Game instance is a game server represented as an executable application. Executable application here is a .exe file or any other applications available for being started without additional configuration and compilation phases. The Lobby design hides the actual available game instances from the end users. Only administrator is able to see and manipulate the game instances. How to do that and which functionality is available for the administrator, for all these details, see Appendix A.

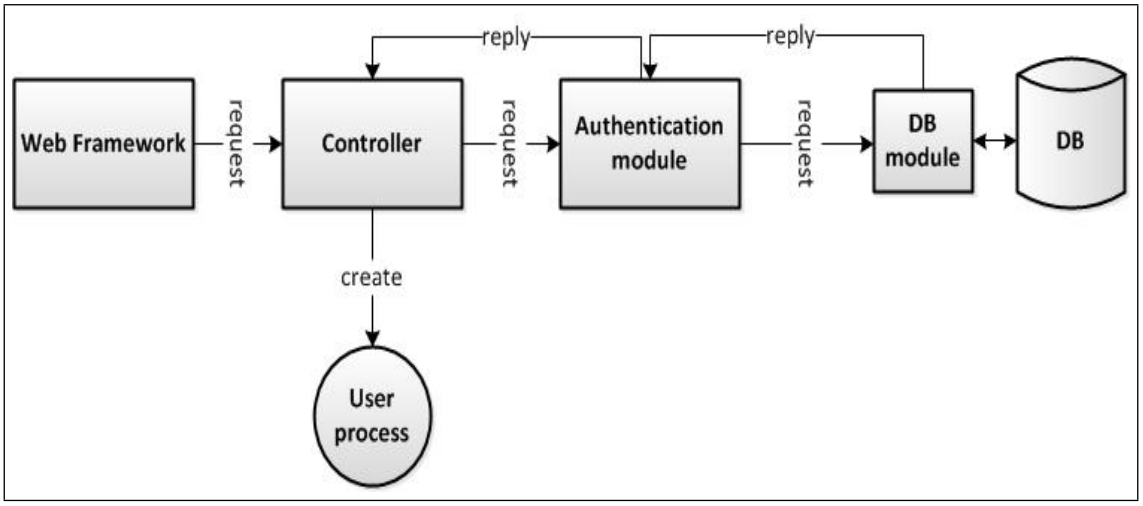

Figure 10: User log-in

When a player has initialized a new request to start some game instance, this request is automatically handled by the Web Framework. The Web Framework will make an appropriate request and sends it to the Controller (see the Design 4.4 and Implementation 5.2 sections). As far as the request is initiated by a particular user, it is passed to the process associated with that user (see the Design 4.4 and Implementation 5.2 sections). Then all the requests from different user processes accumulated in the Instance manager (see the Design 4.4 and Implementation 5.2 sections). If the game instance for that game is found, then Instance manager broadcasts the request to all the Game nodes. The Game Node which host the asked game instance, starts it and sends the reply back immediately to the instance manager. In that case Instance manager will make all the necessary manipulation in its state and the database and sends the reply back to the Web Framework.

Thus, we can see that the request is passed or processed through 5 modules with minimum of 7 requests being executed between the initial user's request and the final game starting.

5.10.3 Getting messages from games

Whenever a game decides to send some messages to the Lobby system it communicates with the port which is being listened by one of the processes in the game node. The socket is not shown in the figure 12) but this is how it actually works underneath. How this friendly communication is done in details see chapter 5.7. Game Node retransfers the message to the Controller. We have implemented the message sending as broadcasting. You can find these details in chapter 5.7. Controller does not analyze or manipulate the request rather it spawns a new process and sends the request directly to this process. The only duty of the spawned process is to send the request to the Instance manager. There all the necessary manipulations are done and database is updated.

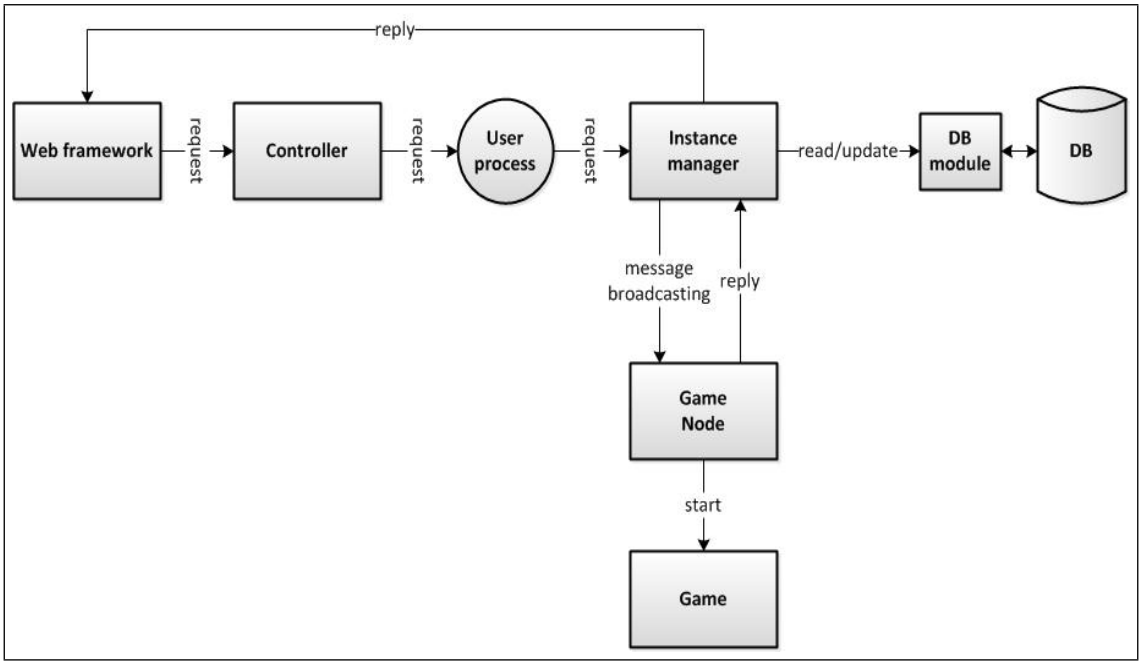

Figure 11: Start a new game

5.11 Instant Messaging

The Chat service enables players to chat with friends during game play. It sits between the Instant Messaging Server (i.e.Ejabberd [13]) and the Authentication Module for handling messaging passing between these two components, as shown in Figure 13.

The Chat service consists of three components:

- Ejabberd: Instant Messaging Server
- Chat module: a bridge between Ejabberd and Lobby System
- iJab: Instant Messaging Client [45]

On the server side, Ejabberd uses Mnesia [26] as internal data storage by default. In order to integrate Ejabberd into the Lobby system, the first thing that was focused on is to change the data storage to Riak [15] rather than Mnesia. According to the design principle of the Lobby system, a Chat Module is implemented that receives requests from Ejabberd and forwards requests to another services in the Lobby system. By having this Chat Module, Ejabberd would become general to the lobby system and it can be easily replaced by other instant messaging servers.

The Chat Module follows the general module structure that consists of three layers:

- API
- Gen_Server
- Callback Module

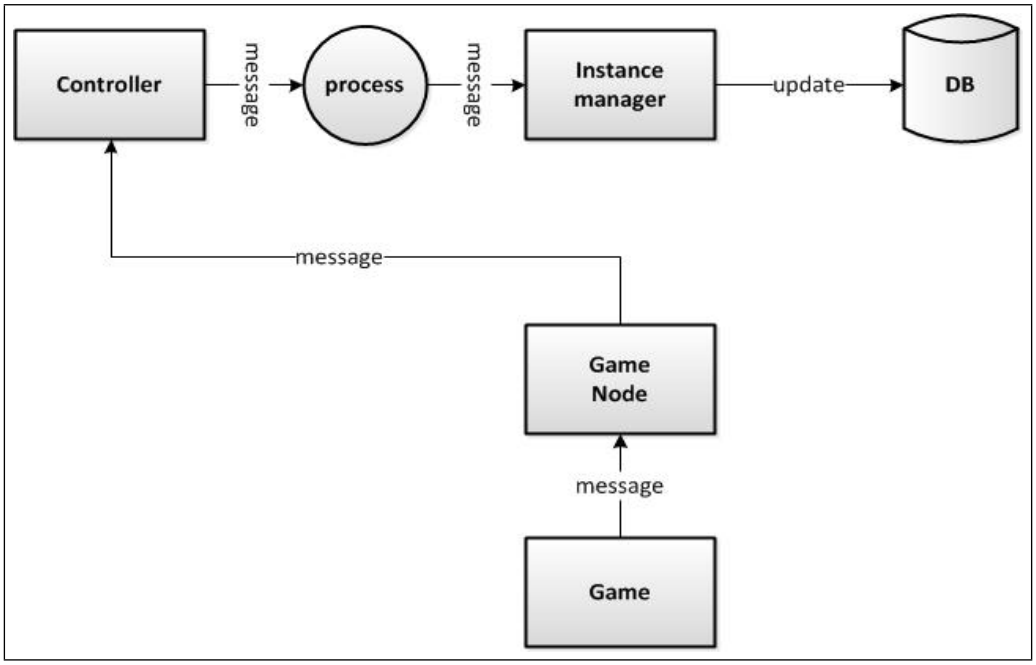

Figure 12: Game's messages

The API is public to all other services in the Lobby system. It is the only thing that Ejabberd knows about the Lobby system. Each request comes from Ejabberd must be passed into the Lobby system over this API. Currently the API contains a login function with two parameters: Username and Password. This function is called by several functions in Ejabberd when player's trying to login. These are where database calls to Mnesia in Ejabberd are replaced by database calls to Riak in the Lobby system.

Upon receiving a login request, the API will pass a synchronous message request on to the server. The server will response by spawning a new process with the callback module to process the request. In the callback module, a login function in the API of Authentication Module is used to authenticate players in the Lobby system. The Authentication Module will further talks to database (i.e.Riak) to verify player's authenticity. The Authentication Module will send the result back to the callback module of Chat Module, and the callback module will forward the result back to Ejabberd.

On the client side, iJab works happily with Ejabberd. It is running with Ejabberd's internal http-bind support as well as the web server support using the internal mod_http_fileserver module.

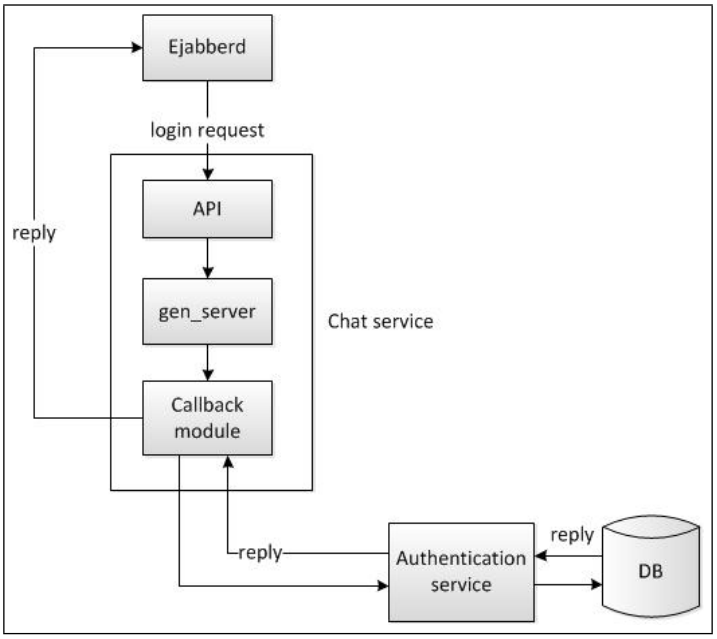

Figure 13: Communication between Chat Service and Authentication Service

6 Testing

Implementation is an important and time consuming part of the system development but testing it also another crucial part of the development process that requires a lot of time. It is this phase that checks if the system built is correct or not. The general conception of testing is to do it after the implementation phase but it is not the right one. The testing of the system should be done from the beginning while the system design and the on every step of the system implementation. There are various ways to test the system and one can argue that testing the system at every step is not possible. For the system that we have implemented we have two kinds of testing. One is the Unit testing where we test each module and its functions one at a time and then the final testing where we do the load testing.

6.1 Unit Testing

Since we are using Erlang for developing the system, we used the EUnit [36] library provided by the Erlang to construct the unit tests for our modules. As the name suggests, EUnit testing is used for testing each unit i.e. testing each functions in the modules independently of the other parts of the system. The testing is carried by calling the functions and then let the function do the processing and then matching the result of the function with our expectation. If both of them are the same, then the test is passed or else failed. In case of failure, it will also display the result of the function and that it did not match with the expected value. Since EUnit is also a part of Erlang, it comes with the support for parallelizing the tests for the multicore systems. Parallelism does not mean that the tests are carried out in random and cannot have control over which tests to run first.

The test cases can be kept inside the actual module being tested but this is not preferable. Normally , all the test cases are kept in a separate file and the test module should be named the same as the module being tested with "_tests" being appended to it.

We have used "Rebar" [37], one of the Erlang built application, for compiling our codes and one of the many features of Rebar is that it can also run the EUnit test case and generate the coverage report where one can see how much of the code is covered or executed by the test cases.

EUnit test is useful when there is no side-effects in the function being tested and if the function being tested has side-effects then if we make the generic calls to that side-effects as well then it is no longer a unit test and it will be very difficult to find where the function failed. So, to make the unit test independent of the side-effects, we have used Meck [38].

Meck is an Erlang Library, which is used for spoofing the side-effects (function calls). In Meck, we just specify which function in a module we want to spoof or mock and then add an assert function to that call, i.e. we specify the function and its arguments (if any) and just give a return value to it so that when that particular function is called, the return value is returned instead of doing the actual function

call. A simple example is as shown in figure 14.

```
1> meck:new(dog).
ok
2> meck:expect(dog, bark, fun() -> "Woof!" end).
ok
3> dog:bark().
"Woof!"
4>
```

Figure 14: A simple Meck example

For testing our system, we have considered the entire service as a module and then function as an unit. Say, if we are to test the authentication service, then there is no way to test the API and the server as they just do the function call and relay the message while all the logic is in the callback module but still we need to test that they work. So, in one context we are doing a system test for a service but a unit test for the callback module. The callback module is then again dependent upon other services in the system. This is where Meck comes into rescue. We spoof all the function that leads to calling the function outside the current service being tested. For instance, for authentication service, if we have DB calls, then we spoof all the calls to them and so on for other modules. This has helped us quite well to find the bugs and fix them in that service. We have covered more than 90% of the code in most of the services but there are some services where it is impossible to spoof the components and thus have the low test coverage. But still for those parts, we have manually tested them quite sufficiently to say confidently that they work as we intend them to work.

6.2 Load Testing - Tsung

For any system, it is always necessary to find out the breaking point of system or many requests can it handle before the system starts to lag behind in the response time and finally fall apart. Depending upon various kinds of application, there are various tools to do these kinds of testing. For our system, we have Tsung [39] as the load generator.

Tsung is an Erlang [3] built distributed load testing tool to test the performance and the scalability of the client/server applications. It is protocol-independent and current it supports to stress test HTTP, WebDAV, SOAP, PostgreSQL [25], MySQL [23], LDAP, and Jabber/XMPP servers. The key features of Tsung includes

- High Performance, simulation of large no. of concurrent users per computer
- Distributability, generating loads from various clients
- Support for large number of protocols
- SSL support
- Monitoring of CPU, memory and network traffic

- Recorder for recording the session and replaying it

For current testing our system we have recorded the login, logout and view games session from the Tsung. After completion of the recording, Tsung generates an XML file containing all the information and we just copy and paste it to the session part of the actual testing module which is also an XML file specifying where to find the server and client how the tests should be carried out.

Since we had implemented an distributed application, we did a series of tests with all the services on one single machine and then later distributing them over different machines to see if we had performance advantages.

Setup #	Machine 1	Machine 2
1	Database Webserver Webframework Authentication Module Admin Module Core	
2	Database Webserver Webframework Admin Module	Authentication Module Core

Table 3: Tsung test setups

Figure15 displays the result for the test when all the components are running on one machine. As one can see, the system is able to handle around 600 transactions/sec but the response time is bit slower.

Figure16 displays the result for the test when all the components except the core and the authentication services are running on one machine. As one can see, the system is able to handle a bit more than 600 transactions/sec but the response time is far better than when they are kept altogether.

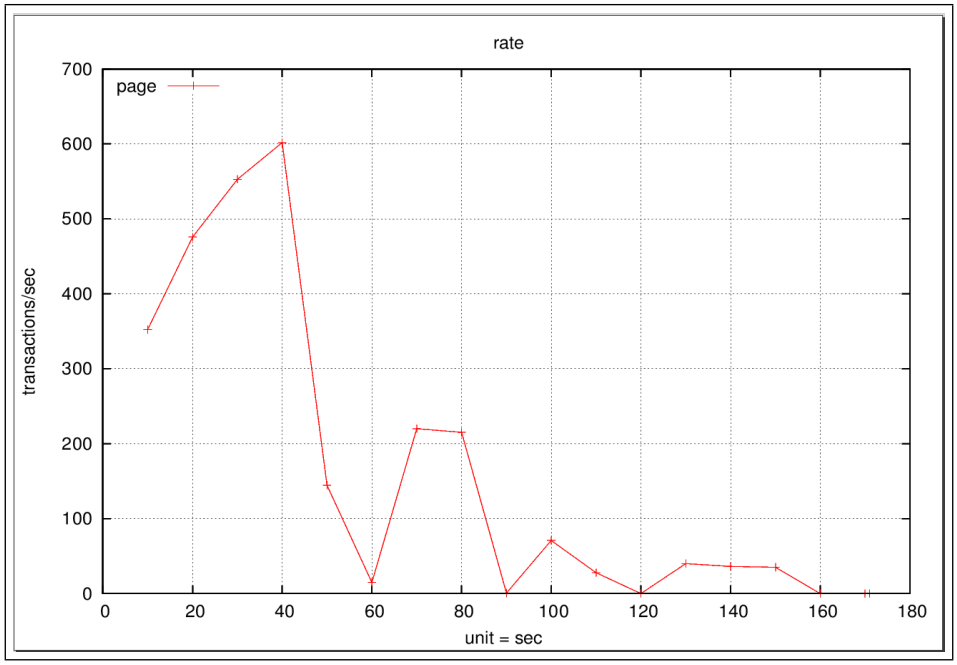

Figure 15: All in one node

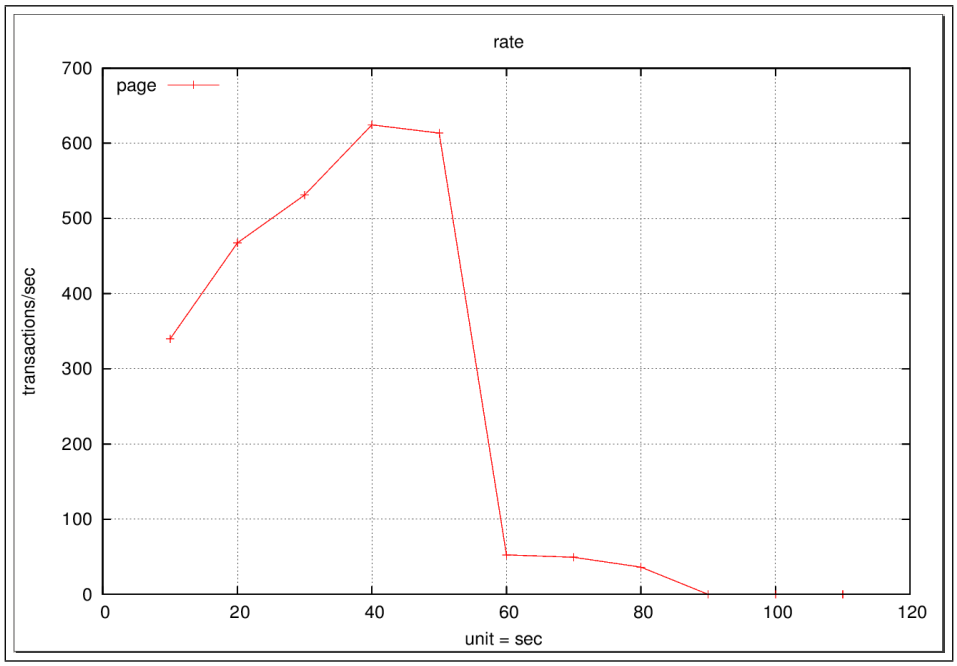

Figure 16: Core and Authentication separated

7 Problems and Issues

This section describes the problems and issues that we encountered during the project and how we managed to solve it (in some cases at least for our system).

7.1 Riak

The current version of Riak [15] does not support the latest version of Erlang [3]. As we were using the latest version of Erlang, we were stuck with this problem also for quite sometime which was fixed when we downgraded the Erlang to one version below.

We had a lot of issues with installing and running Riak, even with the older version of Riak in all our systems. We managed to run it one of our laptops but it just did not just work with the other two laptops. After trying for some time, we just left it as it is and just used one laptop as the database central. After being able to use Riak, we started implementing the basic database operations and all worked fine but when we tried the indexing of the values of the bucket, it did not seem to work even after following all the instructions given in their official webpage. After trying out a lot of variations, we finally decided to ask Rusty what we were doing wrong and got to the conclusion that they had missed few details in their documentation and after the fix it worked pretty well which was in a very frustrating at that time and pleasing to see when it actually worked.

Another limitation of Riak is that it does not have the auto-increment feature that are available in relational databases for automatic sequential key generation (primary key). So, we had to find an alternative option to this and after doing bit of research on this, a lot of people recommended to use "Redis" [40] as an option. Thus, for this reason only to generate auto ids we had to use Redis which worked quite well for us.

7.2 Ejabberd

After painfully fixing all the dependencies which were not mentioned in their documentation, the Ejabberd [13] server ran successfully on the one of our system. But it ran in the machine locally only. So, to make it accessible from other computers, as suggested in the documentation, we changed the hostname in the "ejabberdctl" configuration file. Theoretically, it should have worked but it did not. The Ejabberd server crashed with error. We searched and tried to fix the error but still the same error. After going through most of the source code we found out the template which generated the configuration file and changed the hostname there. And after the compilation of the source code again, it resulted the change in the same place that we changed earlier in the configuration file but this time the server started instead of crashing. We were surprised to find that the hostname had some dependencies in the source code and for some reasons it did not read from the compiled configuration file. Aren't configuration files are written to avoid having to recompile the source at the deployment site?

Ejabberd comes with Mnesia [26] database as default but also has support for some other databases such as databases that run under ODBC [90] bridge, Ldap [91]

and Pam [92]. But it had no support for Riak, which we were using. So, our task included swapping the current database with Riak. But Ejabberd was developed with a lot of dependencies to Mnesia, so swapping out Mnesia with Riak, I would not say impossible, but would rather take a longer than we could afford to spend. We also found out that there has been a project going on to swap out Mnesia with Riak. So, to make work with our system, we just swapped the few functionalities in Ejabberd to call our system that we actually use and made it work with our version of the system. We know that this is a ugly hack, but they are made such that when the Ejabberd comes in with support with Riak, which is expected soon, it will still be easy to swap and be compatible with our system.

7.3 Testing

During the testing of our system, we found in a situation where we could not do the unit testing for all the modules in our system since the modules were very much dependent upon each other. Some functions in the modules do not serve any purpose, but they just call other functions in the modules and provides an extra layer of abstraction. For instance, the API's in the system were difficult to test as they only call the gen_server. So, to solve this problem, instead of treating each function as a unit, we decided to make the entire service a unit. So, for testing the function for a service, the function calls that modules were not mocked away but simply let them call that function and only mocked (spoofed) only if there is a called to the function of different service that is being tested.

One more problem that arose during our testing phase was in the load testing phase. Since load testing would generate a lot of requests, there were a lot of open TCP [93] connections and the framework(Nitrogen) crashed with "file descriptor" error. This was due to the fact that the limit for number of files that could be opened in the system was exceeded. The system came with the default of 1024. We found out about this problem and later fixed it by increasing this count. The command to check and increase the count is as follows:

```
ulimit -n #sh
ulimit -n 4096 (some greater number, preferrably multiple of 1024)
```

7.4 Game Node

As mentioned earlier, Game Node resides at the game hosts side and is a standalone application. But this was not the case when we first designed it. At the beginning we designed the game node to be a part of the lobby tightly coupled with as we did not know how the Unity games behaved and how to communicate with them. But when we started to implement this model, it did not work out as we thought that it would.

The first problem that we had was with starting the game. Since Unity [2] required to start the game from running the ".exe" file, we had to use Erlang shell commands to run the files. But that was fine, but after opening the game file, the the process would wait until the game was closed, thus the node was able to handle one game start at a time only. To fix this problem, we made a newly spawned process handle

this. But then again, we found out that there will be multiple game nodes and each game node will be have different game instances and how do we know which node to send message for starting the node, so we decided to broadcast the messages to all the game nodes. By doing so could also control the nodes that need to reply to the lobby, which is the case when the node has that requested game instance. All other nodes would just ignore the request and continue waiting for the new request.

Generally, we had a supervisor monitoring the all child processes, but in the game node this was not possible. Here, we had the game node server which needed to be monitored and then the listener process that needed to be monitored. But since listener was a normal spawned process and the since only the process that follow OTP standard can not be monitored, so it was impossible for it to be monitored. To fix this issue, we had the listener process spawn processes for handling the incoming messages. Now, since listener process is not doing any process by itself, so it will never crash and in case it does means that there is something wrong in the server, so we have linked the server with the listener process so that if something goes wrong in the server both of them crashes and the supervisor will restart the server which will in turn start the listener.

8 Future work

8.1 Game-Lobby two way communication

As was mentioned in chapter 4.5 Game-Lobby communication is implemented as one directional. It was done for couple of technical reasons and we were tightly restricted with time. One directional Game-Lobby communication makes the Game-Node implementation simple and clear to understand. Meanwhile, it puts one restriction on the system functionality. Lobby system is not able to manipulate with the game whenever it is necessary.

The current program implementation might be extended. For that you need to:

1. Start Game Node
2. Modify the configuration file
3. Start UDP listener
4. Add "trap_exit" flag
5. Start new game instance
6. Terminate the UDP listener
7. Keep track of system stability
8. Change Game-Lobby API

Start Game Node. Currently the UDP listener described in chapter 4.5 starts concurrently with the Game Node during its initialization phase. This operation should not be done.

Modify the configuration file. The unique UDP port should be assigned for each game instance. So the new parameter should be added to the section "instances".

Star UDP listener. The process which is responsible for starting UDP listener should be spawned when the new request to start a game instance is received. It is important to take into account that the new UDP listener should be started only if the game instance is successfully found in the configuration file. After the process is successfully started, its PID should be registered and kept in the Game Node gen_server State. The gen_server State should contain the following information about the registered process: PID and Game instance id.

Add "trap_exit" flag. The process flag "process_flag(trap_exit, true)" should be established in the "init" function of gn_srv module.

Start new game instance. This mechanism should be changed as well. Currently all the games send the messages to the same UDP port (1234). The additional parameter "port number" should be added to the function call "open_port" which is responsible for starting game application. This port number should be read from the

configuration file.

Terminate the UDP listener. The UDP listener should terminate normally when it gets the message of Game termination. The message should be sent to the Game Node gen_server as well. Game Node should process the message and delete the information about the process from its State.

Keep track of system stability. All the abnormal process terminations has to be handled and processed. Current version of the Game Node is developed in such a way that if one of the two processes, gen_server or UDP listener, terminates then the second process terminates as well. After that the supervisor restarts the gen_server and the UDP listener is restarted as well. This system behavior does not suite if there are more than one process monitoring the game instances. In that case the Game Node gen_server should handle all the abnormal terminations of its processes (that is why we need to set up the trap_exit flag). The abnormal termination should be handled in handle_info callback function. The terminated process should be immediately restarted and registered PID should be rewritten in the State. Thus, the Game Node plays the role of supervisor for the UDP processes.

Change Game-Lobby API. Game-Lobby API needs to be changed as well. Besides being a message sender the Game should always listen to a particular UDP port. The UDP listener should be initialized when the game starts.

8.2 Lobby system and non-Unity games

Initially the Lobby system was planned to work only with the Unity [2] games. However, it might be extended to support different game engines. What is important here? The main factor which restricts Game-Lobby compatibility is the type of Game Client. The Lobby system was designed to work with online games where the Client is hosted in Web Browser [47]. The Web Client is described in chapters Working environment/Unity 3.5 and Design/Unity 4.6. The main advantage of current implementation of the Lobby system is that the system does not depend on inner game logic and implementation. The only thing we have done we provided the mechanism for Client delivery to the Player machine.

To start non-Unity games, Game Node should not be reimplemented. The current mechanism allows to start any types of games on different platforms. The only thing which might be changed is the the way to start game instances. Instead of running the application directly Game Node might run the startup script which in turn starts the actual application according to the game requirements.

8.3 Conferencing and video chat

Currently, we have looked into imposing the chat module to the lobby system so that the users can chat with each other before starting the game. However, the chat will be still available even when the game is being played. For this we have used Ejabberd

[13] as an simple application and integrated it into our system. But it would be an additional feature if the system had at least an audio chat or if possible even video chat. This could be still expanded to incorporate the conferencing mechanism for text, audio as well as video chatting. From what we have heard, Ejabberd developers are working on putting audio chat into it, if it is so, then it could be easily added into our system as well without much change of code.

8.4 Lobby as a bunch of stand alone applications

Another change that might be done is to change the inter-communication behavior between the all the modules in the lobby system. Currently, to distribute the Lobby modules on different machines we have to copy the whole system to that machine to keep the dependencies between modules as the APIs are inside the service. But if we separate out all the external APIs to one folder then to run a service all that is required it that service and the API folder on the machine where the service is being run.

Also, the gen_server running in the services are declard to be "global". So making a service run on multiple nodes at the same time is not possible. To make the Lobby modules to run multiple copies of itself possible, the server should be declared "local" and the gen_server calls should be replaced by the RPC (remote procedure call) calls.

9 Conclusion

During the project, we have come up with and implemented an architecture which were highly distributable in nature with each component in itself being able to distribute and expand to the extent required in the future case without much changes to the current system. They are scalable horizontally as well as vertically.

We have made a successfully system where one can host any games that is built with unity Gameing Engine. We have tried this with few games provided by Pikkotekk and it has worked quite well in those games. With the lots of Unit Testing, we ensured that the system is stable enough and with the load testing we ensured that the system is able to handle lots of loads with failover and recovery mechanism thus giving us with the high availablity required by the system.

References

[1] Pikkotekk [Online]. Available at: http://www.pikkotekk.com/. [Accessed 04/04/2011].

[2] Unity [Online]. Available at: http://unity3d.com/. [Accessed 04/04/2011].

[3] Francesco Cesarini and Simon Thompson. (2009) *Erlang Programming-A Concurrent Approach to Software Development*. 1st ed. O'Reilly Media.

[4] The Unix system [Online]. Available at: http://www.unix.org/. [Accessed 04/04/2011].

[5] Nitrogen Web Framework [Online]. Available at: http://nitrogenproject.com/. [Accessed 11/04/2011].

[6] Yaws [Online]. Available at: http://yaws.hyber.org/. [Accessed 08/04/2011].

[7] SpringLobby [Online]. Available at: http://springlobby.info/landing/index.php [Accessed 11/04/2011].

[8] RTS game engine [Online]. Available at: http://skatgame.net/mburo/orts/ [Accessed 11/04/2011].

[9] Erlang Web Framework [Online]. Available at: http://www.erlang-web.org/. [Accessed 11/04/2011].

[10] Concurrency Oriented Programming in Erlang [Online]. Available at: http://www.sics.se/~joe/talks/ll2_2002.pdf. [Accessed 11/04/2011].

[11] CouchDB Project Website [Online]. Available at: http://couchdb.apache.org/. [Accessed 11/04/2011].

[12] International Computer Games Association (ICGA) [Online]. Available at: http://www.icga.org/. [Accessed 19/04/2011].

[13] Ejabberd Community Site [Online]. Available at: http://www.ejabberd.im/. [Accessed 25/04/2011].

[14] Ejabberd Module Development [Onlin]. Available at: http://www.process-one.net/en/wiki/ejabberd_module_development/. [Accessed 15/04/2011].

[15] Riak database [Online]. Available at: https://wiki.basho.com/Home.html. [Accessed 25/04/2011].

[16] Dynamo Paper [Online]. Available at: http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf. [Accessed 25/04/2011].

[17] CAP Theorem [Online]. Available at: http://portal.acm.org/citation.cfm?doid=564585.564601. [Accessed 25/04/2011].

[18] User Datagram Protocol [Online]. Available at: http://tools.ietf.org/html/rfc768. [Accessed 29/04/2011].

[19] C++ Language Tutorial [Online]. Available at: http://www.cplusplus.com/doc/tutorial/. [Accessed 02/05/2011].

[20] C# programming language [Online]. Available at: http://en.wikipedia.org/wiki/C_Sharp_(programming_language). [Accessed 02/05/2011].

[21] Database [Online]. Available at: http://en.wikipedia.org/wiki/Database. [Accessed 06/05/2011].

[22] SQL Server [Online]. Available at: http://en.wikipedia.org/wiki/Microsoft_SQL_Server. [Accessed 06/05/2011].

[23] MySQL [Online]. Available at: http://sv.wikipedia.org/wiki/Mysql. [Accessed 06/05/2011].

[24] Oracle DBMS [Online]. Available at: http://en.wikipedia.org/wiki/Oracle_Database. [Accessed 06/05/2011].

[25] PostgresQL [Online]. Available at: http://sv.wikipedia.org/wiki/PostgreSQL. [Accessed 06/05/2011].

[26] Mnesia [Online]. Available at: http://en.wikipedia.org/wiki/Mnesia. [Accessed 06/05/2011].

[27] Software Foundation Apache [Online]. Available at: http://sv.wikipedia.org/wiki/Apache_Software_Foundation. [Accessed 06/05/2011].

[28] SQL language [Online]. Available at: http://sv.wikipedia.org/wiki/SQL. [Accessed 06/05/2011].

[29] Sales Force Automation [Online]. Available at: http://en.wikipedia.org/wiki/Sales_force_management_system. [Accessed 06/05/2011].

[30] Basho Technologies [Online]. Available at: https://help.basho.com/home. [Accessed 06/05/2011].

[31] Web Framework [Online]. Available at: http://en.wikipedia.org/wiki/Web_application_framework. [Accessed 09/05/2011].

[32] Web Server [Online]. Available at: http://en.wikipedia.org/wiki/Web_server. [Accessed 09/05/2011].

[33] Erlang - ETS [Online]. Available at: http://www.erlang.org/doc/man/ets.html. [Accessed 10/05/2011].

[34] Erlang - Dets [Online]. Available at: http://www.erlang.org/doc/man/dets.html. Accessed 10/5/2011].

[35] Håkan Mattsson, Hans Nilsson and Claes Wikström. *Mnesia - A Distributed Robust DBMS for Telecommunications Applications.*

[36] EUnit (2011) [Online]. Available at: http://www.erlang.org/doc/man/eunit.html. [Accessed 10/05/2011].

[37] Rebar (2011) [Online]. Available at: https://bitbucket.org/basho/rebar/wiki/Home. [Accessed 10/05/2011].

[38] Meck (2011) [Online]. Available at: https://github.com/eproxus/meck. [Accessed 10/05/2011].

[39] Tsung (2011) [Online]. Available at: http://tsung.erlang-projects.org/. [Accessed 10/05/2011].

[40] Redis (2011) [Online]. Available at: http://redis.io. [Accessed 11/05/2011].

[41] RAM memory [Online]. Available at: http://en.wikipedia.org/wiki/Random-access_memory. [Accessed 11/05/2011].

[42] MVC architecture [Online]. Available at: http://en.wikipedia.org/wiki/Model-view-controller. [Accessed 12/05/2011].

[43] Wikipedia - eXtensible Messaging and Presence Protocol [Online]. Available at: http://en.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol. [Accessed 12/05/2011].

[44] Internet Engineering Task Force [Online]. Available at http://www.ietf.org/. [Accessed 12/05/2011].

[45] iJab! XMPP in the cloud [Online]. Available at http://www.ijab.im/.[Accessed 12/05/2011].

[46] iJab - An Ajax Web Jabber Client [Online]. Available at http://opensource.ijab.im/. [Accessed 12/05/2011].

[47] Web Browser [Online]. Available at http://en.wikipedia.org/wiki/Web_browser. [Accessed 13/05/2011].

[48] GUI - Graphical user interface [Online]. Available at http://en.wikipedia.org/wiki/Graphical_user_interface. [Accessed 26/05/2011].

[49] C programming language [Online]. Available at http://sv.wikipedia.org/wiki/The_C_Programming_Language. [Accessed 26/05/2011].

[50] Delphi programming language [Online]. Available at http://en.wikipedia.org/wiki/Embarcadero_Delphi. [Accessed 26/05/2011].

[51] PHP [Online]. Available at http://sv.wikipedia.org/wiki/PHP. [Accessed 26/05/2011].

[52] JSP [Online]. Available at http://en.wikipedia.org/wiki/JavaServer_Pages. [Accessed 26/05/2011].

[53] ASP.NET [Online]. Available at http://sv.wikipedia.org/wiki/ASP.NET. [Accessed 26/05/2011].

[54] Ericsson [Online]. Available at http://www.ericsson.com/. [Accessed 26/05/2011].

[55] Amazon [Online]. Available at http://www.amazon.com/. [Accessed 26/05/2011].

[56] Facebook [Online]. Available at www.facebook.com. [Accessed 26/05/2011].

[57] Yahoo [Online]. Available at www.yahoo.com. [Accessed 26/05/2011].

[58] Inets [Online]. Available at http://www.erlang.org/doc/apps/inets/http_server.html. [Accessed 26/05/2011].

[59] Nginx [Online]. Available at http://en.wikipedia.org/wiki/Nginx. [Accessed 26/05/2011].

[60] Mochiweb [Online]. Available at http://dawsdesign.com/drupal/mochiweb-docs. [Accessed 26/05/2011].

[61] Zotonic [Online]. Available at http://zotonic.com/. [Accessed 26/05/2011].

[62] HTML [Online]. Available at http://sv.wikipedia.org/wiki/HTML. [Accessed 26/05/2011].

[63] CSS [Online]. Available at http://sv.wikipedia.org/wiki/Cascading_Style_Sheets. [Accessed 26/05/2011].

[64] Java script [Online]. Available at http://sv.wikipedia.org/wiki/Javascript. [Accessed 26/05/2011].

[65] Openfire [Online]. Available at http://www.igniterealtime.org/projects/openfire/. [Accessed 26/05/2011].

[66] Pidgin Chat Client [Online]. Available at http://www.pidgin.im/. [Accessed 26/05/2011].

[67] Gajim Chat Client [Online]. Available at http://www.gajim.org/. [Accessed 26/05/2011].

[68] Java programming language [Online]. Available at http://en.wikipedia.org/wiki/Java_(programming_language). [Accessed 26/05/2011].

[69] SNMP [Online]. Available at http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol. [Accessed 26/05/2011].

[70] Python [Online]. Available at http://www.python.org/. [Accessed 26/05/2011].

[71] Ruby [Online]. Available at http://www.ruby-lang.org/en/. [Accessed 26/05/2011].

[72] Lucene syntax [Online]. Available at http://lucene.apache.org/java/2_4_0/queryparsersyntax.html. [Accessed 26/05/2011].

[73] Action Script [Online]. Available at http://en.wikipedia.org/wiki/ActionScript. [Accessed 26/05/2011].

[74] Go programming language [Online]. Available at http://golang.org/. [Accessed 26/05/2011].

[75] Haskel [Online]. Available at http://www.haskell.org/haskellwiki/Haskell. [Accessed 26/05/2011].

[76] Node.js [Online]. Available at http://nodejs.org/. [Accessed 26/05/2011].

[77] Perl [Online]. Available at http://www.perl.org/. [Accessed 26/05/2011].

[78] Erlang Solutions [Online]. Available at http://www.erlang-solutions.com/. [Accessed 26/05/2011].

[79] Wparts [Online]. Available at http://edoc.erlang-web.org/wpart/. [Accessed 26/05/2011].

[80] ErlyDTL [Online]. Available at http://wiki.erlang-web.org/ErlyDTL. [Accessed 26/05/2011].

[81] DJango [Online]. Available at http://wiki.erlang-web.org/ErlyDTL. [Accessed 26/05/2011].

[82] Ajax [Online]. Available at http://sv.wikipedia.org/wiki/AJAX. [Accessed 26/05/2011].

[83] CMS [Online]. Available at http://en.wikipedia.org/wiki/Content_Management_System. [Accessed 26/05/2011].

[84] XML [Online]. Available at http://sv.wikipedia.org/wiki/XML. [Accessed 26/05/2011].

[85] AIM [Online]. Available at http://www.aim.com/. [Accessed 26/05/2011].

[86] ICQ [Online]. Available at http://www.icq.com/en. [Accessed 26/05/2011].

[87] Multi-User Chat Protocol [Online]. Available at http://xmpp.org/extensions/xep-0045.html. [Accessed 26/05/2011].

[88] XMPP/Jabber [Online]. Available at http://sv.wikipedia.org/wiki/XMPP. [Accessed 26/05/2011].

[89] MMO [Online]. Available at http://en.wikipedia.org/wiki/Massively_multiplayer_online_game. [Accessed 26/05/2011].

[90] ODBC [Online]. Available at http://sv.wikipedia.org/wiki/ODBC. [Accessed 26/05/2011].

[91] Ldap [Online]. Available at http://sv.wikipedia.org/wiki/LDAP. [Accessed 26/05/2011].

[92] Pam [Online]. Available at http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules. [Accessed 26/05/2011].

[93] TCP [Online]. Available at http://sv.wikipedia.org/wiki/Transmission_control_protocol. [Accessed 26/05/2011].

A Appendix A: User Manual

A.1 System Setup

The first thing that is needed to be installed in the machine is Erlang[3]. Before installing Erlang make sure that you have all the dependency libraries installed. Some of which are

1. libssl-dev
2. ssh
3. libcurses5-dev
4. git
5. mercurial

Then, the required components like Riak[15], Nitrogen[5] (Webframework + Webserver), Ejabberd(IMS)[13] should be installed. We have provided these components in our product folder as well but it may work or may not work. If it does not work due to some dependency issues, please download the latest source versions of those components and compile them yourself (sorry for this inconvenience but it is out of our hands). After the compilation, just replace the compiled versions of the components with the one that was given. Also make sure to keep the configuration files (DO NOT REPLACE THEM). In case of Nitrogen [5], remember to keep the "site" folder as well which contains the pages, templates, Java scripts [64] and CSS [63]. Also, Riak requires some pre-configuration before we can use it i.e. we need to install the schema. The required schema for the initial setup are given in the schema folder and also a script for running those schemas. Go into that folder and run the command

```
./run_schema.sh
```

After all the components have been installed and configured properly, we are ready to enter the next phase of the setup.

We have kept all our Lobby code in the "lib" folder following the Erlang/OTP standards

- /src contains the source code
- /include contains the hrl files
- /test contains the eunit test cases
- /doc contains edoc files
- /ebin contains .beam files

We have also provided the compilation tool for compiling our modules. We have used "Rebar"[37] for compilation and it can be found in "bin" folder. Now, you can compile all the modules by using the command

```
bin/rebar compile
```

This will compile all the source files in all the modules and generate their corresponding beam files and keep them in the "ebin" folder of that module.

After following all of the above steps, the system is ready to use. To run the system, we have provided a script "start.sh" that will enable the users to run the system all at the same time or one at a time depending upon the users choice.

Before starting the system there is one last thing that needs to be configured. There is the .hosts.erlang file given in the trunk which may be hidden. It should contain the IP addresses of the machines where the components may be started. This will enable the nodes to get connected as soon as they are started. If the system is not started properly check this file first!

To start everything at once use the command,

```
./start.sh all
OR
./start.sh everything
```

This will start everything in the system Nitrogen, Riak and the entire Lobby nodes.

To start all the components individually, use the commands

```
./start.sh nitrogen
./start.sh riak
./start.sh redis
./start.sh db
./start.sh wfw
./start.sh auth
./start.sh admin
./start.sh chat
./start.sh gamenode
```

One can also choose

```
./start.sh lobby
```

This will start the components which belongs to the Lobby only i.e. Wfw, Db, Auth, Nitrogen, Admin and Chat modules.

A.2 Game Node configuration

In this section you will find the information how to configure and administrate the Game Node.

Game Node service consists of the follow parts:

- start_erl.bat
- config.cfg

- start_socket_policy.bat
- .hosts.erlang

Start_erl.bat starts Game Node application. The application starts on the Erlang node. Node name consists of two parts: nod_name@IP. node_name is constant part but IP is machine specific. Open the start_erl.bat and put the correct IP address.

Config.cfg file details and example:

```
{udp_port_default, 1234}.
{instances, [{"GI1", [{path, "D:/Snowbox/World1.exe"}, {port, "7101"},
                      {host_ip, "192.168.1.3"}, {start_regime, batch}]},
             "GI2", [{path, "D:/Snowbox/World2.exe"}, {port, "7102"},
                      {host_ip, "192.168.1.3"}, {start_regime, win}]
               ]}.
{statistics, ["colour", "game_time", "number_of_bolls", "score"]}.
```

Section 'udp_port_default' is used for UDP Game Node-Game communication. If changed then game source code code should be updated as well (see chapter Game Instance Configuration A.3).

Section 'instances' is used by Game Node for starting the Game. First parameter 'GI1' is the unique game id. This id you will get from the Lobby administrator when the game is registered in the Lobby.

Section 'path' there you provide absolute path to the game instance.

Sections 'port' and 'host_ip' are used to provide Player-Game connection. Each game instance should be configured to the unique port. host_ip might be the same for the all instances. Note! Be sure that the port number mentioned in configuration file and those you put in your game instance are similar. Otherwise players will never manage to connect to the game!

Section 'statistics' if you going to store some statistic information of your games in the Lobby database put the fields you want to store in this section. Before sending the statistic message to the Lobby Game Node checks if the fields you send in the message present in the section 'statistic'. If a field from the message is not found in the section then it is deleted from this message. If the section 'statistic' is empty then messages will never be checked. The format of the message see in chapter Game Instance Configuration A.3.

Start_regime - two possible regimes are available: 'win' and 'batch'. When 'win' then game server is started in GUI [48] regime, when batch then without graphics.

start_socket_policy.bat - open this file and put the absolute path to the SocketPolicyServer application. This application is a part of Unity [2].

.hosts.erlang the correct IP address of the Lobby server should be provided there.

Install Erlang on your machine correctly. After installation put the Erlang path to the Windows environment variables.

A.3 Game Instance Configuration

Lobby API consists of two C# [20] scripts: Connect.cs and ServerConnect.cs. Add them to your game scripts. Lobby API is a bunch of function calls. Part of them are compulsory and part is optional. The compulsory calls are:

- Start game - send notification to the Lobby that game is successfully started
- Close game - send notification to the Lobby that game terminates
- User disconnect - send notification to the Lobby that a user log out

The optional call is:

- Game statistics

Currently function calls Start game and User disconnect are not implemented. API might be easily extended.

Start game. Whenever a game instance is started API sends message to the socket. Message format: "{key, value};{key, value};..." where the first two tuples are {transaction, game_start};{instance_id, value}. Other tuples might be optional.

Close game. Whenever a game instance is closed (normally or abnormally) API sends the message to the socket. Message format: "{key, value};{key, value};..." where the first two tuples are {transaction, game_close};{instance_id, value}. Other tuples might be optional. API call is ServerConnect.CloseGame(). It requires one parameter: Dictionary¡string, string¿ dataDict. Dictionary might be empty. Before calling this method always do ServerConnect.Init(String) where the String is the Game id. Game id is the second parameter of array System.Environment.GetCommandLineArgs(). Note! If the close game notification is not sent then this game instance will not be available for starting and joining!

User disconnect. Whenever an user is disconnected from the game API sends the message to the socket. Message format: "{key, value};{key, value};..." where the first three tuples are {transaction, game_quit};{instance_id, value}; {user_name, value}. Other tuples might be optional.

Game statistics. Game might send the information to the Lobby. Message format: "{key, value};{key, value};..." where the first two tuples are {transaction, statistic};{instance_id, value}. Other tuples are optional.
API call is ServerConnect.SendStatistics(Dictionary¡string, string¿ dataDict). All the keys should exist in the Lobby database. Lobby administrator adds them when game is registered in database.

The game instance currently is started with two parameters "regime" and "instance id". The number of parameters might be extended.

All the errors are stored in Game Node log files. File name format: Log.Year.Month.Day.log

A.4 Start Game Node

- put correct path and run start_socket_policy.bat
- configure config.cfg file
- check .hosts.erlang
- start start_erl.bat

A.5 iJab Configuration with Ejabberd

In order to run iJab, an Ejabberd server with http-bind support needs to be up and running. In the Ejabberd Configuration file ejabberd.cfg, you make sure that the mod_http_fileserver module is configured as well as the http_bind module.

```
{5280, ejabberd_http, [
 {request_handlers,
  [
   {["web"], mod_http_fileserver}
  ]},
 captcha,
 http_bind,
 http_poll,
 %%register,
 web_admin
]}
```

In the modules section of ejabberd.cfg, you could specify the doc root and access log for the mod_http_fileserver module, as follows:

```
 {mod_http_fileserver, [
                         {docroot, "$EJABBERDDIR/var/www"},
                         {accesslog, "$EJABBERDDIR/var/log/ejabberd/access.log"}
                        ]},
```

The files of iJab are located at $EJABBERDDIR/var/www. The files are:

- iJab.html
- ijab_config.js.
- ijab_i18n_en.js.
- ijab_i18n_zh.js.

- /ijab

In iJab.html, you can configure the following variables:

```
var host = YOURHOST;
var port = 5222;
var domain = "YOURDOMAIN;
```

The communications occur at port 5222. You may need to change your firewall in order to make it work. If you specify the host to be "localhost", then the URL in the web browser will become: http://locahost:5222/web/iJab.html.