Universiteit Utrecht

Developing a Client-Server Architecture and Minimizing Data Transfer for a Massively Multiplayer Online Game

- Vlad Mihai Alecu -

(ICA - 3570401)

Master of Science Thesis

Utrecht, the Netherlands, October 2012

MSc Thesis

Title:

Developing a client-server architecture and minimizing data transfer for a massively multiplayer online game

Master of Science in Game and Media Technology (GMT).

Utrecht University, Connected Dreams B.V., Amsterdam.

Author:	Vlad Mihai Alecu
Academic Supervisor #1:	Drs. Arno Kamphuis (Utrecht University)
Academic Supervisor #2:	Drs. Hans Philippi (Utrecht University)
Company Supervisor:	Gert-Jan van Rootselaar (Connected Dreams B.V.)

Abstract

This research tries to determine a sound and solid solution to creating a scalable client-server architecture for a massively multiplayer online game that is completely automated. This implies the automated spawning and closing down of servers based on CPU and memory load, data transfers between servers, and database sharding.

As the gaming industry starts to have a clear trend in the last years towards online multiplayer games, the optimization and atomization of a client-server architecture dedicated for such a game represents a very high interest subject in the last period.

The paper will focus on two main parts. The first part mainly discusses the actual client-server architecture. It takes into account how and why the architecture is layered, the functionalities and responsibilities of the servers on each layer, and proposals and comparisons between different ways of server-server communication and overall server system supervision.

In the second part, this paper will extensively analyze which is the best approach to take regarding player placement throughout the servers. While a number of very popular games choose to have a very clear cut system in which each server is allocated to a given geographic region and players from different servers can never interact, the solution proposed here wants to allow any two players from any two servers to interact with each other, keeping the data transfers, however, to a minimum. To reach this goal, this paper proposes a novel solution that consists of keeping track of the "closeness" between players in real time, and predicting future interactions based on this closeness graph.

Finally, the results of applying these discussed techniques on a massively multiplayer online game are shown and analyzed. Based on these results, conclusions are drawn, and the best configuration of the proposed architecture for the given game is presented. In addition, recommendations are given in the end for possible further improvement of the given system in the future.

Table of contents

1. INTRODUCTION

1.1. Context

The past years saw the game industry focus more and more on the capabilities of online gaming. Running a game using a client-server architecture, although being more expensive in terms of server maintenance, represents an almost unbreakable anti-piracy protection for that given game, and also opens up other significant opportunities such as multiplayer interaction, or real-time feedback from players. Recent games such as *World of Warcraft*, *League of Legends*, or *Diablo III* clearly exposed the advantages of using this sort of architecture as they represent, to date, one of the most successful games ever made, adding up together tens of millions of online players.

Because of these increased interests regarding massively multiplayer online gaming, methodologies for development and possible improvements applied on a client-server architecture for such a game have represented a very sought after topic.

1.2. Background

Especially in the last years, supporting thousands, even hundreds of thousands of online players has become a very important issue when it comes to online games, as the efficiency of these backend systems finally plays a very significant role in the game's overall success.

To support hundreds of thousands of players in massively multiplayer online games, a distributed client-server architecture is widely used in which multiple servers are deployed. The way these servers are connected between each other and the way the information is distributed is mostly related to each game in particular, as the structure must usually be moulded on the game's data flow. A common issue, however, is the flexibility of the server system, as the unpredictable movements of the in-game characters or the high concentration of characters in certain worlds can lead to the overload of some servers.

Because of these reasons, constant research is being done on these topics, trying to distribute the load as even as possible between the servers, being it either computational requirements that the machines have to do for each player, bandwidth limitations, or database querying.

1.3. Problem description

To determine an efficient solution for such a server architecture is a difficult task and a number of issues have to be taken into account.

Firstly, the normal data flow of the game has to be analyzed. That means looking at the usual information exchange that has to take place on a normal playing session for a certain user. Using this information, the transferred data can then be divided into content requests (requiring entire files such as images, xml files, etc.), computational requests or database requests.

Secondly, user interaction has to be carefully analyzed. Moving through worlds and changing environments within the game normally means moving information across servers, as interacting player's information normally needs to be stored on the same machine.

In addition, the connections between the players has to be constantly monitored. Based on this information, when a user logs in he must be placed on the machine that has the highest chance not to request any further movement of server data.

Taking these facts into account, a first draft architecture has to be designed, deciding first of all, on what basis the virtual world needs to be divided between servers. The server-side architecture also needs to support a high level of players on each machine, and divides the tasks between the machines as efficiently as possible. Then, player movement and interaction has to be examined and an efficient way to transfer the information between servers has to be implemented, taking into account how this affects the real-time flow of the game on the client side.

Finally, the server architecture has to be made scalable and flexible, starting up new machines and redistributing the data between servers without interrupting the game flow of the online players.

To do this efficiently, real-time monitoring has to be done, and a prediction system has to be set up in order to minimize the amount of data transferred between servers. This plays a very important role in the overall performance of the system as the amount of transfers done is very closely related to the amount of latency that the clients experience when playing the game.

1.4. Main objectives

This research tries to offer a solution for the problem described in the above paragraphs. The goal is to find an efficient (from the point of view of the number of users supported per server and client-side latency), flexible and responsive (have a very small client-side latency) client-server architecture for the given massively multiplayer online game.

1.5. Research questions and research method

The aforementioned problem and the objective that needs to be reached form the following research question:

> ***RQ:*** *What is an effective and flexible client-server architecture design for a massively multiplayer online game?*

And the sub-questions that arise are the following:

> ***SQ1:*** *What is an efficient way of distributing the workload between machines?*
>
> ***SQ2:*** *How can the database be distributed in order to keep a minimal duplication and a maximum reduction of access requests?*
>
> ***SQ3:*** *What is an effective and secure system of monitoring the server side activity (CPU usage, memory usage, bandwidth) in order to make the best decision about spawning a new machine?*
>
> ***SQ4:*** *What real-time analysis can be used in order to determine the "closeness" between players and thus lower the data transfer between servers?*

The research method that will be followed can be seen in Figure 1.1 below.

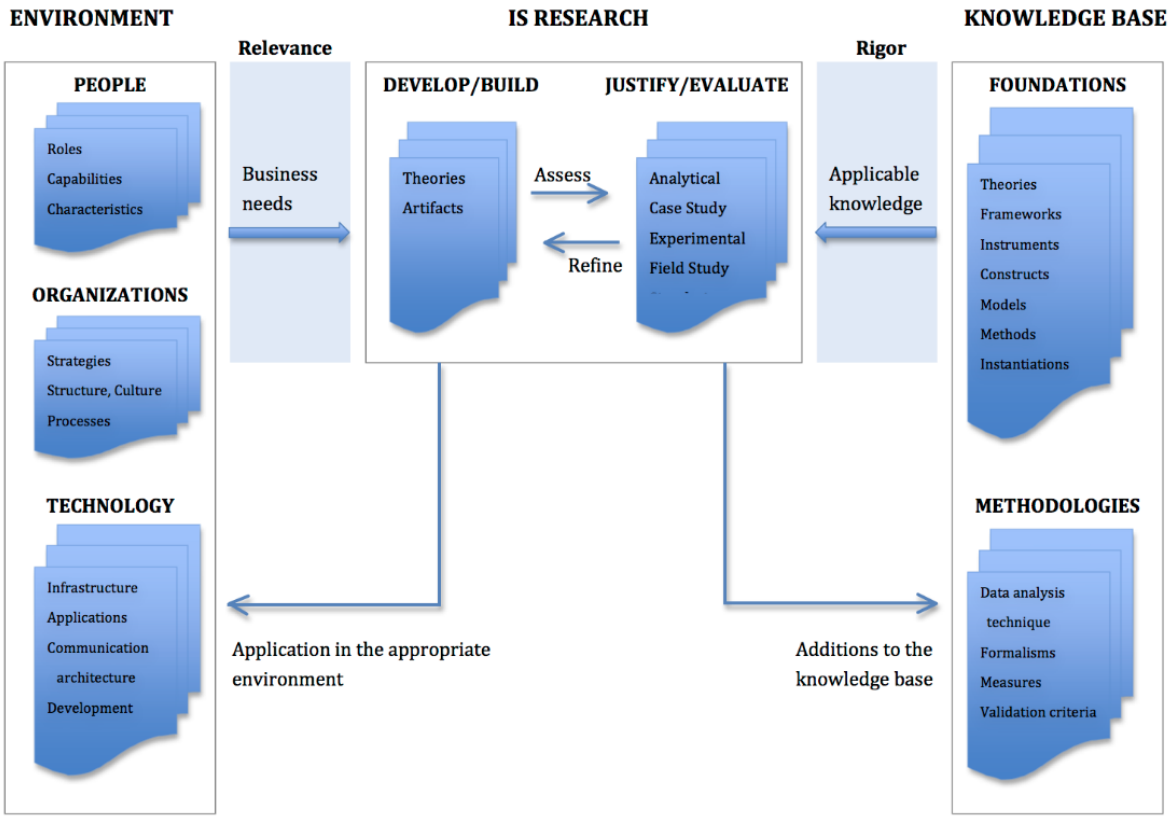


Fig. 1.1: Information Science research framework [14]

As we can see from the left side of the model in Fig. 1.1, the environment, hence the people, organization and the technology define the problem domain from which this study is inspired. The knowledge base on the right side of the model provides the appropriate knowledge from already existing research that can be applied to our case. After some cycles of development, assessment, evaluation and refinement, an artefact is expected to be produced which is an addition to the knowledge base and is applicable to the environment. The social and scientific relevance and the contribution that is expected from this study are explained in the following paragraphs.

The problem of constructing an efficient client-server architecture can have a significant relevance especially from a scientific point of view. This can be applied to a number of applications, not only games, that require more than one server-side machines that need to work concurrently.

In addition, for the game development industry, as the structure of the data flow in this particular case is different than normal multiplayer games that divide the server over their overall map, based on location, finding different approaches can lead to very interesting and useful results.

From a contribution point of view, the academic contributions that this project will bring are the following:

- a state of the art review.
- a description of a server-side architecture that can support thousands of players.
- a database sharding method that allows easy scaling up, as well as down, of the server systems depending on the fluctuation of online players.

Besides these academic contributions, the contribution to the company is a working client-server architecture that will be able to support a high number of online users, and also adapt to the number of online users in real-time, only using the minimum amount of resources it needs.

1.6. Company profile

The company's name is *Connected Dreams* and the game they develop is called *Tailplanet*. *Tailplanet* is a 3D game that can be player using a web browser and does not require any installation. In the game the player is assigned a pet who he then has to raise, take care of and train in order to complete missions and win challenges against other players. The game is a combination between a simulation game (through the fact that you have to keep your pet happy, similar to "The Sims") and a role playing game (because you have to complete missions, get items and gain levels), all of this being done in a multiplayer environment.

Fig. 1.2: A screen shot of the game Tailplanet

Besides the typical elements found in similar games that help the player interact with his in-game character, the application also offers two other main interfaces to the user: a web shop in which the player can buy items to improve his pet or decorate his room, and a chat messenger that allows the user to add other players in his contact list, see when they are online in the game and chat with them inside the game.

The overall application is split in two main parts: client and server. As everything was developed from the ground up and the game wants to be completely independent from any type of programs or plug-ins, the server side was developed using PHP and MySQL and the client side was developed using JavaScript.

All the dynamic objects in the game have state machines that define their behaviour. For any updates that are done on the client (animation information, object creation, object deletion, trigger information, etc.), a request is sent to the server, the server processes it and returns a response which can be either a static response that does not require any computation, or a dynamic response.

1.7. Structure of the thesis

This thesis consists of five chapters and some appendices. The contents of these chapters (except the first one which is the introduction) are briefly described here:

Chapter 2

This chapter describes the general structure of a client-server architecture. Different approaches are explained, with the advantages and disadvantages of each of them. Finally, the data flow of the game on which this research has revolved around is analyzed. Based on the

conclusions drawn from this step, a general structure for the client-server architecture that would work best in this particular case is described.

Chapter 3

This chapter focuses on the ways of communication between the different layers of the client-server structure. It describes the tasks and responsibilities of each type of machine, different approaches to do a real-time monitoring of the overall architecture, and the limitations and bottlenecks of this system.

Chapter 4

Here, the main focus is represented by the description of a real-time prediction system, that allows the servers to know the likelihood of two players interacting with one another. The concept of a *closeness graph* that is updated in real-time is explained, and the definition and calculation of the closeness between two players is described. Finally, different case scenarios are elaborated, showing how the decision process works, based on the values in the closeness graph.

Chapter 5

This last chapter contains the results of implementing the client-server system for the given massively multiplayer online game, but also the possibility of applying this architecture for any general MMOG is analyzed. In the end, conclusions are drawn and recommendations are made in terms of other small possible improvements of the overall architecture and interaction between the servers.

2. CLIENT-SERVER ARCHITECTURE DESIGNS

2.1. Introduction

The basic concept behind a client-server architecture is to distribute tasks between the providers of a resource (which are the servers), and the ones who request that resource (the clients). Thus, the clients are the ones that determine when a session starts and ends, as they are the ones who send out requests to the servers. The later normally operate in a latent state, waiting for new packages from clients, computing them, and sending out responses.

If the two main parts of this structure are separated and analyzed, the client side is very straight forward. A client represents an application or device that is not capable of running on its own a stand-alone program, but instead needs to communicate with another entity via a network. With regards to online multiplayer games, the client is the application that the player has install on his personal computer or console. However, in order for that game to run properly, that application needs to establish a connection with a server.

The clients are not represented only by applications that the user explicitly installs on his machine. A client can also be a simple web page. Simple online games that are player through a browser also represent a client, although no explicit installation is required. This scenario can be seen as a two layered structure, as the web browser itself acts as a client at the top level, requesting a URL of the page on which the online game is. When receiving its response, the web server practically received the client-side application of that game. Once the user interacts with that page, the client-side of the game starts sending requests to the game server(s), which then send feedback to the client application. This is the type of client that will be mostly targeted, as the given online game on which this research was developed on is a browser based game.

While the client side is very unambiguous and is always represented by a single entity, the server side can be very complex, and this represents the part on which this chapter will mostly focus on.

Firstly, there are numerous types of servers: web servers, FTP servers, database servers, file servers, application servers, computational servers, mail servers, etc. The ones that are of interest with regards to online games are:

- file servers: which receive only requests for certain files, no computation needed.
- computational servers: which have no analyze and compute the contents of the received packages in order to obtain a response.

- database servers: which store player and game-play information.
- application servers: which usually have a global view over the whole structure and handle issues such as load balancing or data management.

The way these types of servers can be set up and communicate with each other is a topic that will be treated thoroughly later in the paper.

The rest of this second chapter will examine different ways of structuring the server architecture. Afterwards, the data flow of the given game will be presented and, based on that, a general structure for the client-server architecture that would best fit this particular game is described.

2.2. Server-side structuring

This part of the chapter will look at different approaches of dividing the servers based on game structure and requirements, load balancing issues, and also taking into account overloading scenarios and the scalability capabilities of the system.

2.2.1. Server regionalization

When thinking of ways to divide a game's virtual world for the purposes of modelling player interactions, but also providing manageable consistency, the choice normally stands between two very opposite approaches (as also described in [9] and [15]):

- Geographical: the virtual world is divided into regions from the moment the game is initialized, each region being assigned to a given server.
- Behavioural: the virtual world is sub-divided in a way which reflects the interactions between players.

The geographical approach is very often used when the virtual world presents very clear spatial borders, that do not allow players outside a given space to interact with them in any way. For example, if a player is in a room, he cannot, at that moment, interact with anyone else except the people in that room.

The behavioural approach is not determined by static spatial constraints such as walls, but by the area of influence that each player has and the ability of other players to express interest. For example, if in a game the player can choose to either be in an aircraft or a tank, a player manoeuvring the aircraft will have a clearly higher area of influence that the one who chose a tank just by the amount of space that he can cover.

A middle-way approach between the two was tried by MASSIVE Systems in the 90's (as detailed in [13]). It was called the aura/nimbus approach and the main idea was to model the influence for each player in particular. The aura was the area in which the player can express

influence, while the nimbus was the area in which others can express interest from that player. So, while focusing on the behaviour of players, this method still assumes that only geographically close players are allowed to interact with each other. The main limitation of this method however, is its scalability, which is strongly limited by the amount of extra processing power needed to determine each player's aura/nimbus.

2.2.2. Server data distribution

The way the data is spread across servers is a topic that mostly depends on the basis on which the game information can be divided. To achieve load balancing (an efficient distribution of data), two main approaches stand out (as mentioned in [9]):

- Player-based: players are allocated to a server at the moment that he enters the game.
- Interaction-based: the servers allocate their resources based on the interaction patterns between players.

The player-based approach is very unambiguous. The player is allocated to a certain server at the start of the session. Using a NAT (network address translator), the connection between the client and the server it was allocated to is remembered, and all the requests of that client are sent to the given server for the lifetime of that session.

The interaction-based approach offers the player a higher degree of freedom. The player can communicate with players from other servers and players are able to move from server to server during a session. The decisions of moving data from one server to another or of just sending messages from one server to another are taken based on processing load of each server and the latency that each operation produces on the client side.

2.2.3. Server overload solutions

The most difficult problem to solve with regards to a client-server structure is known as overloading or crowding. Overloading take place when a server exceeds its processing capabilities due to a very high number of concurrent requests. This can lead to increased latency, package loss or the server can even crash.

One solution for this problem is to set up the servers as a grid, allowing each one to communicate with its neighbours (described in [1], [6] and [10]). Once a server is close to its maximum load, it communicates with its neighbours and transfers data to one of them that has a lower load. If after this transfer, the neighbours also reach their maximum limit, the propagation continues further, until all the machines are stable. The limitations of this method are that each server has to constantly know the state of its neighbours, making each machine a miniature master over its neighbouring region. This results in each machine losing a part of its processing capabilities that could be otherwise used to handle client requests. While for each machine this loss is not that significant, over the entire system this may not be the case. In addition, because of the increased bandwidth usage of each server communicating with its neighbours, the latency to the client layer may also be affected.

Another approach is to have a single master over all the other servers that act as slaves (discussed in [4]). The master does not handle any incoming requests from the clients, its only job being to oversee the state of the other machines, and handle cases in which data needs to be transferred from one machine to another. One advantage of this method compared to the previous one is that all the slave servers are considered to be equally connected. So if one machine is overloaded, a single data transfer to another server will solve the issue and no propagation through neighbours is necessary. Furthermore, as a single machine handles all these responsibilities, the loss of processing capabilities to manage the overall architecture always remains static. Thus, while the first approach may give better results on smaller scales in terms of processing power loss, this approach is superior on a larger scale. The disadvantage however, is the fact that the master acts as a bottleneck. As it is the only managing entity over the server-side system, there must be well designed backup systems that treat scenarios in which the master server crashes or experiences other fatal problems.

2.3. Game data flow

As mentioned in chapter 1, the game that this research is based on is a browser based massively multiplayer online game called *Tailplanet*.

The game starts with your main character in his own house. Besides the house, the player also has a gym, an arena and a number of shops that he can visit and that are not a global instance, but different for every account. Interaction with other players can be done either by visiting some else's house, someone visiting you, or inviting someone to the arena for duels. The updates between players are done by sending motion and position information across at the start of each animation, which is usually every second.

The starting point of this project is represented by the server side consisting of a single machine that handles both the HTTP requests sent by the clients and the database processing. The HTTP requests can be split into two categories: static requests that just require the contents of a file (JSON files or images) and do not require any processing or connection to the database, and dynamic requests that do have computational expenses and do execute SQL queries.

An important aspect of this game's structure is given by the way the dynamic client requests are categorized:

- Webshop: requests that are made when a player buys a new item in the game.
- Chat: the game contains a chat messenger through which the players cat communicate with one another. When a message is send from one player to the other, it is initially passed to the server, which then notifies the user at the receiving end that he has to update his chat data, thus fetching the new message. Here there are also included

requests to add a new contact, and accepting or declining an invitation from another player.

- User-specific: requests that are related to the user information such as password recovery, user account data modification, or buying premium items in the game (using real money).
- Game-play related: these represent more than 90% of the actual requests. For every animation that needs to run, any pose or position information of a persistent object that has been modified, or for every persistent variable of a state machine that was been changed, the client sends a request to the server. Without any latency, these cumulate in total to around 15 requests/responses per second between client and server.

Based on these types of requests, an even more clear classification immediately comes to mind: one based on frequency. Thus, only two categories remain: game-play related and the second one can be called user-driven, as all the others are sent by the client only as a result of a specific action of the player. This is very important when databases are also taken into account, as big improvements can be obtained by treating the two separately. Frequently modified, volatile data that does not need to be persistent can be cached in memory, using a memory engine, increasing the response time to the client, while the persistent tables that keep track of the user information can still use normal database engines such as InnoDB or MyISAM.

2.4. Proposed client-server architecture

Based on the information given in this chapter, the different designs of the server-side structure and the data flow of the given game, a multi-layered client-server architecture is presented, giving arguments for each choice.

2.4.1. Server regionalization choice

On the subject of server regionalization, the most suitable approach is always strongly dependent on the game-play structure and how the game designers want the player interaction to take place.

In the case of *Tailplanet*, a very attractive feature of this game is given by the multiplayer interaction. The game should allow any two players to interact with one another, without burdening the player with decisions regarding choosing a certain server either at login or at any other time during game-play. Thus, all decision making processes should be done on the server side of the application. This results in a server structure that is not focused on separating the player groups geographically, but more focused on facilitating a good interaction between players. Thus, the behavioural approach should be the one that best fits the characteristics of the given game. As mentioned in chapter 2.2.1, this approach does not

restrict the player in any way and does not oblige the game to have any spatial limitations inside its virtual world. Furthermore, from the client layer, the player should not even be conscious of the presence of multiple worlds that are in any way separated from one another. From the player perspective, the wanted outcome is for him to be able to have unrestricted interaction with anyone in the virtual world of the game.

2.4.2. Data distribution choice

With regards to how the data should be distributed across servers, this is usually closely related to the previous topic of regionalization.

Geographical regionalization goes hand in hand with player-based data distribution, as players do not move across servers during the lifetime of one session. Thus, the most simple approach in this case is to have servers defined for a given region, and each time a player logs in a server in chosen at that time based on that player's geographical location, and the given player's interactions are limited to that server's world and the players that are on it.

Similarly, the interaction-based approach of data distribution is more closely related to the behavioural regionalization, as both focus on the actions of the player, trying to facilitate the player with a higher degree of freedom within the game. This approach however implies a higher degree of logistics and server-side computations. A system of real-time data transfer between servers has to be implemented, as this approach allows players from different servers to interact, this implying that either information needs to be duplicated or moved from one server to another. An analysis on what type of information needs to be moved across in this particular case, and what would be the best way to facilitate these requirements will represent the next main topic.

2.4.3. Client request classification

Firstly, based on the information given in chapter 2.3 regarding the game flow, one of the first observations that can be drawn is that the server side processes can divided into two main parts:

- Computational servers: that directly handle requests from clients, doing all the necessary computations and bearing most of the bandwidth usage.
- Database servers: that store all the game related information and that are queried by the computational servers when needed.

Taking this main division as a starting point, each one of the two can be further analyzed.

The computational servers as mentioned above, are the first layer in handling requests from the clients. However, these requests can be separated into two general categories:

- Static requests: are requests that only demand static content from the server. This can be an image or a JSON file that contains details about a given object (such as

bounding box dimensions, an animation graph that shows what animations can be played after a given one, etc.).

- Dynamic requests: are requests that require any sort of computation on the server side, also requiring a connection to the database for entries to either be fetched, updated or deleted.

This division is important especially performance-wise as special dedicated programs can be used to handle static content being fetched (Nginx, G-WAN, etc.). Also, each type of machine can be designed based on its main function, as the static requests server only needs a very good bandwidth connection, with no need of very high CPU power, while the dynamic request servers need to have a higher computational capability and also need to be connected to the database server layer.

2.4.4. Database layer division

Moving on to the database servers, the database contents o the given game has to be first looked over before an optimal division can be made. The table presented in Annex no.1 contains all the data that is stored in the game database, and for each entry it is specified whether it is part of the four categories given in chapter 2.3: chat, webshop, user-specific, or game-play related. In addition, the access frequency is specified.

Besides this general classification of the database tables, an even more important aspect is how these tables are linked through primary key fields. If you look at Annexes no.2 and 3 you can see these connections.

Taking this data into account, it can be observed that there is a clear separation between the main game-play related tables (seen in Fig. 2.1) and the others. Also, in Annex no.2 it can be seen that multiple user related operations can be easily distinguished, having only a single table (the *user_names* table) as a common connection. The webshop related connections were now shown here as the only tables that are connected in such a way are the ones that are queried only once at login ,and as they are independent from tables that are not webshop related, they do not have any significant effect on how this database should be optimally divided. All the entries that are seen in the table at Annex no.1 but not seen in these two figures above have only simple queries executed on them that do not require data from any other table in any kind of game-play scenario.

Based on the database description done up to this point and the observations stated in section 2.3, a good approach in dividing the database server layer is to separate it into two main layers:

- The Volatile DB layer: this layer contains all the frequently accessed tables that contain unstable information which is very often changed or deleted. The information in this layer is also only session related, so in case of an unexpected error or crash, the integrity of the player accounts will not be affected. The worst case situation is that the final few minutes of game-play would not be properly recorded, so the last actions that

the player has made have to be repeated. Because of this, these machine can keep all their information on memory, making for a very fast query response time, which is essential for this very frequently queried layer.

- The Stable DB layer: this layer contains all the sparsely accessed information, including the user account information, webshop and chat information. As this information is very important with regards to account safety and consistency, all these machines are ACID-compliant (atomicity, consistency, isolation, durability), making all transactions safe even in case of a system crash.

Both these layers will contain multiple machines, and the dynamic request servers will be able to access both these layers.

2.4.5. Handling overloading scenarios

As the main layers of the proposed client-server architecture have been presented, a solution for overloading scenarios can now be discussed. Taking the options presented in section 2.2.3, the most plausible solution for an architecture mostly focused on player interaction is the with an overall master server. This solution also fits with the idea of having a *closeness graph* that is constantly updated, keeping track of the interactions between players. In order to have this structure, an overall server is needed, that can have a global view of the system. This would not be possible in a shared-responsibility system, in which each server would share responsibility for a small part of the overall structure. However, as mentioned in section 2.2.3, this approach presents an important drawback: the master server acts as a bottleneck for the whole system, disrupting the whole structure if an unexpected error occurs with it or because of very high demands from the other machines.

2.4.6. Architecture diagram

At this stage, the overall architecture that would result from the conclusions drawn so far would have the structure presented in Figure 2.1.

The whole structure of the client-server architecture can be described as a four tier application.

The top level consists of the clients, who are the ones that start the requests. The client layer in Figure 2.1. represents the actual users, so it is not part of the actual application. The web servers in the figure (the program that runs in the client's browser) represent the actual top layer of the application.

The second layer consists of the server side machines that only do the main calculations. These can either be dynamic request servers that do require processing power and are linked with the DB layers, or static request servers that just return file contents to the client.

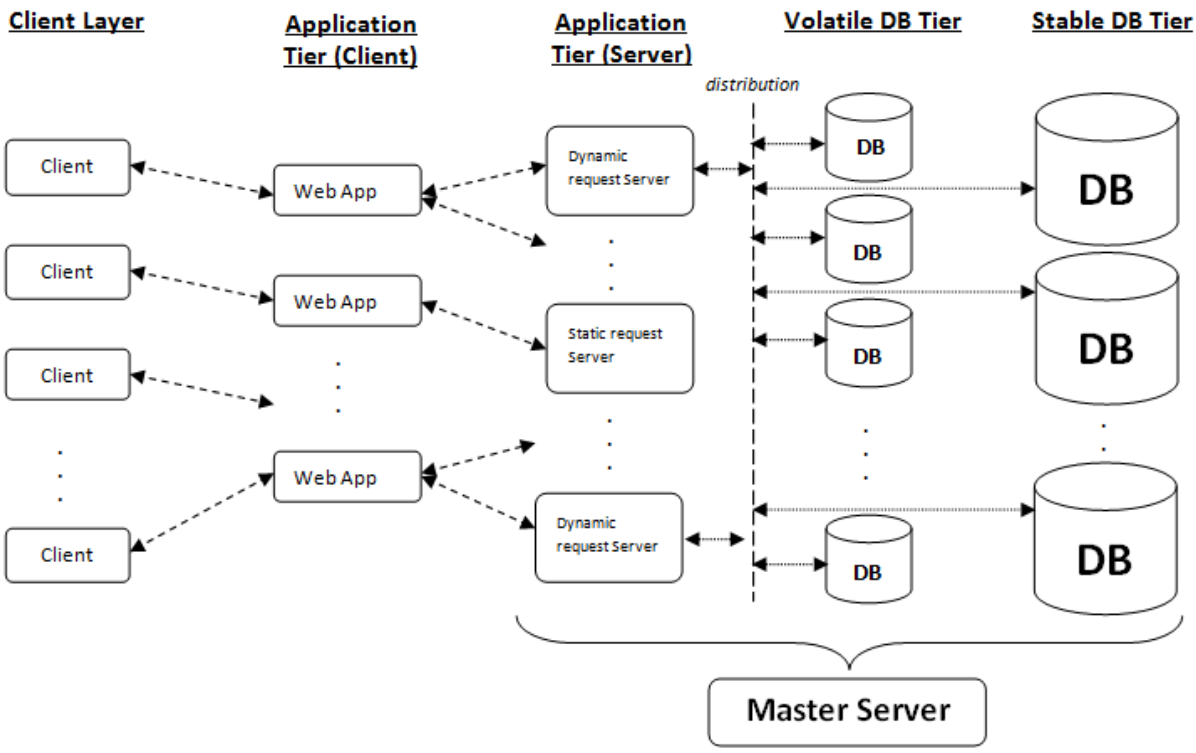


Fig. 2.1: the initial structure of the proposed client-server architecture

The third and fourth tiers consist only of database servers. The third tier has a number of database servers that all contain unstable information (tables that are constantly updated or from which entries are deleted). The fourth and last level is composed of database servers that stores all the stable information. Because the access for the information stored on the forth layer is a lot sparser that for the information on the third layer, the MySQL maximum workload for a volatile DB server will be reached a lot faster than for a stable DB server. Because of this, a fourth tier machine can support a lot more data to be stored on it compared to a third tier machine, so the number of volatile and stable DB servers will mostly likely be unequal at any given time.

The Master Server, as shown in the figure will communicate and oversee the whole server structure, including the client application layer as it would also handle responsibility when it comes to new users accessing the web page of the game. This however does represent a problem as it adds a high risk element to one of the most important elements of the architecture: a peak in client requests can stop or significantly slow down the master server activity. The biggest disadvantage of this is that it is not controllable from inside the system. To solve this, a separation has to be made between the master server and the client application layer. As the only communication between the two taken place at the initial web page access, this responsibility is removed from the master server, adding a new type of machine that specifically handles only this responsibility. Thus the final structure of the client-server architecture will be:

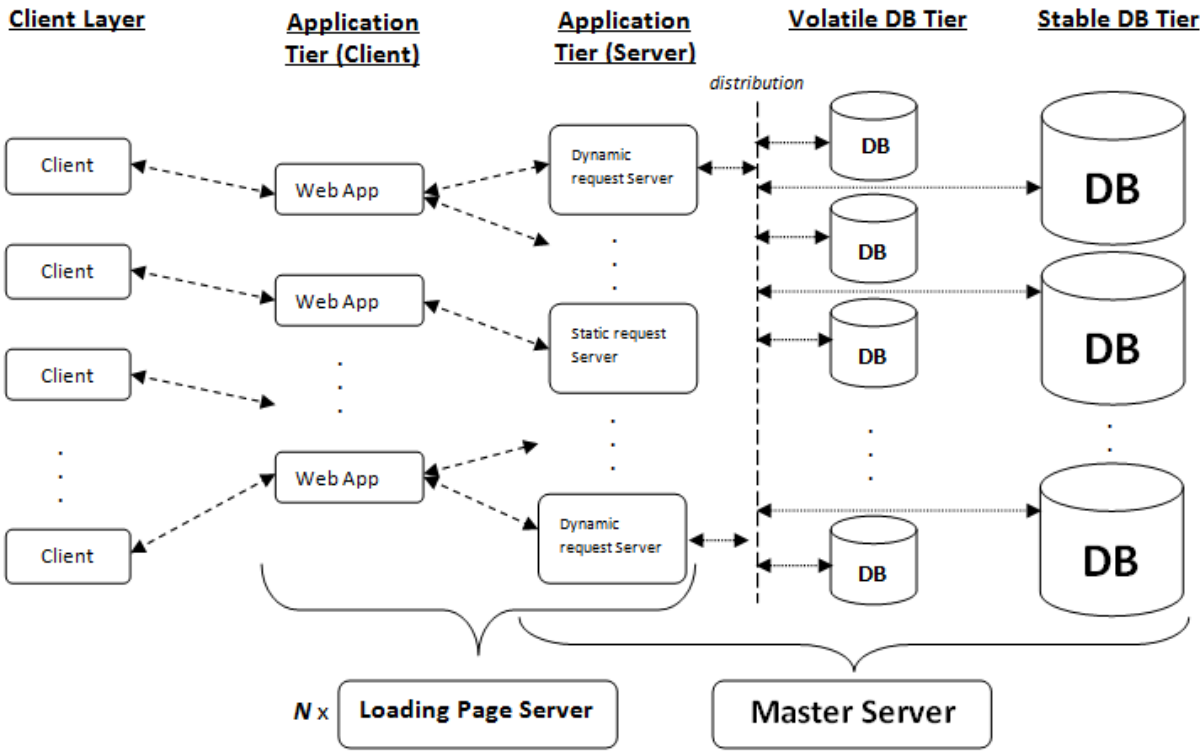


Fig. 2.2: the final overall structure of the proposed client-server architecture

The only modification to the previous design is the addition of the Loading Page Servers. These machines are the ones that actually have the address of the main website, being the first ones that handle requests from a client at the start of a session. Their only responsibility is to establish an initial link between the first and second layer machines at the start of a session. With this approach, the master server is not exposed to any outside threats and the machines that are exposed do not threaten the overall structure in case of an unexpected error.

Thus, the Master Server in this final design oversees only the server-side architecture composed of layers 2, 3 and 4. How these communications will take place will be detailed in the following chapter. Also, as mentioned before, the master server will also be responsible with keeping track of the interactions between players by updating in real-time a *closeness graph*. This data will then be used by the machine to take decisions regarding on which machine player information needs to be stored at login, or how the data can be moved between servers at any point in time.

3. SERVER SIDE INNER-COMMUNICATION

3.1. Introduction

This chapter will focus on the means by which each type of machine described in the structure that was presented in the previous chapter communicates with the other server-side machines.

Each type of server plays certain roles during a playing session of a user. For each type of server, its responsibilities inside the given structure will be first stated. Afterwards, for each action that is presented, details will be given on how that process takes place.

As each layer contains a certain type of machine, the communication will be viewed as in-between different layers of the server-side architecture. A detailed view of how the three server-side layers interact with each other will be presented in the first part of this chapter.

A special attention will be given to the master server as it is the one which has the biggest management role inside the given architecture. Its roles include the following, each one being discussed thoroughly later in the chapter:

- Load balancing: keep track in real time of the load on each machine and act accordingly when a server reaches its maximum load threshold.
- Login administration: although the loading page server is the one that directs the clients to the second layer machines, it is not the one that decides which players connect to which second layer server; the master server specifies to the loading page server to which machine it should direct the new players; further in the login process, the master also allocates the player to the appropriate DB servers, the decision being made based on the CPU load of each machine, and the closeness values of that user to the other online users on the servers.
- Closeness graph updating: the master is in charge of the player closeness graph; it updates the values constantly, and it is also the one that uses those values to make decisions.
- Data management across DB servers: either at login when the decision is made that players need to be moved from a server to another, or when to players situated on different servers interact, the master server is in charge of all the logistics; based on the closeness values and the probabilities of those players interacting with other on that server or any other server, a decision is taken with regards to who's data is moved and on machine it is moved to.

In addition, its weaknesses and limitations will also be presented, and solutions will be proposed.

Furthermore, the scalability capabilities of the proposed architecture will be analyzed. Scenarios will be discussed in which either servers from each layer need to be shut down, or new machines need to be started up. In each case it will be specified what actions need to be taken by each machine in part.

3.2. First layer communication

As seen the Figure 3.1, the first layer, or client application layer, handles the user input and transmits and receives packages to the servers.

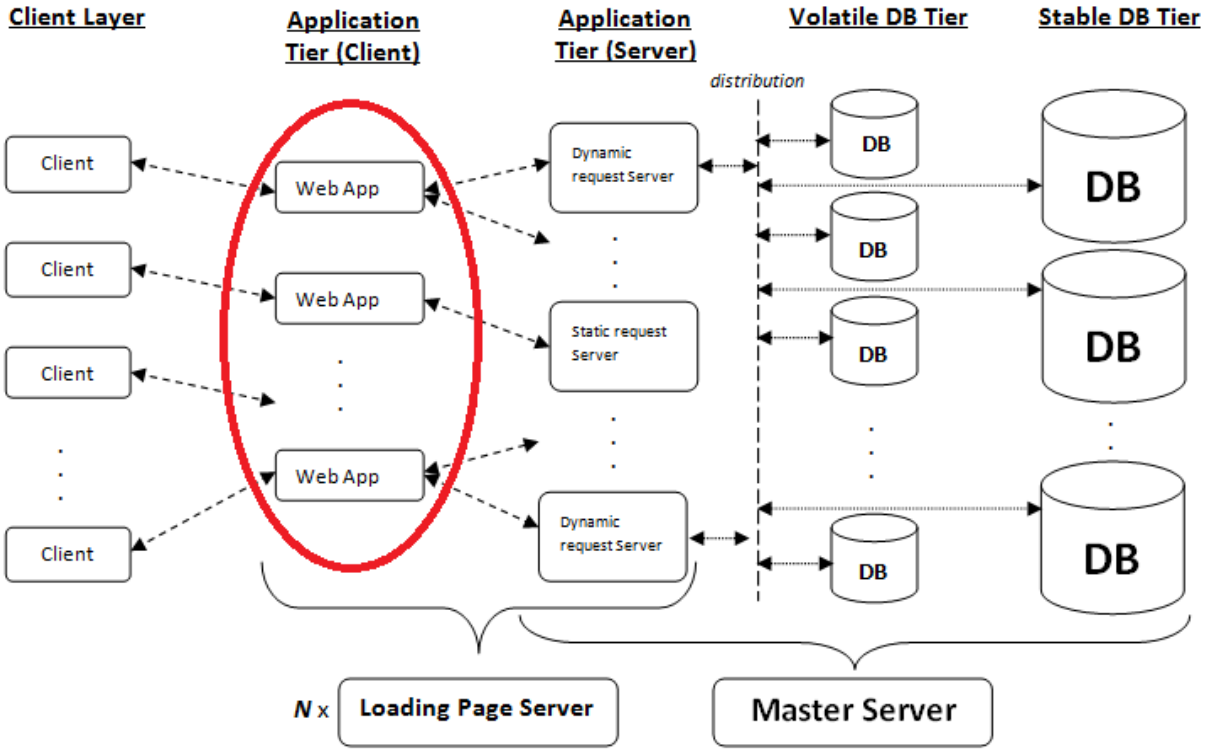


Fig. 3.1: the overall 4 tier client-server architecture with the first layer being highlighted

The first type of interaction that this layer handles is with the users. The user input is taken by the client-side application which then decides, based on that information, if any server-side acknowledgement or information is needed, and if so, the appropriate package is sent across.

The interaction between the first tier and the server application tier (second layer), is dependent on the type of request that the client does.

If static content is needed, an HTTP GET request is sent to a static request server, and the first layer then awaits the requested file as a response.

In the case of an action that requires server-side processing (animation being updated, persistent data being changed, chat or webshop interaction, etc.), the web server sends out an HTTP POST request to a dynamic request server. That sent request contains a JSON string that is decoded on the receiving side, the necessary operations are executed, and a response is then given back to the client layer, if required. A detailed explanation of how these requests are handled by the dynamic request servers and what types of requests can be received will be given in the following section.

There is also one scenario when the client layer communicates directly with the loading page server. This is the very first server-side interaction that the client application has at the beginning of each session, when the player actually accesses the web domain of the game. The flow of events can be seen in the sequence diagram below:

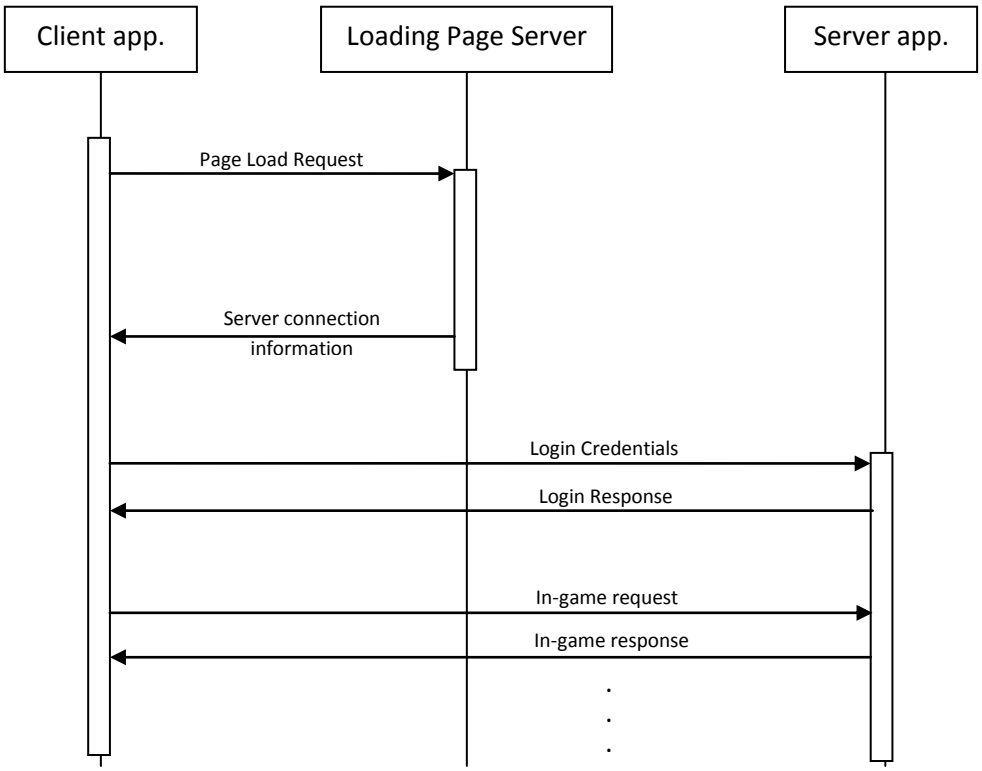


Fig. 3.2: a sequence diagram of the login communication that involves the client application layer

At login, the client application first sends a request to the loading page server, letting in know that it wants to start a new game session. The loading page server sends back to the client the connection information for both a dynamic and static request server. This information is passed on to the loading page server by the master server, the later being the one that makes the actual choice. How this decision is made will be explained later in the chapter when the master server interaction will be discussed. Once this is done, the communication can flow normally between the first and second layer. When pressing the login button, the first layer sends its login credentials that the user inputted to given dynamic request server on the second

layer. That server then returns a login response, and if that response is positive, the in-game communication can then take place between the client application and the two second layer servers. If the credentials are wrong, then the process is repeated from the very start. This is because the data on the base of which the master server chooses the second layer computational servers may have changed from the previous attempt.

3.3. Second layer communication

The second layer of the client server architecture, as mentioned a number of times before, contains two types of machines: static request servers and dynamic request servers. Both are highlighted in the figure below. This section will discuss both of them, one by one, with a more emphasized focus on the dynamic request server, as it also communicates with the database server layers.

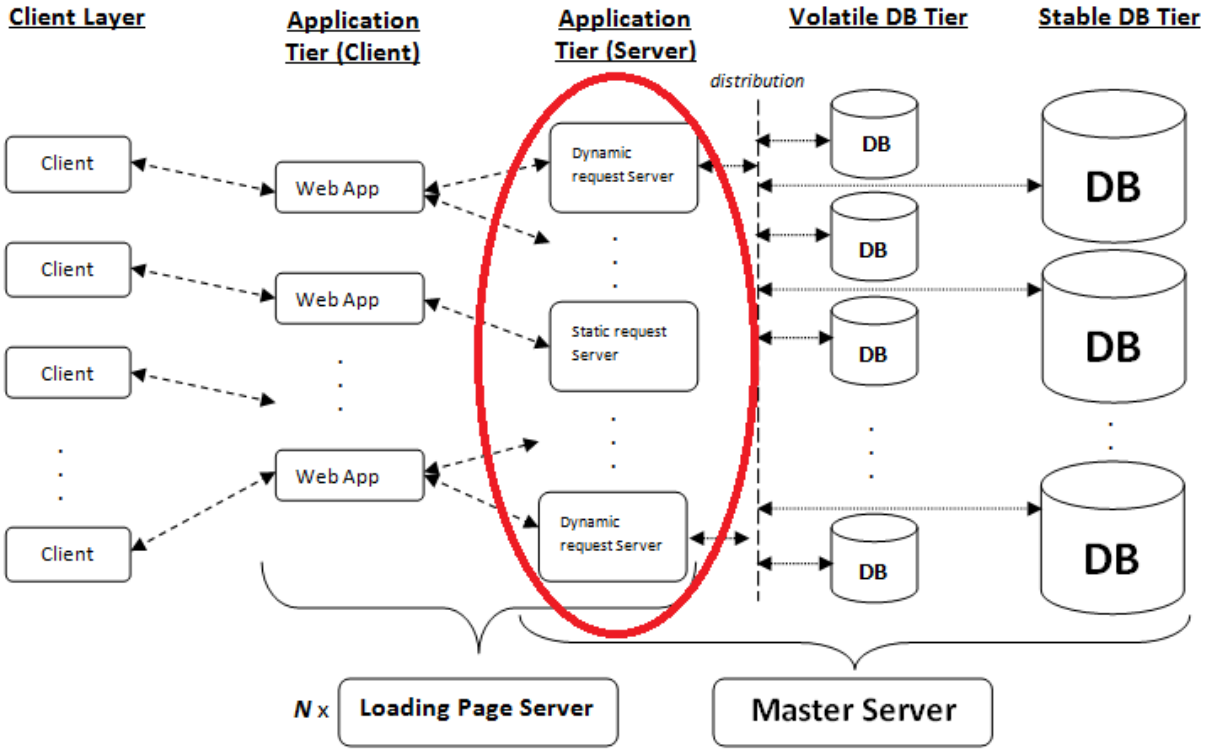


Fig. 3.3: the overall 4 tier client-server architecture with the second layer being highlighted

3.3.1. Static request server communication

As mentioned above, the static request server does not communicate with the database server layers. It does communicate with the client application layer, responding to HTTP GET requests. When receiving this type of request, the server responds by sending the contents of

the requested file back to the client. To facilitate this type of interaction, the server uses dedicated applications that keep the static data cached in order to reduce the latency.

The static request server also exchanges information with the master server, sending to it information regarding the state of the machine, most importantly regarding CPU usage. This is done every time a significant change to the state of the machine take place, and the master server uses this information to choose an appropriate machine for users stating a new session.

3.3.2. Dynamic request server communication

The dynamic request server acts as one of the footstones of the proposed client-server architecture. It is the one that handles the largest amount of the client requests and it also interacts with all the other architecture members: both database server layers and the master server.

The communication with the first layer is very similar to the static request server. The client application sends a JSON string through a HTTP POST request to the server. The server then parses the string, computes the given information, connects to the database servers if necessary, and then responds to the client.

The fields that can be found in the JSON string, along with an explanation of their role can be found in Annex no.4.

When receiving these packages, the server has two main programs running:

- The first program parses the received JSON string, execute the appropriate commands, and then returns a JSON packet as a result, if necessary.
- The second program handles all the multiplayer related activities and runs continuously in the background. The first program checks in different scenarios if the given user's actions affect other players. If that is the case, the player adds database entries that informs the affected users that changes occurred. At that moment, this multiplayer activity program that runs for those players fetches the changes and sends them across to the client layer. Finally, those players see the actions of the given user, whether it is an animation if a character or received chat messages or invitations.

Moving on to the dynamic request server's interaction with the database server layers, this takes place based on what machines the player information is stored on. This information is kept on a local table on the dynamic request server. This table contains the connection information for one volatile database server and one stable database server for each player that is handled by that second tier server. Using this table, the server connects to one of the two servers and queries them for the required information. How the information of this table is set is explained in the sequence diagram above.

At login, the dynamic request server starts its activity when the first layer application sends across the login credentials. At that time, the second layer server starts communicating with the master server. It sends a request for it to determine the appropriate database servers that

the dynamic server needs to access in order to fetch the client's data. The master server then has two important operations to execute. Firstly, it has to choose the best servers on which to place the user information. The details of this process are thoroughly explained later on in the chapter when the master server interaction is explained. Secondly, after this decision is made, the user information is placed on those two chosen database servers. Afterwards, the master server informs the dynamic request server of which machines it needs to connect to in order to get the player related information. The second layer server stores that information on its local table and finally returns a login response to the client layer.

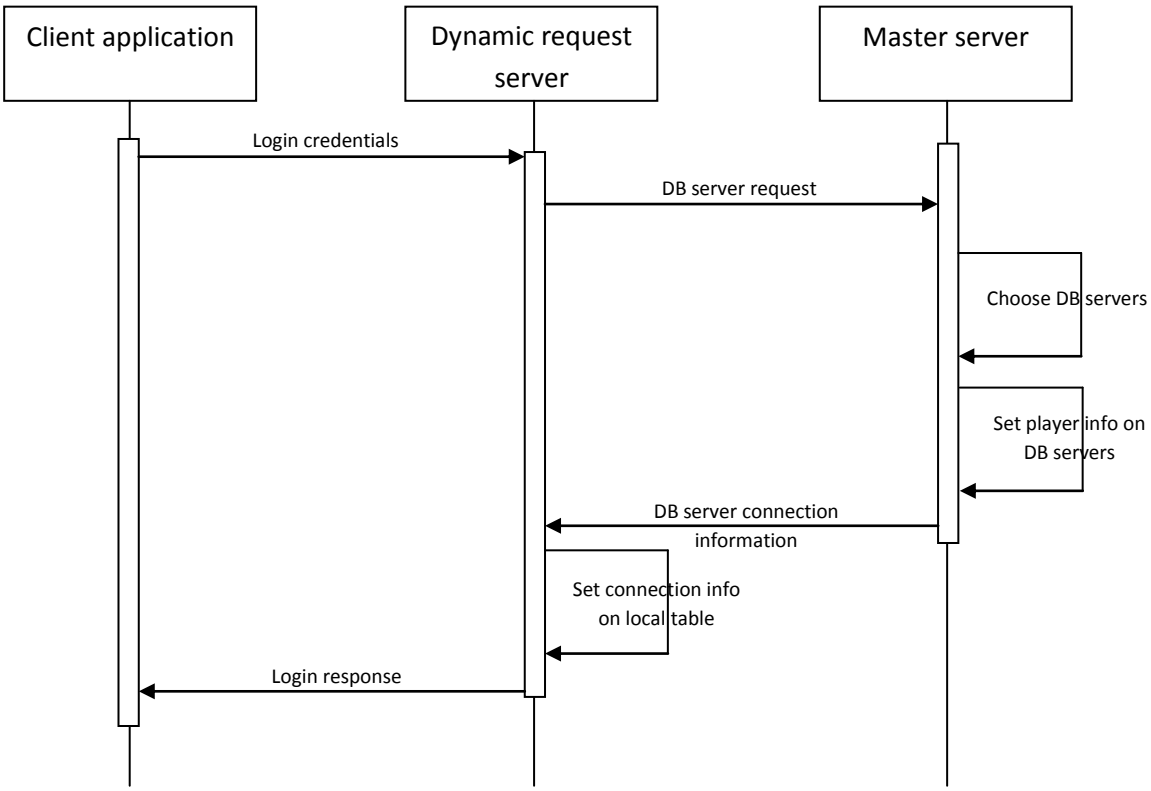


Fig. 3.4: a sequence diagram of the login communication that involves the dynamic request server

Moving on to the dynamic request server's interaction with the database server layers, this takes place based on what machines the player information is stored on. This information is kept on a local table on the dynamic request server. This table contains the connection information for one volatile database server and one stable database server for each player that is handled by that second tier server. Using this table, the server connects to one of the two servers and queries them for the required information. How the information of this table is set is explained in the sequence diagram above.

At login, the dynamic request server starts its activity when the first layer application sends across the login credentials. At that time, the second layer server starts communicating with the master server. It sends a request for it to determine the appropriate database servers that the dynamic server needs to access in order to fetch the client's data. The master server then

has two important operations to execute. Firstly, it has to choose the best servers on which to place the user information. The details of this process are thoroughly explained later on in the chapter when the master server interaction is explained. Secondly, after this decision is made, the user information is placed on those two chosen database servers. Afterwards, the master server informs the dynamic request server of which machines it needs to connect to in order to get the player related information. The second layer server stores that information on its local table and finally returns a login response to the client layer.

Once the lines of communication are set, the second layer servers connect to the appropriate database server by taking into account the data affected by the queries (volatile or stable) and the information in the local lookup table.

It is important to mention that the lookup table information is not unchangeable over the length of a session. There are scenarios when players need to be moved across database servers during a session. When that happens, a similar interaction as the one presented in figure 3.4. take place between the master and the second layer server. The details of the sequence of events that take place in this case, and also the enumeration of scenarios that produce such an event will be given in the following sections of this chapter.

3.4. Volatile database server communication

The volatile database servers represent the third layer on the client-server architecture that this paper proposes. This can be seen in Figure 3.5. below. These machines store the game-play information for the online players, responding to queries that are being executed by the dynamic request servers. However, communication also takes place between this layer and the stable DB layer, and also between it and the master server.

Regarding the communication with the second layer, this takes place when persistent game-play related data needs to be modified, saved, or fetched from the server, by the client. The dynamic request server queries the volatile DB server for the required information of the given user that made that initial request. Those requests can fall into the following categories:

- Animation requests: a client application may want to request information for a given animation (such as bounding boxes, number of frames, loop sequence, etc.), push an animation into the motion queue of an object, or notify the server that it started an animation.
- Object state requests: notifications have to be sent when an object moves from one world to another and that information needs to be stored server-side. In addition, when a player enters a new world, the client application need to fetch all the new object data from the server.

- Variable state requests: the game engine contains a number of variables that need to be persistent throughout the playing sessions. When their value changes, the new value needs to be updated on the server.

It can be observed that all client side requests that involve the volatile DB layer are not explicitly invoked by the user. They are game-play related events that happen at a very frequent rate.

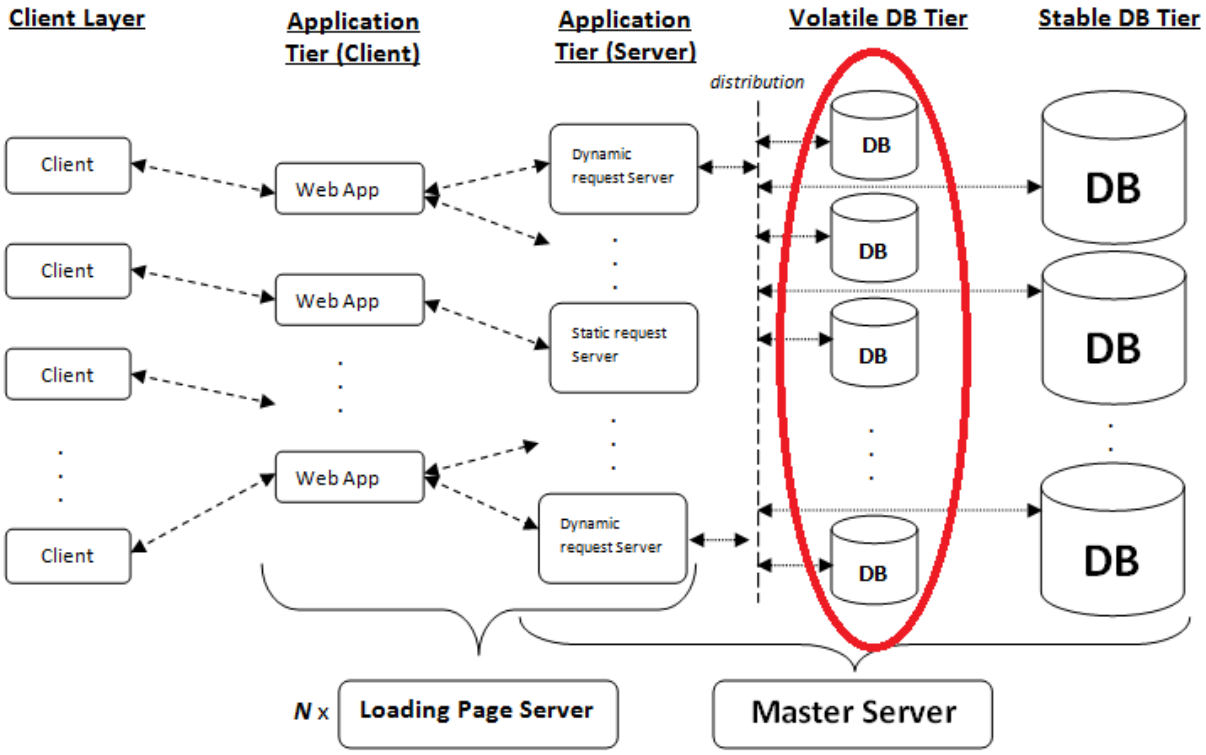


Fig. 3.5: the overall 4 tier client-server architecture with the third layer being highlighted

Although the volatile and stable database server layers fulfil different functions, there is a common element between them: the user information on the basis of which the player for which the queries are executed is known. This shared user information is physically represented by the *user_names* table. Fortunately, this table changes its contents only in the case in which the player goes into his account settings and modifies his account details, which is a very rare happening. When this takes place, the request is initially carried out to the stable DB layer that updates the information and then it asynchronously updates the information on the third layer. In order to avoid race conditions while these operations are done, the inner-server user-id that is used for the given player is never modified, even though the username or password is changed.

To know which volatile database server contains the information of a given player from a stable database server and vice-versa, a lookup table is used on both sides, similar to the one used on the dynamic request servers. And similar to the second layer servers, the information in those tables is modified through a message exchange with the master server. The sequence diagram below presets how that communication takes place:

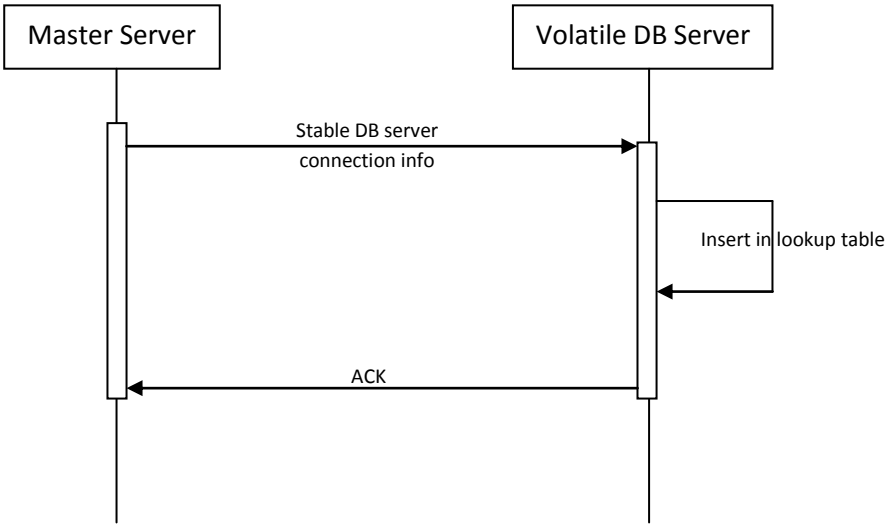


Fig. 3.6: a sequence diagram of the login communication that involves the volatile DB server

At login, after the master server chooses the database servers that will be used for the given user and before it responds to the second layer server (see figure 3.4.), the master server also communicates with the third and fourth layer. For the third layer, the master sends a message to the volatile DB server that is chose giving the connection information for the corresponding stable DB server. In the end, the database server sends the master an acknowledgement message to let it know that it can continue its operations in the login process.

3.5. Stable database server communication

The stable database servers constitute the fourth and last layer of the proposed client-server architecture. The servers on this layer store user-related information, chat and webshop information. The frequency of the client requests that involve the information stored on these machines is a lot lower than the one for the volatile DB servers, as all requests that affect this information have to be explicitly started by the user. These requests fall into one of the following categories:

- Modifying user account details.
- Requesting the webshop content.
- Notifying that a webshop item has been bought.
- Sending or fetching chat content.

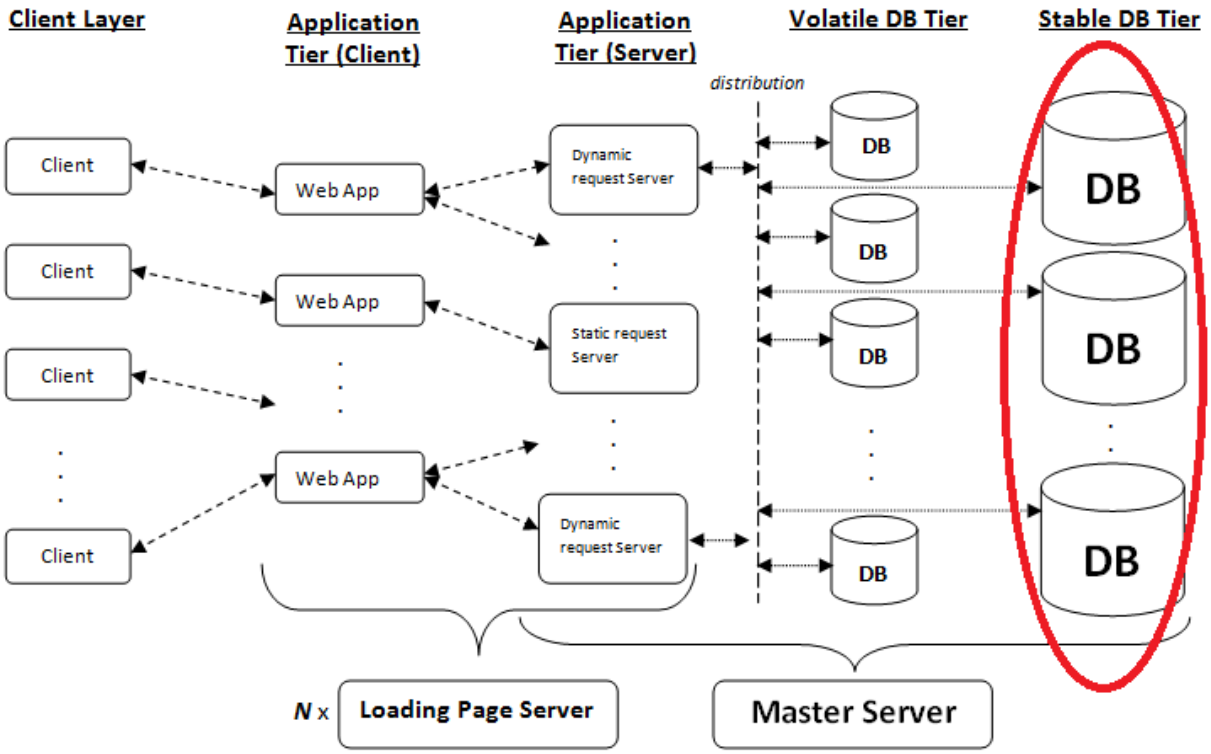


Fig. 3.7: the overall 4 tier client-server architecture with the fourth layer being highlighted

The communication between this fourth layer and the volatile database layer is the one that has been already mentioned in section 3.4, based on the information that the given stable DB server shares with some of the servers in the third layer. To keep track of these connections a lookup table is used. That lookup table is updated by the master server, the message exchange taking place between the two being very similar to the one between the master server and the volatile database servers.

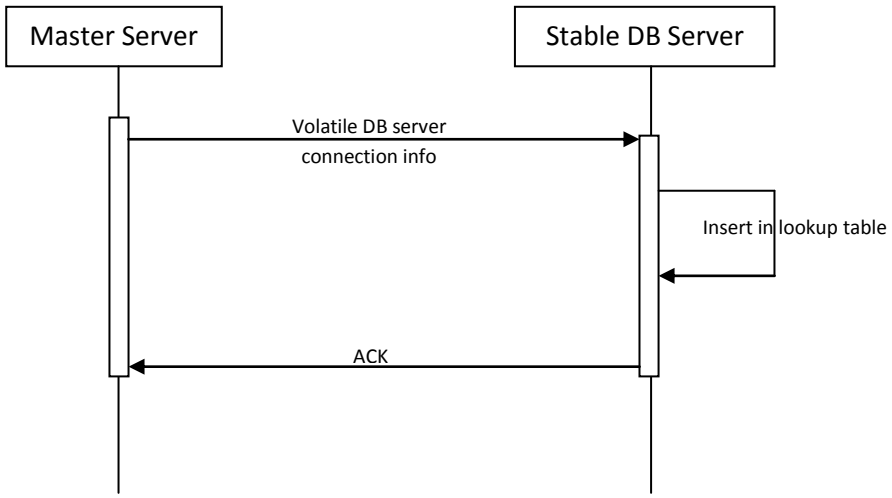


Fig. 3.8: a sequence diagram of the login communication that involves the stable DB server

As mentioned before, this message exchange takes place after the master server chooses the database servers that will be used for the given user and before it responds to the second layer

server (see figure 3.4.). For the fourth layer, the master sends a message to the stable DB server that is chose giving the connection information for the corresponding volatile DB server. In the end, the database server sends the master an acknowledgement message to let it know that it can continue its operations.

3.6. The Loading Page Server

The Loading Page Server has the responsibility of handling the initial interaction between the client and the server-side structure, linking the client to the second layer of the client-server architecture. How the communication between this machine and the client application takes place has been discussed previously in section 3.2. when the client application interaction was described. However, the manner in which the loading page server knows to which machines it has to redirect the clients has not been explicitly described.

The loading page server receives the connection information that it passes on to the clients from the master server. In order not to impose a message exchange with the master for every client request of a new session, this communication between the two takes place completely independent from the client requests, being based instead on the server loads of the second layer machines. To be more precise, the master server notifies the loading page servers that the players have to be directed to certain machines, and when those machines reach their maximum load, the master observes that, and it chooses new machines for the loading page servers to redirect the new clients to.

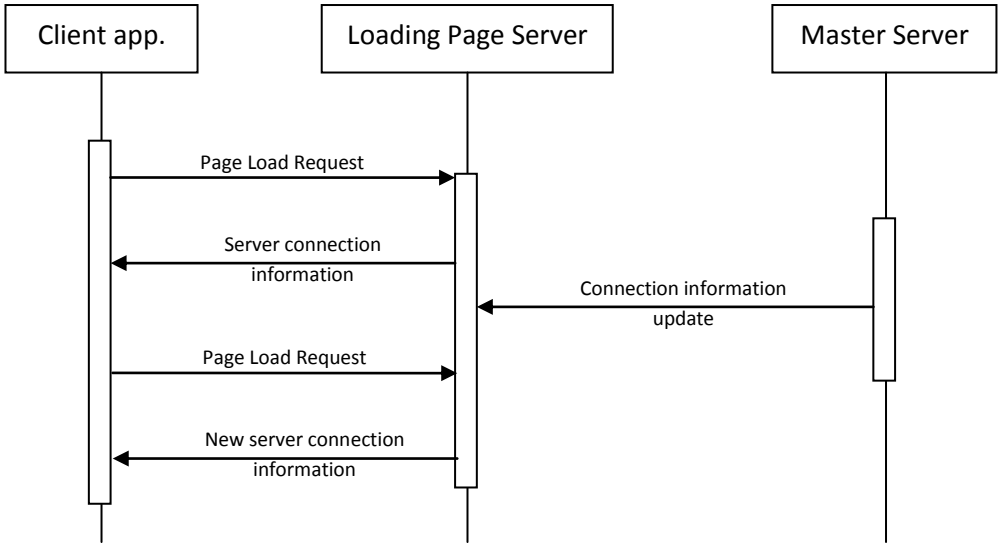


Fig. 3.9: a sequence diagram of the communication between the master server and one of the loading page servers.

This approach allows a clear separation between client and server. The Loading Page Servers act as a bridge between the two, establishing an initial connection and separating the master server from the interaction with the client layer.

3.7. The Master Server

The master server is the one machine that carries the most responsibility from the entire client-server architecture. It communicates with every layer on the system being the one that handles all the logistics and decision making inside the server structure.

This section will be divided based on the master servers' responsibilities. These responsibilities include:

- Login administration.
- Load balancing.
- Closeness graph updating.
- Data management across DB servers.

3.7.1. Login administration

One of the most important scenarios that is overseen and controlled by the master server is the login process. As seen in the previous sections, the master server communicates with every layer of the server-side architecture at login. While in the previous sections the main focus was set on the other types of machines and only the part of the login process that included that type of machine was presented, figure 3.9. presents the entire login process, practically stitching up all the interactions that were presented before.

At the moment the client accesses the game website, the client application first sends a request to a loading page server to find a free dynamic and static request server. The loading page server, having this information from the master server, sends the result to the client layer. Then, the client starts the communication with the dynamic request server. Once the server is notified that a new user wants to start a session, it asks the master server to choose a volatile and stable database layer. At that moment, the master sends the player information to the two database servers in order for them to store it in their database. Along with the player information, the master server also sets up the communication lines between the two database server layers. It does this by sending the volatile database server the connection information of the according stable database server for the given player, and vice-versa. When the database servers finish storing their data, an acknowledgement message is sent back to the master letting it know that the database servers are set up correctly for the given user. The master server can then send the dynamic request server the connection information for the database machines. Finally, the dynamic request server stores that information in the local lookup table and sends a login response back to the client layer.

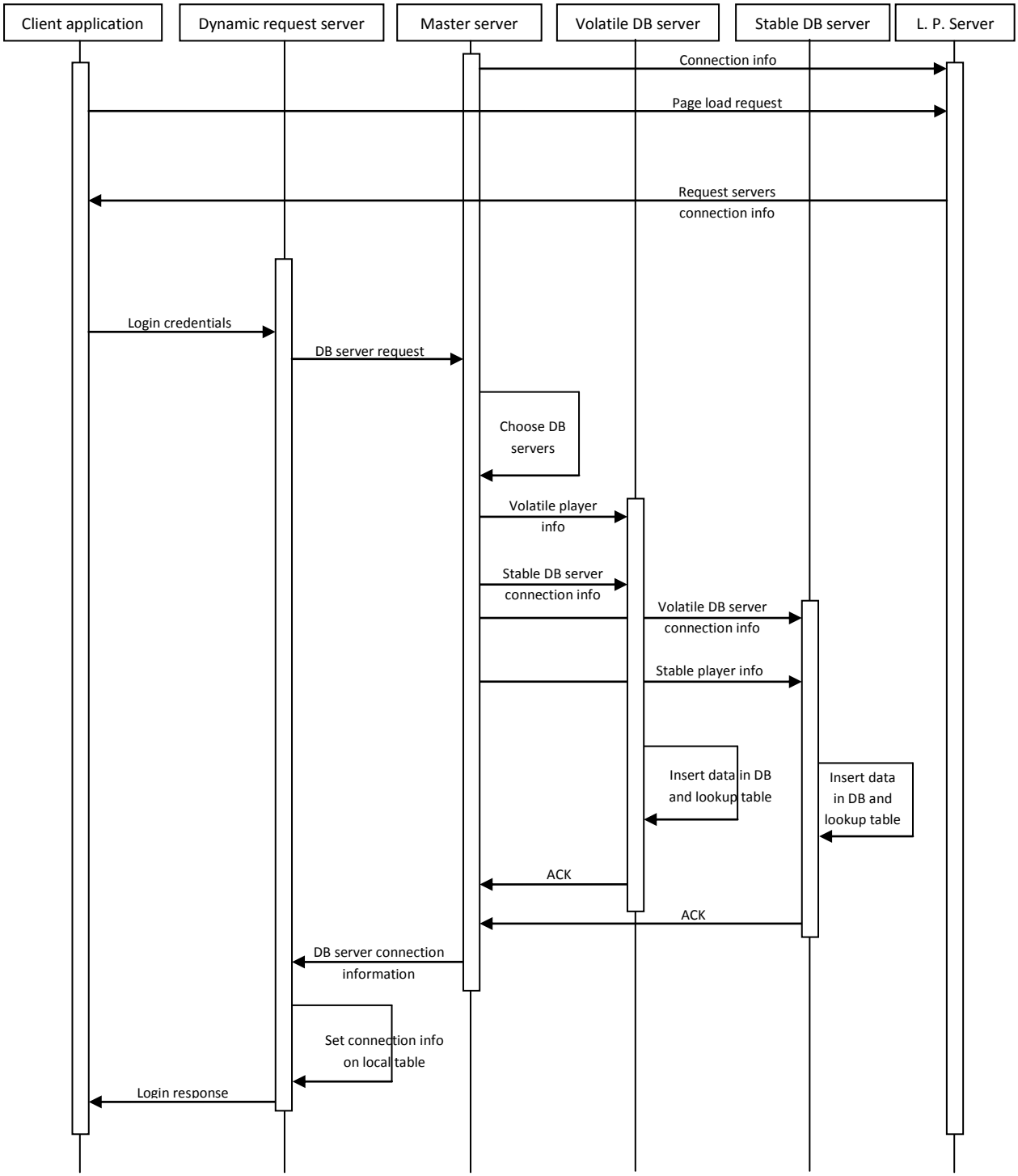


Fig. 3.10: a sequence diagram of the overall login process

Once this login process ends, the client application then communicates directly to the second layer and if needed, the dynamic request server connects directly to the database server layers using the information in the lookup table that was initially stored by the master server. During a session, the master does not interfere with this flow under normal circumstances. It does interfere when data needs to be moved across servers, because of either server overload or player interaction. These scenarios will be discussed later on in this section.

3.7.2. Load balancing

The master server is the one that is in charge of distributing the data equally across the machines of each layer. However, each layer has different rules in terms of scalability.

The most important fact related to the server application layer with regard to load balancing and scalability is that each user is assigned to a server from that layer for the whole length of the given session. As a result, data transfers because of overloading between machines from the second layer is never necessary. However, the following scenarios are possible:

- At least one server has not reached its full capacity at the moment a new player enters the game, so that player can be assigned to the given machine.
- All the servers have reached their full capacity and a new machine needs to be started up.
- After numerous logouts there are multiple servers with very low capacity that can be compacted into a fewer number of machines, closing down the unnecessary ones.

These three scenarios apply to both the static and dynamic request servers.

The first one is the one that occurs in most instances when a player starts a new session. In this case the normal login scenario that was described above takes place, and no load balancing operations are required.

The second scenario does not necessarily take place when a user logs in. The master server communicates with the second layer servers constantly, receiving information regarding their memory and CPU load. Based on this data, it foresees if a new machine needs to be set up, taking action in advance so that the players that can no longer be assigned to the full machines no dot have to wait for the new machine to be set up. The operation of starting and configuring a new machine is completely handled by the master server and no external human involvement is needed. The technical aspects of this operation will be described in the results section of the paper.

The third and last scenario is the one that does require a data transfer between two machines of the second layer of the architecture. This can take place only between two machines of the same type, either two dynamic request server or two static request servers. Because when placing the players on these servers, the master does not choose at random a machine, but tries to fill one machine before moving to the next, complex scenarios such splitting the data from one machine over a list of other servers that each have a small free capacity are not taken into account. This is done because of the small probability of this being ever necessary due to the way the players are stored on the machines and because of the high latency which is produced by involving more machines into this data transfer. Thus, the only situation that is treated is when one machine can transfer all its information to another, that machine being shut down afterwards. The decision of taking this action is made by the master server, based on the CPU and memory load information that it receives from the servers. The sequence diagram below shows how this interaction occurs.

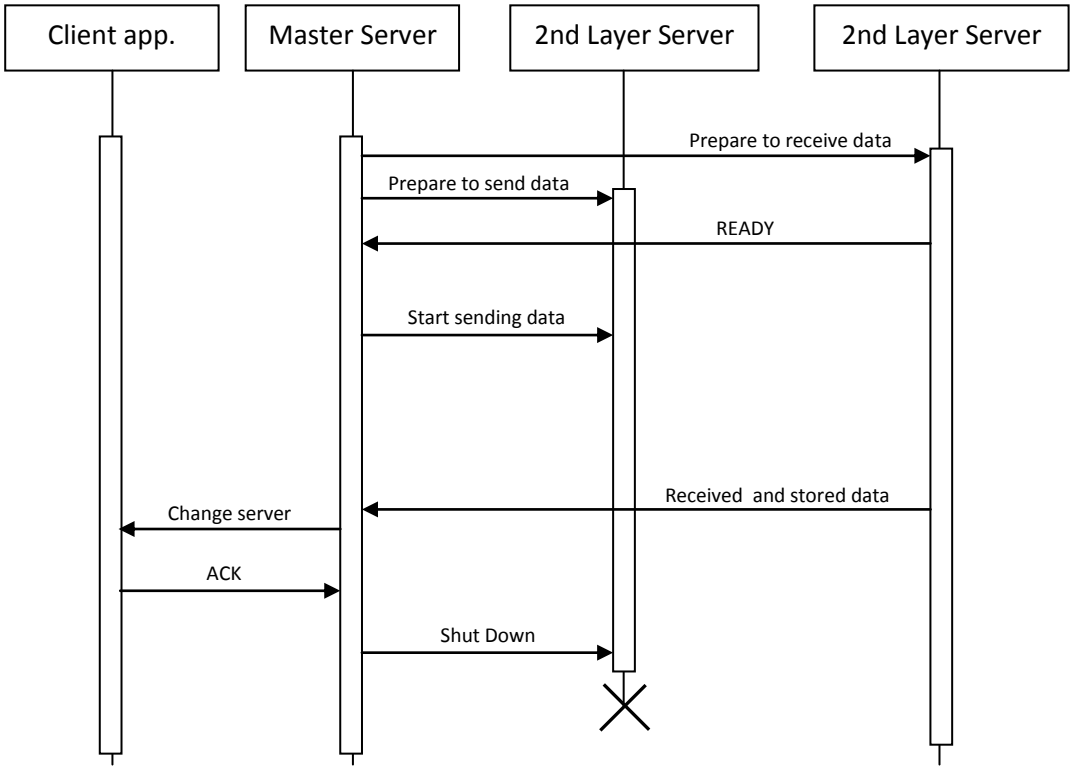


Fig. 3.11: a sequence diagram of the scenario in which a second layer server (either dynamic or static request servers) needs to be shut down.

It can be observed in the above figure that the clients continue cu communicate with the machine that will be shut down until the master server notifies them that they have to connect to a different machine. This only take place once the master is certain that the given server has stored the information correctly and is ready to receive the new players.

In order to avoid a situations in which machines are shut down and then immediately a new machine is requested, the maximum load accepted for a server after receiving data from a machine that needs to be shut down is not the same as the maximum load that the server can carry. This assures that even if there are two machines that can be combined into a single one and all the rest are at full capacity, when new players log in there are still enough resources on the new combined machine to handle them, so that a new machine does not have to be spawn in the very near future.

Regarding the other two database layers, the master servers avoids server overloading for those machines exactly in the same way as it does for the second layer. The three scenarios described above also apply for the volatile and stable database layers.

The major difference between the second layer and these two database layer is that besides these overloading scenarios, there are also other situations in which data needs to be transferred from one machine to another. These situations, however, will be described in section 3.7.4. where the main focus will be how the master oversees data transfers in the database layers.

3.7.3. Closeness graph updating

One of the operations that the master server is responsible for is updating the closeness graph. The master server does this by using the session information of each user that keeps track of the interaction that player has with other users. The structure of this information, the moments at which it is sent and how it is processed will be detailed in the next chapter that focuses in the closeness graph itself.

3.7.4. Data management across DB servers

As mentioned in the previous sections, there are special cases in which player data needs to be transferred between database servers that are not related to load balancing. These operations needs to be done due to player interaction requirements, to allow players to interact seamlessly with other users even if server-side they are initially situated on different servers. In all these cases, the master server controls and oversees the data transfers.

At login, the master server decides on which machine the player information is set. This is done based on the closeness relations that the given player has with the other players that are online at that given time. However, this attempt at predicting the player interaction does not guarantee that players from different servers will not connect to each other. When that happens, the master must facilitate the movement of one player or both to another machine, also updating the other servers that are affected by this change.

When a player wants to start a multiplayer activity with a player that is on a different server, that request is sent from the client to the second layer server and then to the database server. The later then checks if the other player has his information stored on the same machine. When it determines that this is not the case, the master server is notified. As the master server stores locally on what servers each player information is stored, it determines which are the two machines on which the two players are currently situated on. Then, based on the closeness graph information, the other active connections that the players have at that moment, and the load of each machine (a detailed description of this decision process will be made in the next chapter when the concept of the closeness graph is explained), one of the following solutions will be applied:

- One of the two players will be moved to the other player's database server. This does not means that his data is the only one that will be moved across, but the data transfer will only be unidirectional between the two database servers.
- Both players will move to a newly created machine. This scenario can happen when the data that needs to be transferred one way or another would overburden the machine that would receive the data. It that case a new database server needs to be initialized and both machines will send the data to that newly created machine.

Both these situations and the flow of events that takes place in each case will be explained below with the help of sequence diagrams. The starting point will be the moment after the master server decided which is the appropriate action that needs to be taken.

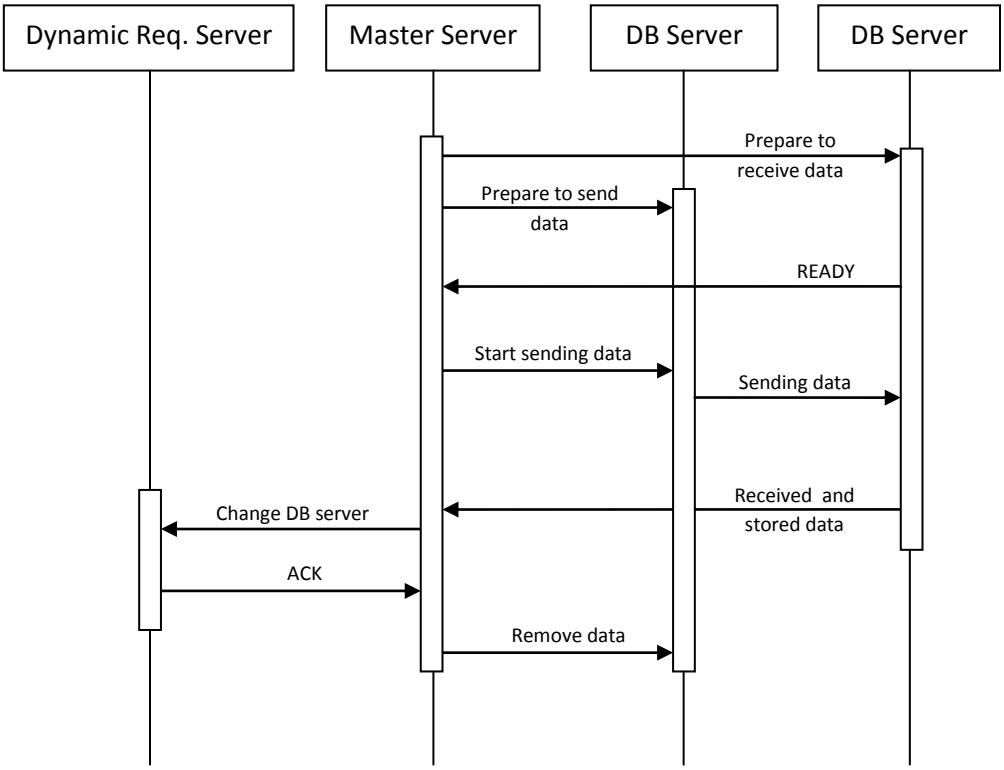


Fig. 3.12: a sequence diagram of the scenario in which player data needs to be transferred from one database server (either volatile or stable) to another.

In the first situation, the master server starts by notifying the sender and the receiver about what data they will send/receive and to/from whom. Once the receiver is ready, the master notifies the sender to transfer the data across. Afterwards, the master awaits for the receiver to send a confirmation message that the data has been received and stored properly in the database. At this time, the master can notify the second layer dynamic servers that they have to update their local lookup tables for the players that had data transferred to the new machine. Then, when the master server receives and acknowledgement that this has been done, the data from the sending database server can be removed, as all the requests concerning that data will now be directed to the receiving machine. The two database servers in this, but also in the following scenario, are of the same type: either two volatile database servers or two static database servers.

The second scenario differs from the first in the sense that two servers need to transfer data to a single receiver. As that receiver is a newly spawned machine, the first action taken in this case is for the master server to set up the new machine. Once the server is set up, it lets the master know that it can be included in the overall structure. At that time, the server notifies it that it will receive data, giving the credentials of the senders and the machine responds with an acknowledgement. Then, the two senders are informed to pump the data across and the master waits for the receiver to inform it that the data has been received and stored. In the end, as in the previous scenario, the master communicates with the second layer servers updating their lookup tables, and finally notified the senders to delete the data they transferred.

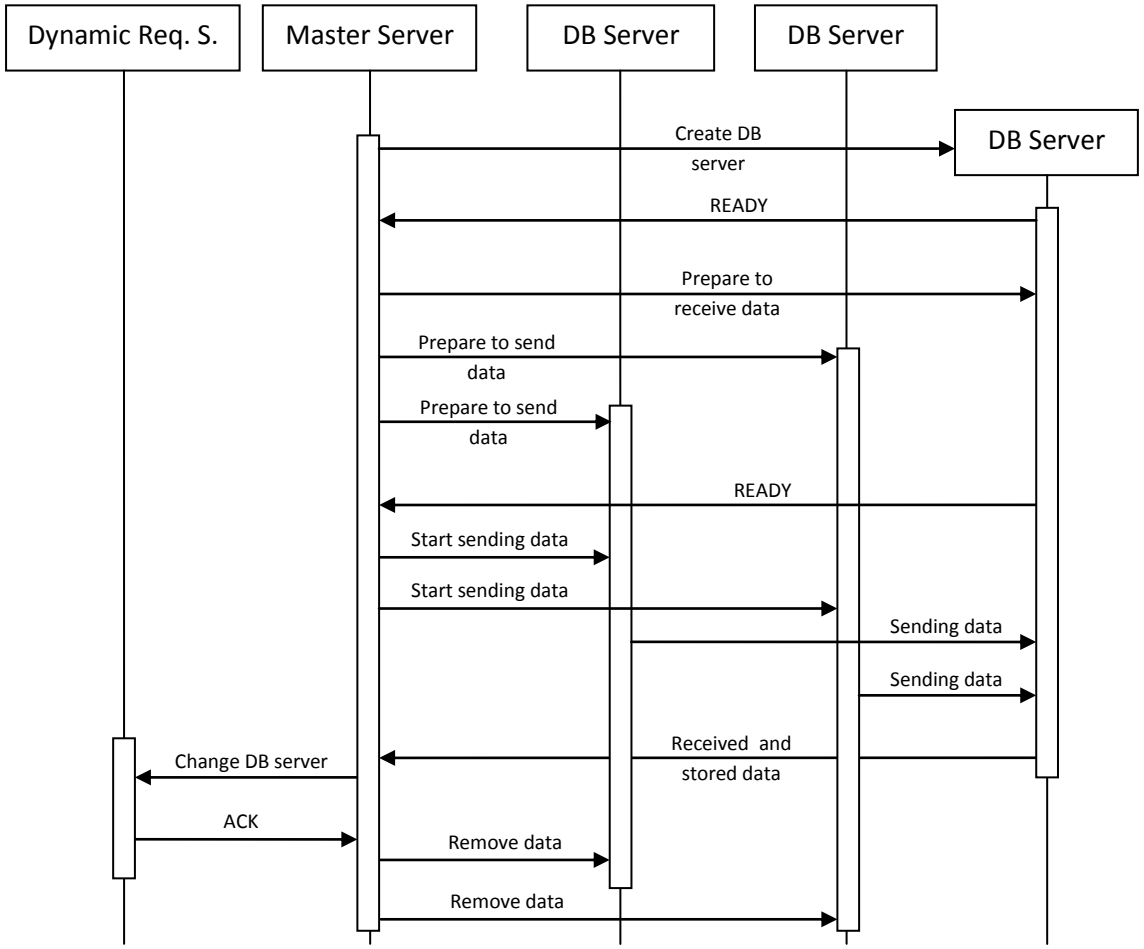


Fig. 3.13: a sequence diagram of the scenario in which player data needs to be transferred from two database servers to a newly created machine of the same type.

In can be observed that in both scenarios there is a time interval in which the data is stored on both machines. This is done so that situations in which a database machine is queries for invalid information can be completely avoided. The second layer servers start querying the new machine only when the master can assure it that the given information is stored properly and that machine can handle requests.

3.7.5. Limitations

The main limitation of having a single machine that oversees and handles data transfers within the entire architecture is the fact that it acts as a bottleneck. If something goes wrong with the master server, then the entire architecture stalls and cannot function properly.

A proposed solution for this is to have a backup machine that sits in the background, having sporadic communication with the master server. This backup machine would contain all the closeness graph information that is normally handled exclusively by the master, so that it can step in very quickly if an error occurs. All the other information such as which servers handle the data of which users is not necessary to be constantly updated. Instead, in the scenario of a

master server failure, the new machine asks each server for their lookup table information and using that, it makes its own table of player-server relations.

Another limitation is the fact that as the master has to communicate with multiple machines and has numerous responsibilities, it can become overloaded if the size of the overall system becomes too big. Although theoretically this can be an issue, the number of machines that would trouble a properly configured high capacity machine is higher than the maximum amount of machines that the architecture will be limited to. The backup solution for this highly implausible scenarios (as well as for the case in which the maximum capacity of the system is reached) is for the master server to send a message to the users that want to log in at that time that the servers are full and they should try again later. While this is not a very elegant approach to solve this issue, it is a common failsafe solution that is used even by the very big companies in the gaming industry.

4. THE CLOSENESS GRAPH

4.1. Introduction

This chapter will define and describe the functions of the closeness graph. This graph is used by the master server in order to determine the most efficient way of transferring data between servers. The closeness graph does not only offer information regarding the best data movement that needs to be done at a certain point, but based on its contents, it also offers a prediction of future possible transfers.

The closeness graph is defined as an undirected graph whose nodes are represented by the users of the game. The connection between two nodes is established when there is a relevant interaction between the two players that represent the nodes. In addition, each connection also has a value associated to it. The value is determined based on the strength of the interactions between the two given players. A connection is formed when two players add themselves as contacts, as without that, they cannot interact in any way. The value of that connection however remains at a minimum if no further communication takes place between the two.

In this chapter, the first part will give a detailed description of how the calculation model of the value of each connection is determined. Then, once the value ranges and the dynamics of the graph are explained, the closeness graph is placed into the overall architecture and a description is given of how and in what cases it is used.

4.2. Connection value determination

The value associated with each connection has to give a clear description of the strength of the given connection between the two players. In order to see the best approach of modelling the function to determine this value, a quick analysis of how these values will be used needs to be given.

The main use of these values and, implicitly, the closeness graph is determining how data can be moved most efficiently between the database servers. Because the database servers are split into two categories (stable and volatile DB servers) and, as it was mentioned in the previous chapter, data transfers only take place between machines of the same type, the contact between the players can also be separated into two categories.

Regarding player interaction, the volatile servers can be correlated to in-game interactions between players as they contain all the game-play related data. All the actions that involve the actions of the in-game characters can be associated with the volatile database servers. On the other hand, the only data contained by the stable database servers that is related to multiplayer interaction is the chat communication. Because the in-game interaction is completely independent from the chat interaction, and each one has an impact on data transfers in two different independent layers of the architecture, the connection will also be defined by two separate values for each type of player interaction. From now on, the two will be referred to as game-play related closeness value and chat related closeness value. Although these two values are totally independent, they are determined in a similar manner, and they are also used in the same way by the master server. They are just used on different layers of the architecture (the stable and the volatile DB layer). Both values are determined in a similar manner, with minute differences, they just have a different input: one has the game-play interaction data between the two players and the other has as input the chat interaction data.

4.2.1. Parameters

To measure the closeness value between two players a number of elements have to be taken into account. Each factor will be discussed below. When differences occur between the calculation method of the two closeness values (game-play related and chat), they will be specified for each factor.

Firstly, the length of the interaction plays an important role (*Li*). Although the data transfer has to take place regardless of the length of the interaction, the length does play a role in predicting the chances of future contact between the two. For the game-play related value, the length of an interaction is simple to define: it is the time in which the two player's characters interact; and this information can be easily fetched from the servers. For the chat related value, it is more complicated to define the length of a given conversation. If the length of the messages is taken into account, false evaluations can occur when people copy/paste big texts through the chat messenger. If the number of messages is taken into consideration, this can also give mixed results, as the writing style differs for different people. While some write high numbers of small sentences, others write bigger texts more rarely. There are a number of researches that have been done, but none actually gives an answer regarding how to define the length of a conversation taking place in writing on the web. Most research is focused either on classifying users as a certain type (looking at message length, pauses between messages, etc) as in [3], either their focus is mostly related to linguistics, developing algorithms in order to spot forbidden words in a chat room, as in [7] or [8].

Taking all the factors from the previous paragraph into account, a small scale testing session involving approximately 20 individuals was organized. Using this data, the chat interaction that took place between these users was analyzed, with focus on the time lapses between messages. This was important in order to define when a conversation should be considered to have ended. The result was that each conversation from the testing session could be considered as ended after a gap of more than three minutes between two consecutive

messages. Thus, the final chosen definition is the following: a length of a conversation is defined as the length of time in which messages are sent between the users in both directions with a maximum gap between two consecutive messages of three minutes.

Secondly, the total time in which both players are online must also play a role in the closeness value (*Ti*). With this we can determine, combined with the length of interaction, the percentage of time in which two players are in contact out of the maximum amount of time that they could have been in contact. This value is determined in the same manner for both closeness values, taking the login and logout times for each user and comparing them with the times of the other players that they are connected to. Using this value along with the previously defined one, a percentage of the time in which two users interact out of the entire session length can be determined. This value can then be used as a parameter in the final function.

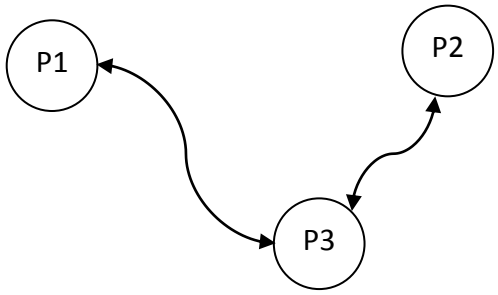


Fig. 4.1: the way the three players in the example are connected in the closeness graph.

Furthermore, the period of time in which the contact between the two players takes place with regards to the present moment also has a significance. This practically exposes the evolution of the interaction between the two. To illustrate this, imagine there are three players: P1, P2 and P3. P1 is connected to P3 and P2 is also connected to P3. If both couples have had 10 sessions, but P1 and P3 have only interacted in the first five and P2 and P3 only in the last five, to assign the same closeness value to both pairs would not be fair. It is clear that the interaction of the first pair is on a downward slope while the second one has an opposite development.

	Sessions (0=no interaction, 1=interaction between the two)									
P1&P3	1	1	1	1	1	0	0	0	0	0
P2&P3	0	0	0	0	0	1	1	1	1	1

Fig. 4.2: a table representing the interaction between two pairs of users over a length of 10 sessions in which both of them were online.

To differentiate between these types of situations and to determine the *staleness* of a connection between two users, a weight is assigned to each session. If *X* is defined as a number of sessions taken into consideration, the weighting system will be as follows:

- The weights will be represented by values in the (0,1] interval.
- The most recent session will have a weight value of 1.

- The second most recent will have a weight value of *(X-1)/X*, the third most recent will be *(X-2)/X*, and so on, until the last session taken into account that will have a weight value of *1/X*.

To apply this schema to the previous example, the first pair P1&P3 will have assigned a value of 1.5 while the second pair who is on an upward slope in term of contact between the two will have assigned a value of 4.

It can be observed that the time passed from previous sessions was measured in the number of sessions passed until the present moment, and not in the actual amount of time that has passed. This was chosen in order to avoid situations in which users are not online for a long time (going on vacation for example). If the later would be applied, then after such a period, all the connections of that user would be weakened significantly although in reality this should not be the case. Real time, however, is still influential as the sessions are taken over a given period of time (the last month or the last 10 sessions if the user has less than 10 sessions in that period).

4.2.2. Closeness function

The closeness function makes use of the parameters that were described in the previous section. It has to be mentioned that this function is used to determine both values for the game-play related value and the chat related value.

The function will be referred to as $F_c(\ t_s(c))$ where c is a connection in the graph and t_s represents the series of sessions that took place in the period of time over which the closeness value is determined. The interval the function is defined on is [0,1] with the minimum and maximum being defined in the following form:

- Minimum: when the two players did not have any contact over the specified period of time.
- Maximum: when the two players have had contact for the entire time that both of them were online at the same time in the specified period of time.

The function is represented by a sum of variables, each one standing for the contact within one session. The variables are also defined on the [0,1] interval, representing the percentage of the session time in which the two players were in contact. These are weighted as mentioned in the previous section, and the final value is divided by the total sum of the weights so that it remains in the [0,1] interval.

The formula definition for $F_c(c, t_s)$ is the following:

$$F_c(\ t_s(c))\epsilon[0,1]$$

$$t_s(c) = \{S_1(c), S_2(c), S_3(c), \dots, S_X(c)\}$$

$$S_i(c) = \frac{Li_i(c)}{Ti_i(c)}$$

$$F_c(t_s(c)) = \frac{X * S_1(c) + (X-1) * S_2(c) + (X-2) * S_3(c) + \cdots + 2 * S_{X-1}(c) + 1 * S_X(c)}{1 + 2 + 3 + \cdots + X}$$

The first equation simply states the interval on which the function is defined. The second one shows that t_s consists of a series of variables, each one being representative for a particular session. The third equation shows how these variables are determined, where *Li* and *Ti* are the parameters mentioned in section 4.2.1, the total time of the session and the interaction time of the session. The final equation is the definition of the function, summing up the variables for each session with an added weight to them based on how recent they are. In the given formula, for the *X* number of sessions that are analyzed, S_1 represents the most recent session and S_X represents the number *X* session going from the current moment into the past.

The same equation is used to determine both the game-play related value and the chat value, the only difference consisting in the values of $Li_i(c)$ and $Ti_i(c)$, as mentioned in the previous section, and both values are assigned to the given connection.

4.3. Updating the graph

As mentioned in the previous chapters, the graph data is stored in the master server and it is constantly updated. The graph is updated whenever a user's data needs to be transferred to a different machine. This can occur in the following two cases:

- The user had logged out, bringing an end to his session. In this case, the user's data is fetched from the servers it was stored and put on the master server.
- The user need to be transferred to a different server due to interaction with a player whose data is stored on a different machine.

When one of these two cases occur, the dynamic request server that hosted that given user composes a packet of data describing the parameters of that session such as:

- Total length of the session.
- Connection to rooms owned by other users (game-play related interaction) and the length of those connections.
- Chat communication that took place during the session.
- The start time (if no end time exists yet) of the users with whom the given user had contact.

Once this data is fetched from the database servers and determined inside the dynamic request server, the packet is sent to the master server. Once the master server received the packet, it analyzes the game-play related interaction and the chat related interaction that it has received. For each one, it looks at the given user and sees with who that user has communicated during that session.

For each one of those connections, it looks at which one of those users was still online at the moment that the current user logged out. This is important because it means that the current user is the one that ended the online session between the two (the passage of time in which both players were online at the same time).

Only for these connections, for the freshly closed online sessions, the closeness values is updated. This is done because for the users with which the current logged out user had contact, but who logged out before him, the same process already took place, and the master server had at that point already all the needed data to update that given connection in the graph.

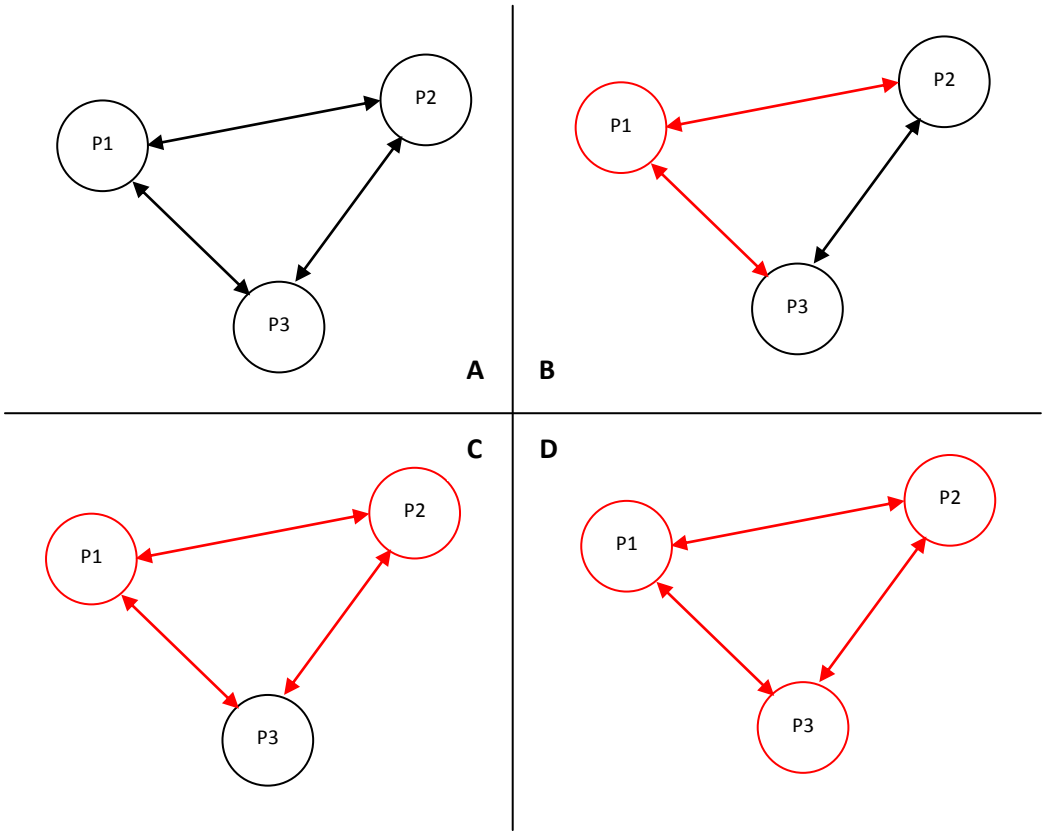


Fig. 4.3: an example of how the closeness graph is updated for three inner-connected players. Figure B is the graph after P1 logs out, C is after P2 also logs out, and figure D is after the last user, P3, logs out. The red arrows mean that the given connection has been updated.

To better understand this, consider an example group of three players: P1, P2 and P3. All three are connected with each other and all three are in contact with each other. At a given point P1 logs out. That means the master server can update the closeness graph connections between P1 and P2 and the one between P1 and P3. When P2 logs out afterwards, the master server now only has to update the value of the connection between P2 and P3 as the other one was already updated. Finally, when P3 logs out, no updates need to be executed on the graph

as all the online sessions in which P3 and another user were online at the same time were already take into account. This can be also observed in Fig. 4.3. above.

As mentioned before, the closeness graph is undirected. This is because the connection between two nodes is always dependent on the common interaction between the two. All the data that is taken into account to determine the value of each connection is common to both players.

4.4. Usages

This section will discuss the scenarios in which the data stored through this graph is used.

As mentioned in the introduction, the closeness graph has two main usages, based on the type of scenario:

- To determine the least amount of data that needs to be transferred between two database servers at the given time in order to properly facilitate a move of one or more players from one server to another.
- To predict future movements of players across servers, the master server using those prediction to limit future data transfers to a minimum.

The scenarios in which the two cases apply can be very easily separated. The first case applies to the situations in which players need to be moved across database servers during the game, while the second case is applied when a player logs into the game.

4.4.1. Login scenario

As mentioned, the closeness graph is used at login by the master server to place the new user on the servers which would facilitate the most that user's interaction with other players. This means that the closeness graph is used in order to minimize the chance of having to move across the user's data during the given session. This is done by analyzing the connections in the closeness graph of that user and overlaying them on the players that are currently online. Each connection has associated with it a given value. For each server that contains online friends of the given user, the values of the connections with those friends are summed up. Once this is done for all the servers, the one with the highest overall value is chosen. This implies that the chosen server is the one which offers the highest probability of the user interacting with contacts already stored on that machine.

To understand this better, look at Fig. 4.4 that gives an example, showing the spread of the players across the servers on the top, and the relevant connections from the closeness graph for these players on the bottom. Keep in mind, P1 (the user logging in) can be connected to

other users in the graph, but the only ones taken into account here are the ones online at the time he logs in.
Even if the given user has other connections that have a very high closeness value with contacts that are offline at the given moment, as the graph is undirected, that value will play a role if the given contact logs in while the user is still online.

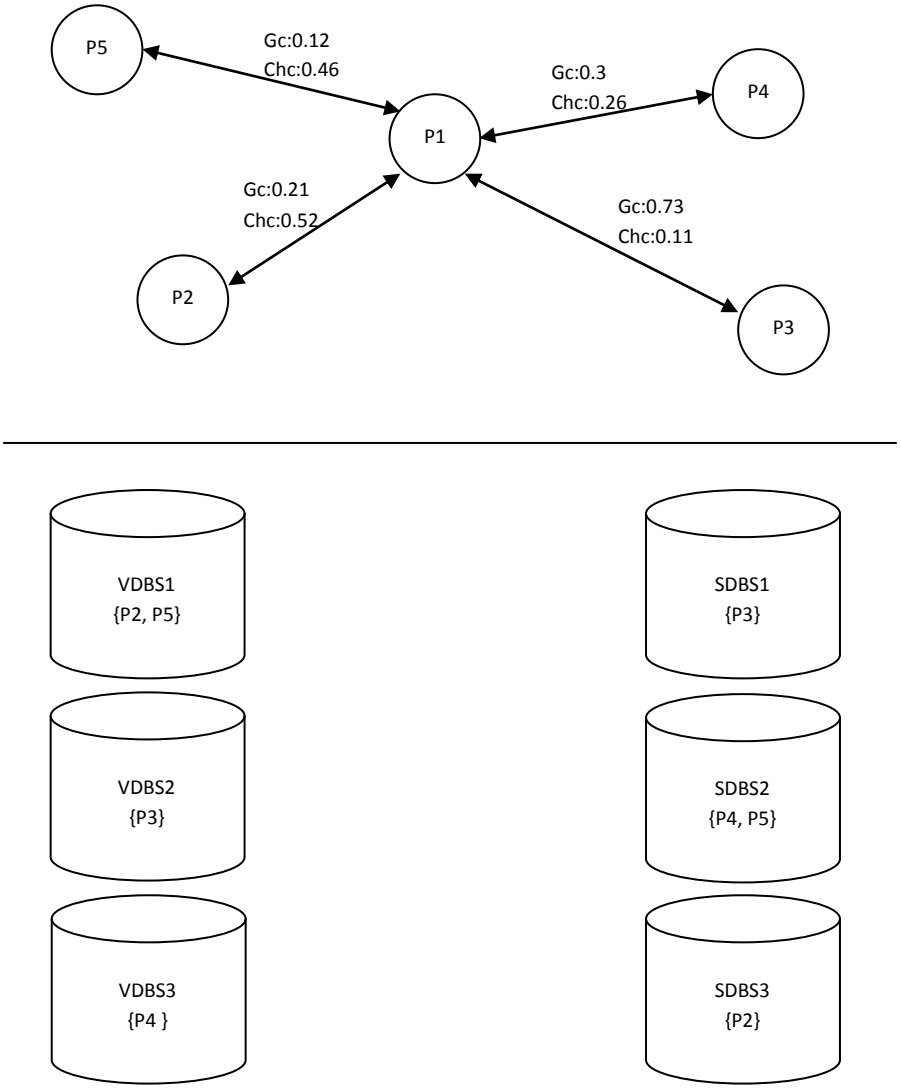


Fig. 4.4: an example how the decision process takes place on where to place the user data of a newly logged in user based on the information in the closeness graph.

In the figure 4.4 it can be seen that each connection has two values attached. A game-play related value (*Gc* in the figure) and a chat related value (*Chc* in the figure). The first is used to determine the most suitable volatile database server (VDBS) while the second is used for the stable database server (SDBS) layer in the same manner. In this example on the volatile database layer three values have to be compared:

$$Cost(VDBS1) = 0.12 + 0.21 = 0.33$$

$$Cost(VDBS2) = 0.73$$

$$Cost(VDBS3) = 0.3$$

Thus, P1's game-play related data is chosen to be stored on VDBS2, as it has the highest chances on P1 interacting with one of its contacts on that machine.

For the stable database layer, the three values that need to be calculated are:

$$Cost(SDBS1) = 0.11$$

$$Cost(SDBS2) = 0.46 + 0.26 = 0.72$$

$$Cost(SDBS3) = 0.52$$

Thus, based on the same considerations as for the volatile layer, SDBS2 in chosen. So the final two database servers chosen for P1 at login for the given session will be VDBS2 and SDBS2.

It is important to mention that the locations on which each online player's data is stored are known by the master server. Thus, all the required operations are exclusively done on that machine without burdening the other servers.

4.4.2. In-game scenario

During the game, if a player wants to visit another player that has data stored on a different machine, the dynamic request server will inform the master server of this action. The master server then, must determine the most efficient way of transferring data across database servers to fulfil this request.

In this scenario, when one player stored on a server called S1 wants to interact with another one from a different server, S2, there are two main outcomes: data is moved from S1 to S2 or from S2 to S1.To make this decision, two aspects are taken into account:

- The size of the data that needs to be transferred one way or another.
- The likelihoods that the moved players would interact with other users on the servers that they were on rather than on the servers they are moved to.

Regarding the first point, it refers to the number of players that need to be moved, as the data stored for each account has little variation from user to user. More than one player may be required to move, even though a single player initiated this action. This is the case when the player than needs to be moved is also engaged with another player at that time. In order to keep that connection, the data for both players requires to be transferred. When this scenario takes place, the first step is for the master server to determine this, the number of players required to be moved in one direction or the other.

As an extra transfer has a more significant impact than a few extra MB of data being transferred, the later has a higher priority than the first. The mentioned likelihoods are determined in the following manner. For every player that needs to be moved, the graph

connections to the players stored on the server he is currently on are summed up. Then, the same process take place taking now the connections to the players stored on the server he would be moved to. Then, the later value is subtracted from the first, thus obtaining the likelihood that the given player will interact with someone from the server he is leaving rather than with someone from the server he would be moved to. This process is done for both transfer scenarios (from S1 to S2 or vice versa), obtaining two final values, one for each case.

Finally, the number of players transferred in each direction in used again. As the two resulted values are sums of probabilities, it means that transferring more players when the likelihood difference between the two values is very low will not be beneficial. To solve this issue, each extra player data that needs to be transferred one way compared to the other is considered equal to a 50% higher chance that another user will need to be transferred. This is done because moving an extra player for a probability smaller than that would be inefficient. To take this into account, to the two resulting values mentioned in the previous paragraph a value of 0.5 is added for each player that would have to be moved. This value represents the equivalent of a 50% chance that another user will be required to be transferred. Afterwards, the final two values are compared, and the smallest one is chosen as best. The direction associated with that value is chosen and the transfer is executed by the master server.

The following example will illustrate how these decisions are made more clearly. The example is only made for one database layer, but for the other one the process is identical.

Consider a group of five online players, P1 to P5, and two servers, S1 and S2. P1, P2 and P3 are on S1 and P1 is in contact with P2. P4 and P5 are on S2. P1 wants to visit P4 while also being in contact with P2. This implies either moving P1 and P2 to S2 or P4 to S1.

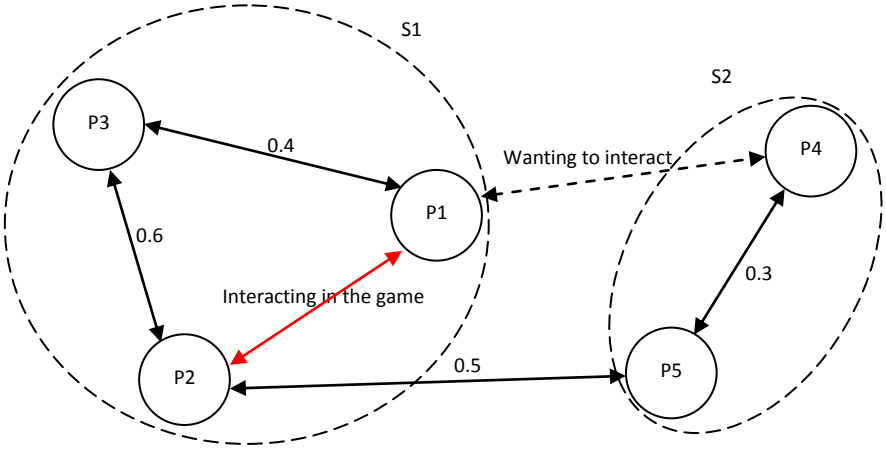


Fig. 4.5: the connections between the players in the given example, with the values of the relevant connections for the example.

As it can be seen in figure 4.5, the values that are relevant in this scenario are the ones of the connections between P1 and P3, P2 and P3, P4 and P5 and P2 and P5. The connection between P1 and P2 does exist in the graph and it does have a value, but because the two are interacting and need to be moved together, that value would be taken into consideration in both data transfer scenarios. Therefore, it can be ignored as the difference between the two

cases remains the same with or without it. Thus, the final value on moving data from S1 to S2 will be the sum of the first two pairs minus the sum of the connection values of the moved players to the players on the destination server (given by the P2-P5 connection). We will name the function that determines the transfer cost in either direction with $T()$, where $T(S1S2)$ represents the transfer cost when data is moved from S1 to S2 and vice versa for $T(S2S1)$. Therefore, the final value for moving the data from S1 to S2 will be:

$$T(S1S2) = P1P3 + P2P3 - P2P5 = 0.6 + 0.4 - 0.5 = 0.5$$

The total on moving data from S2 to S1 will be 0.3 and nothing needs to be deducted because in this example P4 does not have any other contacts on S1 except P1.

$$T(S2S1) = P4P5 = 0.3$$

Finally, to these two resulting values the number of player needed to be moved times 0.5 is added. Thus, the final values will be:

$$T(S1S2)_{final} = 0.5 + 2 * 0.5 = 1.5$$

$$T(S2S1)_{final} = 0.3 + 1 * 0.5 = 0.8$$

Thus, moving data from S1 to S2 involves moving across the data of two players with a total moving cost of 1.5. Moving data from S2 to S1 on the other hand involves only one player data being transferred with a lower moving cost of 0.8. Hence, the second solution is chosen.

4.4.3. Other scenarios

There is also an unlikely scenario in which the data transfer cannot take place due to the high load of the given machines. In this case, the groups from both machines have to be moved to a different one. The master server will first try to find an existing one that has the capacity to take in that load, or a new machine will be spawned.

In the very extreme scenario that the group that would be formed by the two players connecting would be too big even for a newly spawned machine to support, the connection is simply not allowed at that time. However, this scenario is more than improbable.

5. RESULTS

5.1. Introduction

This chapter will present the tests and the results obtained from them on the implemented client-server architecture.

The entire client-server architecture that was described up to this point was created using Amazon Web Services. For spawning and terminating machines a SDK offered by Amazon Web Services (AWS) was used. A detailed description of cloud programming using AWS can be found in [5].

The main test that needs to be done is a comparison between the system's behaviour without using the closeness graph and its behaviour with the closeness graph activated. The most important parameter in this comparison is the amount of data that gets sent across machines in a normal usage scenario.

Unfortunately, the game itself was still in its testing phase when this architecture was made for it, so an online deployment with a high number of real users was not possible. Thus, a testing rig was developed.

5.2. Testing rig description

To properly test the developed architecture a number of simulations were implemented. The two most important scenarios that need to be tested are:

- The login scenario - as at this point the closeness graph is used to put the new users on an appropriate server when active; if the closeness graph is not used, then the users are stored on the machine with the most free space.
- The in-game data transfer scenario - this is when two users stored on different servers want to interact with each other; if the closeness graph is not used, then the user that starts the interaction will always be the one who is moved.

For both these scenarios two aspects will be evaluated. Firstly, in a small scale test, the extra data needed to be transferred as well as the extra CPU usage of the machines when the closeness graph is active will be determined.

Secondly, a number of large scale simulations are done in order to determine the improvement brought by the closeness graph. These improvements should consist in a reduction of the number of data transfers that occur over a certain period of time.

Finally, the two aspects are put head to head in order to see the overall improvement of the system with the closeness graph active compared to the one that functions without it.

5.3. The small scale testing rig

The point of this rig is to compare on a unitary scale the difference between the two systems. That implies that both the login and the in-game transfer scenarios are run for a single user, once with the closeness graph active and once without. Then, the differences between the two in terms of data transfer, CPU usage, and memory usage are analyzed.

5.3.1. The login scenario

The two compared cases here are logging in a user with and without the use of the closeness graph. The only difference between the two consists in the decision process that needs to be done by the master server. On the one hand, without the closeness graph, the master server simply queries a machine, and if it has space, the it stores the user information there. On the other hand, when the closeness graph is active, the master server first checks the connections of the user to any other users currently online and ranks the servers based on this information. Then, the server with the highest rank that also has space for the data is chosen. Overall, the only difference between the two consists is a number of queries executed locally on the master server.

The obtained results showed that for both the data transfer and memory usage the differences were practically inexistent. For both cases the MySQL memory usage was 0.4%. Regarding the data transfer, the apache server on the master was analyzed. In both cases, the data transferred was around 300Kb. This is because in both cases only one or two queries need to be executed to the other databases servers in order to check their capacity status.

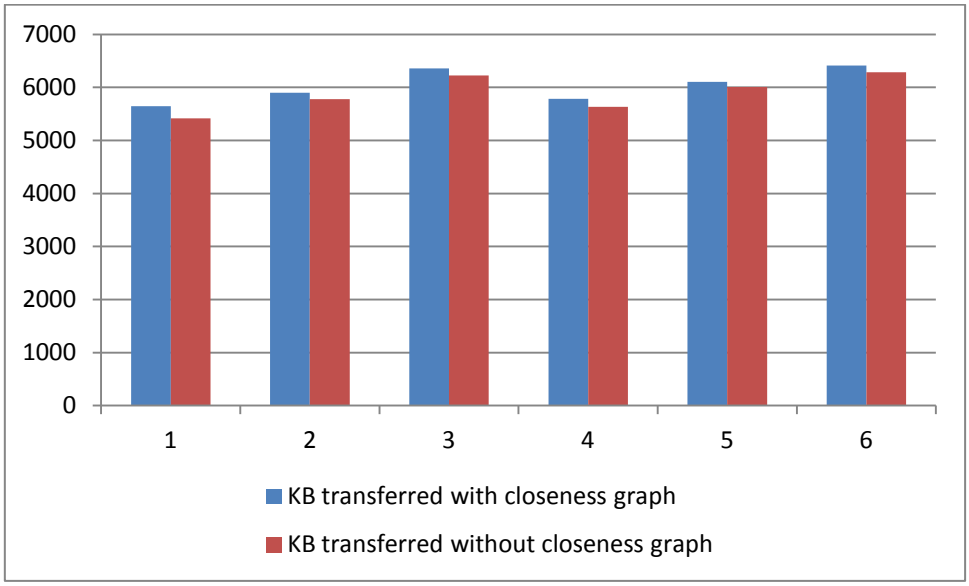
With regards to the CPU usage on the master server, the difference was also small to negligible. This is because the only extra queries that are done on the master server collecting the closeness graph data do not have a high complexity, thus not adding any significant margin to the overall CPU usage. For both scenarios the CPU usage was around 0.25%.

It is important to mention that for this test the databases were filled with 1000 test user accounts in order to take into account the sizes of the tables in a real case scenario when the queries are executed.

5.3.2. The in-game transfer scenario

The same two cases are compared as in the login scenario discussed above. When the graph is not active, the master server only looks at the player that initiated the interaction. It determines if other players also need to be moved and then transfers the data to the second machine. With the closeness graph active, the master server must analyze both sides. Afterwards, based on the connections in the graph (as explained in section 4.4.2) the most appropriate move is chosen.

The results regarding the MySQL memory usage for both scenarios was the same, around 0.7%. Regarding the CPU usage and data transfer, the following diagrams show the differences between the two cases:



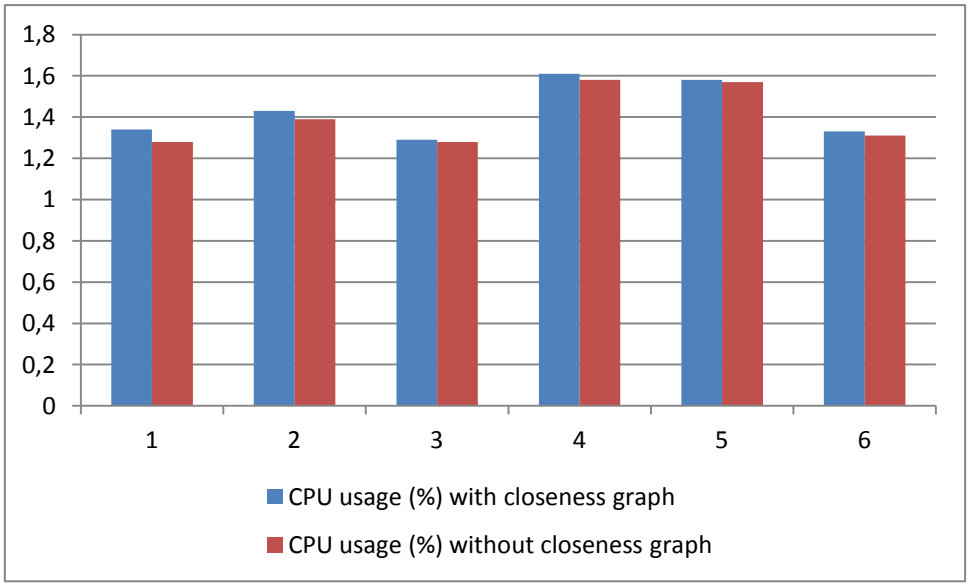


Fig. 5.1: the diagrams showing the data transfer and CPU usage (%) between the two scenarios.

Regarding the data transfer, it can be seem that the differences are minute. This comes down to the fact that the bulk of the data that needs to be sent across consists of the final user data that is transferred in both cases. As an average, an additional 2.2% of data is transferred with the closeness graph activated.

Looking at the CPU usage, the added queries that are execute when the closeness graph is active are simple, fetching data from the table that stores the graph connections and summing them up. In the end, these extra queries have little impact, as it can be seen in the above figures. The difference between the two can be estimated at around 1.3%, which is almost negligible.

5.4. The large scale testing rig

For the large scale test, as spawning tens of servers and simulating thousands actual users with random behaviour over a prolonged time with multiple sessions of gameplay is almost impossible, a testing program was developed that simulated the behaviour of the architecture. This way, thousands of users can be simulated.

The developed simulation works in the following manner: Each user is assigned a list of contacts. In order to have a good simulation of the player interaction, each player is assigned a random number of contacts between 3 and 10. For each contact a percentage value is given specifying how close the two would be in reality. That value represents the real chance that the given player would interact with that contact during a playing session. Then, a number of 10 sessions are run in order to create the closeness graph connections and assign and update appropriate values to them. Finally, once all the input data is obtained, the final simulation is started. Here, all the players are logged in taking into account the closeness graph connections and each player is given a chance to interact with each of its contacts based on the percentages defined at the beginning. When two players interact, the closeness graph values are taken into account and based on them, the direction of the transfer is decided. In the end, the total number of users that needed to be transferred from one server to another is saved.

With the closeness graph not active, the simulation is a lot simpler, as then all the users are placed randomly on the servers and there is no need to run multiple sessions in order to define the connections between the users. This is because when two users from different server interact, the one who initiated the interaction will always be the one who is transferred to the other server. Finally, as before, the simulation is ran and the final number of data transfers is returned.

The three parameters that change for each test are:

- The number of players.
- The number of servers.
- The maximum load of one server.

First, in order to prove the validity of this simulation, a test was run with the same parameters both on the real architecture and using the simulation. For both cases two database servers were used, each with a capacity limited to 100 and 100 users were logged in. Twenty tests were run in total, ten on the real system and ten with the simulation, five with the closeness graph and five without for each.

With the closeness graph not active the following results were obtained:

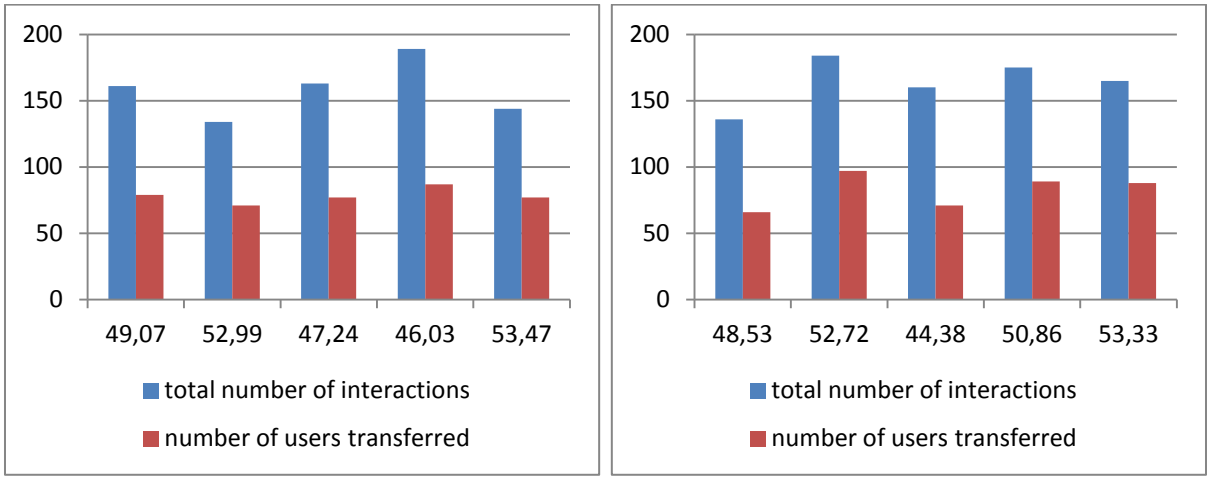


Fig. 5.2: the diagram on the left represents the results obtained when running a session for 100 users on the real system while the one on the right shows the results when using the simulation. The numbers on the bottom stand for the percentage of the total number of interactions that the number of transferred users represents.

When the closeness graph was active, the results were the following:

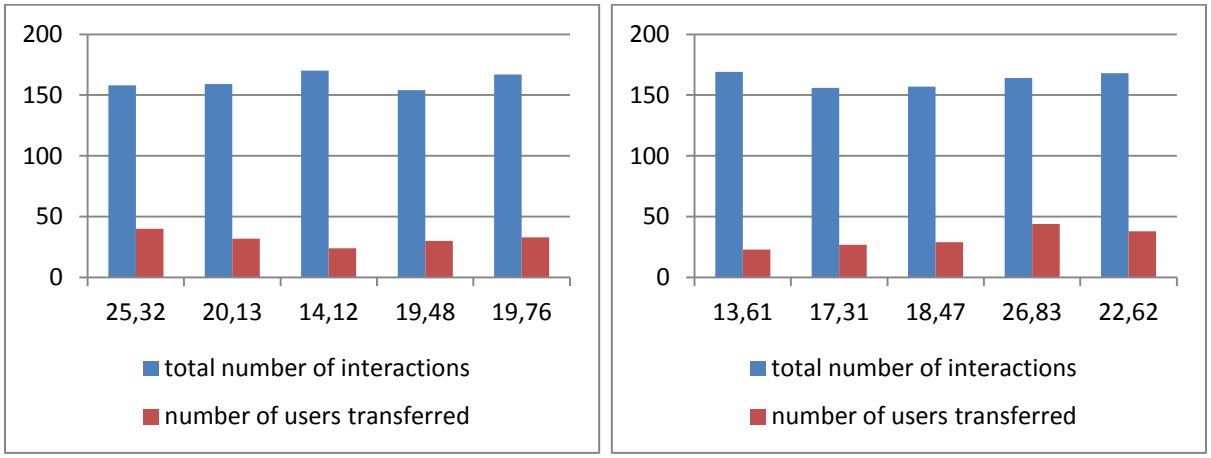


Fig. 5.3: the diagram on the left represents the results obtained when running a session for 100 users on the real system while the one on the right shows the results when using the simulation. The numbers on the bottom stand for the percentage of the total number of interactions that the number of transferred users represents.

To statistically demonstrate that the simulation rig does give very similar results to the actual one, two *T* tests will be executed, one that compares the data when the graph is not active and one for the case in which the closeness graph is used. The sample data will be represented by

the percentages that are also shown in the diagrams above. The null hypothesis will be that the two data samples are similar.

To understand the notation that will be used at this point, R_- and R_+ will represent the data sets obtained on the real system without the closeness graph and, respectively, with the closeness graph. Similarly, S_- and S_+ will stand for the simulation data without and with the closeness graph. For the mean, the M symbol will be used while for the standard deviation σ will be used.

First, for the case in which the closeness graph is not used, the two sample data that are compared are the following:

$$R_- = \{49.07,52.99,47.24,46.03,53.47\} \quad S_- = \{48.53,52.72,44.38,50.86,53.33\}$$

The number of pairs is 5 so $n = 5$, and the number of degrees of freedom is $2 * n - 2 = 8$. The results when running the *T* test on these data are:

$$M_{R_-} = 49.76 \qquad \sigma_{R_-} = 3.35 \quad M_{S_-} = 49.96 \qquad \sigma_{S_-} = 3.63$$

$$t = \frac{M_{S_-} - M_{R_-}}{\sqrt{\frac{\sigma_{R_-}^2 + \sigma_{S_-}^2}{n}}} = 0.1145$$

Looking at the *T* test distributions table (see Annex no.5) it is clear that the probability of the null hypothesis being true is significantly higher than 0.40, well above the statistical thresholds of 0.1 or 0.05. Thus, the null hypothesis stands and the two sets are statistically similar.

The same process is done for the case in which the closeness graph is used. Here, the two data sets will be:

$$R_+ = \{25.32,20.13,14.12,19.48,19.76\} \quad S_+ = \{13.61,17.31,18.47,26.83,22.62\}$$

The number of pairs is, as before, 5 so $n = 5$, and the number of degrees of freedom again 8. The results when running the *T* test on these data are:

$$M_{R_+} = 19.76 \qquad \sigma_{R_+} = 3.96 \quad M_{S_+} = 19,78 \qquad \sigma_{S_+} = 5.09$$

$$t = \frac{M_{S_+} - M_{R_+}}{\sqrt{\frac{\sigma_{R_+}^2 + \sigma_{S_+}^2}{n}}} = 0.0087$$

Again, looking at the *T* test distributions table (see Annex no.5), it can be seen that the null hypothesis stands and the two data sets are, also in this case, statistically similar. Thus, it is statistically proven that the simulation rig does offer very similar results to the actual client-

server architecture. With this demonstration finished we can advance to simulating a much higher number of online players using this rig.

Focusing back on the large scale simulation, the tests were done for one thousand, five thousand and ten thousand users. Regarding the second parameter, the number of servers in chosen to be high in order to offer a higher chance of user interactions between players stored on different machines. The last parameter is simply set by taking into account the first two and offering a reasonable margin so that all servers are not completely full when all the players are online, thus allowing migrations of players between them.

For one thousand users, six servers were used with a maximum capacity of 200 users each. For five thousand users, 10 servers with a 700 users capacity were used. Finally, for the ten thousand users test, 15 servers with a 900 users capacity were used.

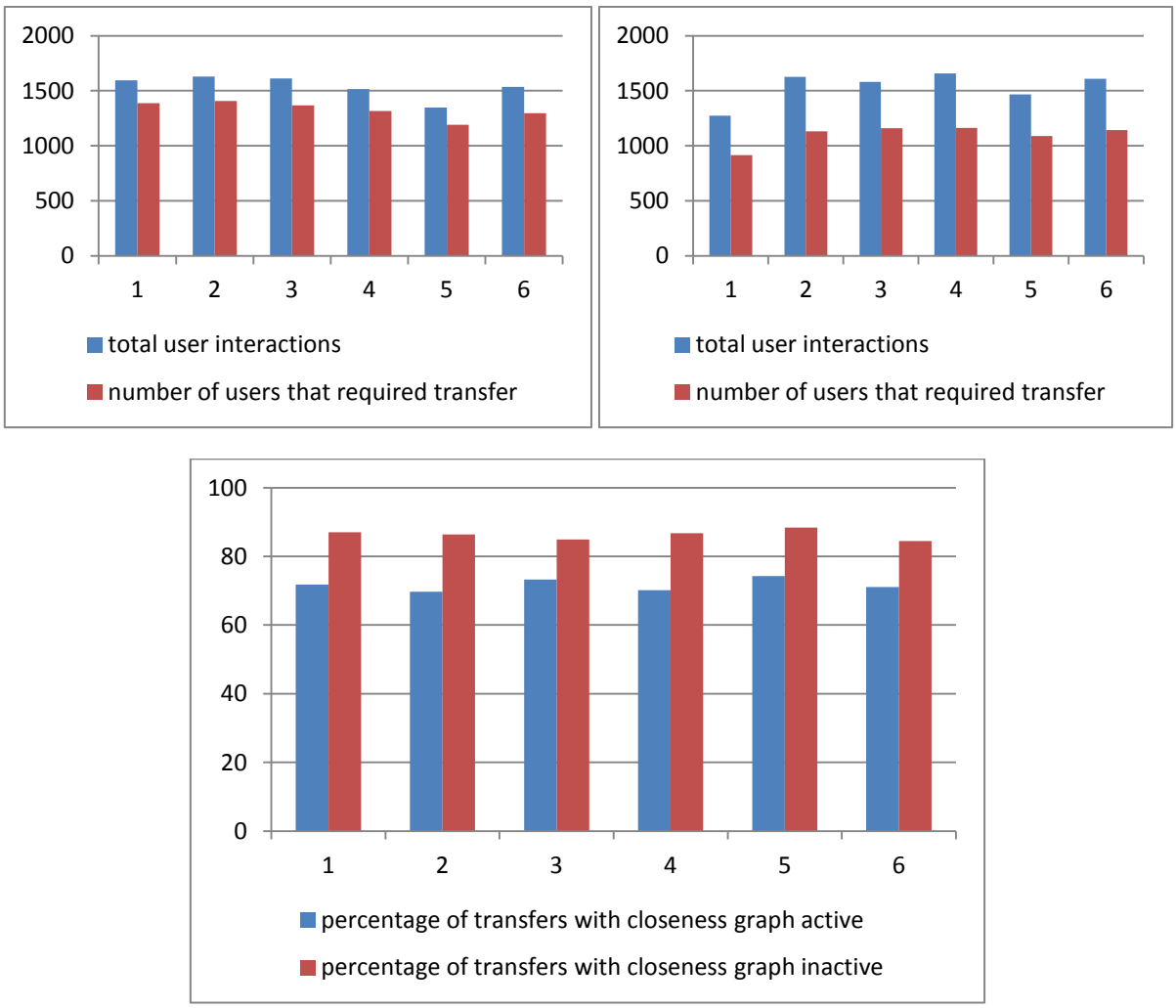


Fig. 5.4: the diagrams resulted for 1000 players. The first shows the number of transfers required when the graph data is not used while the second shows the same statistics with the closeness graph active. The last one depicts the difference in percentages between the two.

For each set of parameters a number of six simulations were ran. For each set three diagrams were made. The first depicts the number of transfers required compared to the total number of

interactions with the closeness graph inactive, while the second one shows the same data when the closeness graph is used. The third and last one puts head to head the percentages obtained in the first two, showing the amount of improvement the closeness graph does when activated.

For 1000 users using 6 database servers with a maximum capacity of 200 per server, the diagrams are seen in figure 5.4.

For 5000 users using 10 database servers with a capacity of 700 each, the resulted three diagrams are:

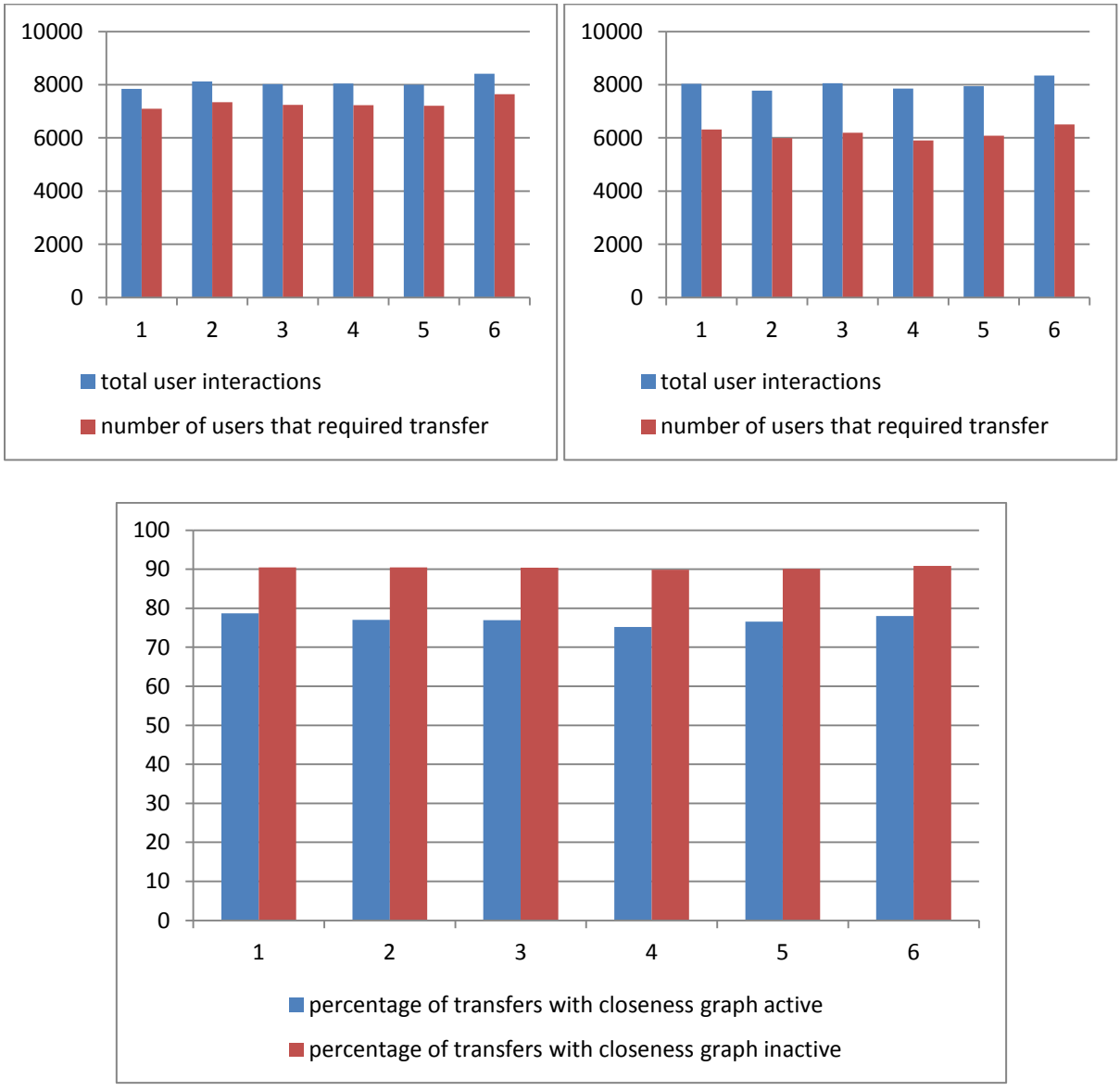


Fig. 5.5: the diagrams resulted for 5000 players. The first shows the number of transfers required when the graph data is not used while the second shows the same statistics with the closeness graph active. The last one depicts the difference in percentages between the two.

For 10,000 users using 15 database servers with a capacity of 900 each, the resulted three diagrams are:

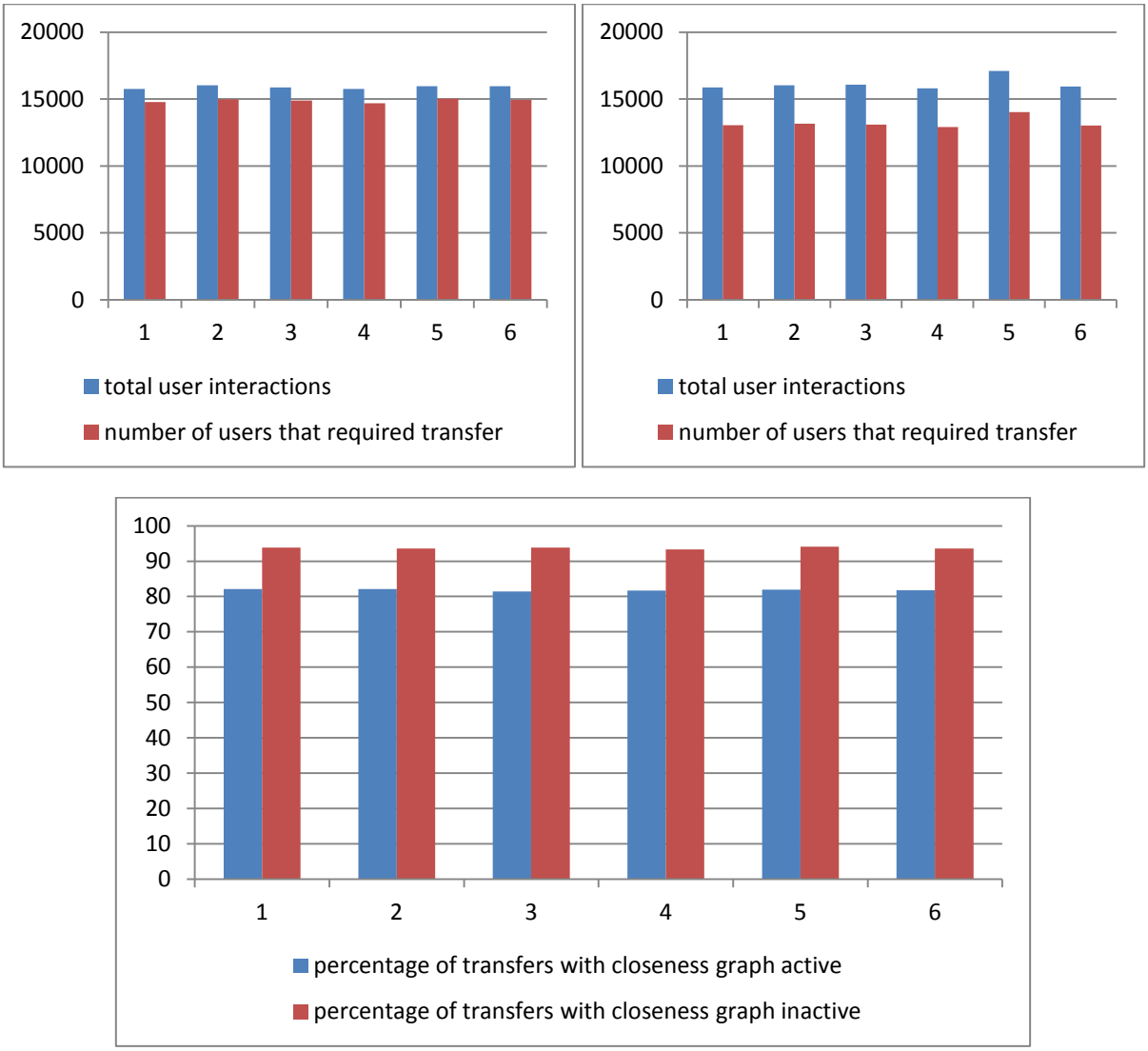


Fig. 5.6: the diagrams resulted for 10,000 players. The first shows the number of transfers required when the graph data is not used while the second shows the same statistics with the closeness graph active. The last one depicts the difference in percentages between the two.

A number of aspects can be observed from the three sets of diagrams.

Firstly, as expected, the percentage of transfers required out of the total number of interactions grows slightly with the number of servers/number of players. This is due to two factors: as the number of players grow, so do the chances of obtaining a more complex web of connections between the users; also, the changes the user interacts with a contact on his own server go down as the number of servers is greater.

Although the percentages do grow slightly, from the last diagram of each set it can be seen that the improvement in percentage brought by the closeness graph remains almost constant. This data will be analyzed more closely in the following section that will highlight the overall effects of using the closeness graph.

The three scenarios chosen here, as it can be observed, offer a certain degree of freedom when it comes to overall server capacity. This is because in a real case usage, there would always be a

certain amount of available server capacity to deal with scenarios such as unexpected spikes in user activity. However, this occupancy parameter can also play a role when it comes to the degree of improvement that the closeness graph brings when active. This can be seen also by looking at the first test using only 100 users where the occupancy was just 50% and the improvements were significantly better. To oversee this, the first scenario is taken (6 servers with 200 capacity each), and the occupancy of the system is steadily increased, starting from 100 users and ending at 1150. The results can be seen in figures 5.7 and 5.8.

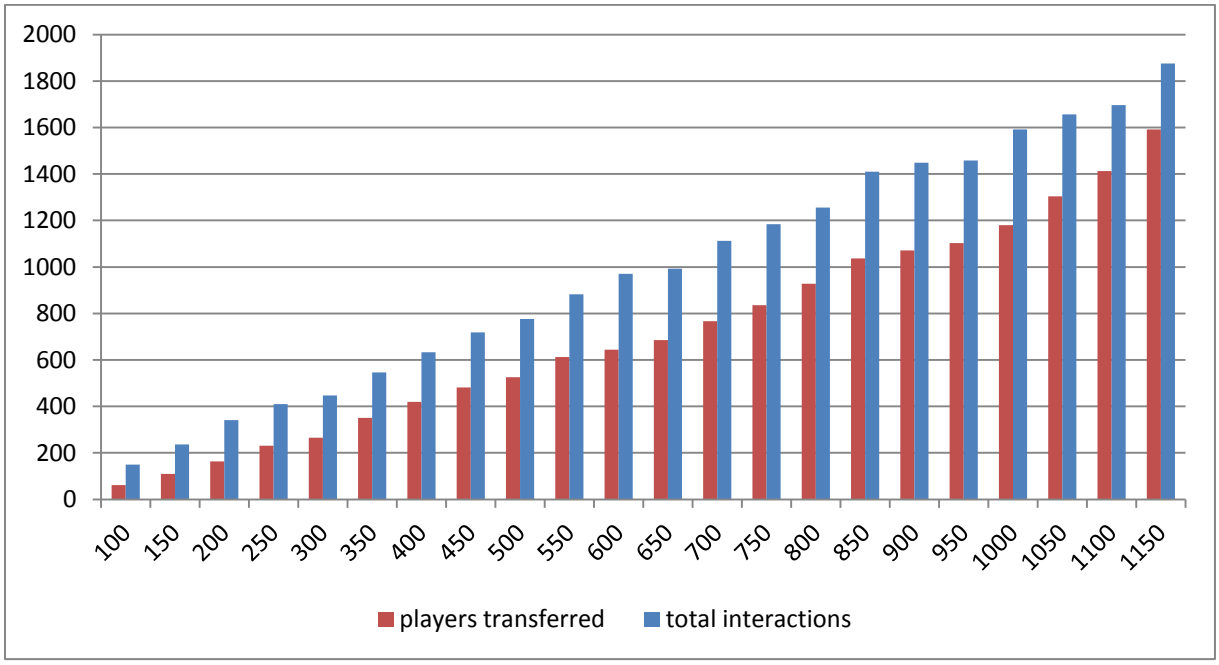


Fig. 5.7: the diagram that shows how many total interactions and player transfers were needed for a system with 6 servers of 200 user capacity when the number of users that were logged in varied from 100 to 1150.

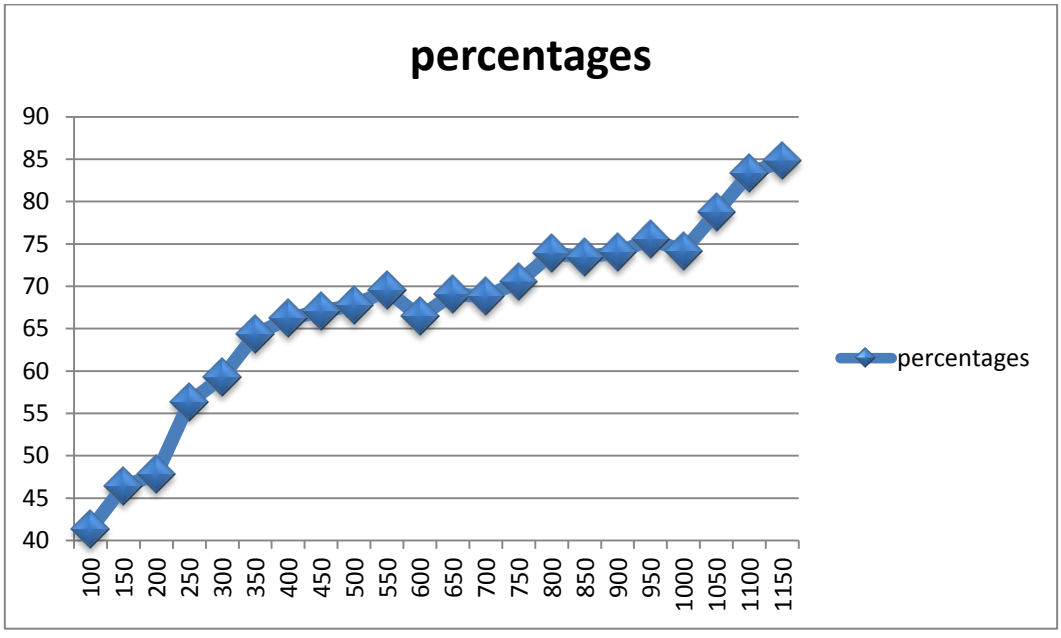


Fig. 5.8: the diagram that shows the evolution in terms of percentage of users transferred out of the total number of interactions when the number of logged in users grows from 100 to 1150.

The first figure shows two columns on each step. The blue one represents the total number of user interactions that took place while the red one represents the number of users that required transfer to a different server.

The second figure gives a clearer picture of how the percentages of data transfers that are required evolve due to the degree of occupancy.

It can be observed here how the closeness graph has the most impact when the occupancy is very low. Keep in mind that with the closeness graph inactive, the percentage of transfers is independent from the degree of occupancy as it just depends on the number of servers. For six servers, the percentage would always be around 84% when the closeness graph is not used and the players are stored at random. Thus, when the occupancy level is very low, an improvement of over 40% can be seen when using the closeness graph. However, a fast growth can be observed up to 400 online users, the equivalent to a 33% occupancy. From that point up to 1000 users, the ascending slope is less steep, registering a growth of only 10% from 400 to 1000 users. The last portion of the diagram shows how the closeness graph becomes less and less efficient once the degree of occupancy goes towards 100%. This is an expected behaviour. Due to the limited space, a high number of closeness graph decisions have to be ignored when the servers are full, placing players on servers that have enough space for them rather than on the ones containing the contacts with whom they would have a high chance of interaction.

5.5. Final results

Before the taking into account both the small scale and large scale tests in order to see the overall improvements brought to the system by the closeness graph, an in-depth analysis has to be done on the data obtained from the large scale simulation.

For the first scenario with 1000 users, the two data sets containing the percentages of players transferred out of the total number of interactions with the closeness graph active and inactive are:

$$D_+(1000) = \{71.82, 69.66, 73.26, 70.14, 74.23, 71.80\}$$

$$D_-(1000) = \{87.01, 86.36, 84.91, 85.74, 87.85, 84.18\}$$

Where D_+ represents the percentages when the closeness graph is active and D_- when the closeness graph is inactive. The means and standard deviations of the two sets are:

$$M_+(1000) = 71.81 \qquad M_-(1000) = 85.97$$

$$\sigma_+(1000) = 1.75 \qquad \sigma_-(1000) = 1.35$$

For 5000 users the same data analysis is done:

$$D_+(5000) = \{78.59, 76.56, 76.45, 75.11, 76.21, 77.78\}$$

$$D_-(5000) = \{90.68, 90.56, 90.48, 89.84, 90.18, 90.92\}$$

$$M_+(5000) = 76.78 \qquad M_-(5000) = 90.45$$

$$\sigma_+(5000) = 1.23 \qquad \sigma_-(5000) = 0.38$$

And for 10,000 users:

$$D_+(10{,}000) = \{81.03, 81.04, 80.29, 80.35, 80.57, 80.42\}$$

$$D_-(10{,}000) = \{94.98, 94.76, 95.02, 94.57, 95.61, 94.71\}$$

$$M_+(10{,}000) = 80.61 \qquad M_-(10{,}000) = 94{,}97$$

$$\sigma_+(10{,}000) = 0.34 \qquad \sigma_-(10{,}000) = 0{,}37$$

Looking at these three data sets, the average improvements registered in each case are of 14.2%, 13,7% and 14.4% respectively, which does represent a significant improvement. To statistically prove this, two big data sample are taken:

$$D_+ = D_+(1000) \cup D_+(5000) \cup D_+(10{,}000)$$

$$D_- = D_-(1000) \cup D_-(5000) \cup D_-(10{,}000)$$

Using these, a T test is run with the null hypothesis being that the two data samples are similar. As the number of samples taken here is $n = 6 * 3 = 18$, the degrees of freedom for this T test is $2 * n - 2 = 34$. For these data sets we will present the mean and standard deviation for each big group and then, finally, the value of t and the *p-value*.

$$M_+ = 76.40 \qquad M_- = 90.55$$

$$\sigma_+ = 3.89 \qquad \sigma_- = 3.83$$

$$t = \frac{M_- - M_+}{\sqrt{\frac{\sigma_+^2 + \sigma_-^2}{n}}} = 10{,}95$$

Using the t distribution table (can be seen in Annex no. 5), as the degrees of freedom are 34 and the determined value is clearly bigger than 3.601, it is clear that $p - value < 0.0005$ and can be approximated to 0.

This means that the probability of the result, assuming the null hypothesis is well below the statistical significance threshold of 0.01. Therefore, the null hypothesis is false and it is statistically proven that there is a significant difference between the two groups.

Taking into account both the disadvantages presented in section 5.3 and the advantages presented above, it can be observed that the closeness graph does have a clear positive impact on the overall architecture.

Regarding the CPU and memory usage, as said before, the master server was almost completely unhindered by the addition of the closeness graph. When the closeness graph was active the added cost in terms of percentage on average was the value of 1.3% which makes it practically negligible.

Looking at the data transfer that goes on within the system, the closeness graph clearly has a positive impact. Without it, as expected, the chances of a player needing to be transferred when an interaction occurs is of (X-1)/X where X is the number of database servers. This is due to the fact that the users are put on the servers randomly and the chances of a user that you want to interact with being stored on the same machine is 1/X.

When the closeness graph is activated, a clear reduction of the needed transfers in observed which seems to remain constant when the number of users/servers grows. Looking at the results obtained from the *T* test and comparing the two means of the two groups analyzed in that test, the overall improvement compared to the random system is of 14.15% which is significant. Taking into account the fact that for each transfer 2.2% extra data is sent across because of the closeness graph queries (see section 5.3.2), it can be said that the overall improvement regarding the data required to be transferred is of 12%.

5.6. Conclusions

This paper presented a solution for massively multiplayer online games that want to allow the player the freedom of interacting with any other online player.

The first part of the paper presented the overall structure of the client-server architecture, explaining its design and advantages. The proposed four tier architecture is an ideal solution for web-based games that can separate the client requests based on frequency, allowing for a very robust system with few active servers that is capable of supporting tens of thousands of online users. Also, this architecture has a very attractive feature: spawning and terminating servers automatically based on the overall load of the system that is monitored by a single master server.

In order to get a greater insight in how such an architecture functions, a detailed analysis was made on how the different types of servers within the architecture interact with one another, presenting each ones roles and responsibilities.

The paper then focused on the performance of the system, proposing a novel solution that keeps track of the closeness between two players based on their session information, predicting future actions of those players and thus reducing the amount of data that needs to be transferred between servers when two players stored on different machines want to interact. The closeness graph solution was compared to the default state in which players would be placed randomly at login and the initiator of an interaction would always be moved

across. Based on small-scale and large-scale tests, the proposed solution made a significant improvement (fact statistically proven by running a *T* test), reducing the overall data that needed to be transferred by 12%.

Regarding future improvements on the solutions presented in this paper, a main area of focus would be the function that updates the closeness between the players and its parameters. The set of parameters used here and the operations done were kept to a simple form intentionally, so that the data transferred to the master server at the end of a session and the CPU power required to do these calculations and update the graph do not hinder the other operations that the master server is in charge of. However, it would be a solution to use more complex data mining techniques, keep track of more parameters for each user and fetch them at the end of a session (for example keeping track of time stamps for each interaction and searching for patterns), and see in the end by what margin the prediction of future interactions can be improved by this.

Also, an analysis can be done on ways of expanding the current model. While the proposed structure is scalable and automated, there are issues that can arise when low latency due to geographical positioning is taken into account. For the current case this was not discussed because the proposed game is only focused on attracting players from Western Europe, allowing a single server location to be enough. However, if it would expand, there are possibilities of spawning the same type of structure in different geographical locations. Discussions can then be made on how players are assigned to a given server or the possibilities of two players from different regions communicating. These types of subjects are touched in [2] and [6].

6. REFERENCES

[1] Ta Nguyen Bihn Duong, Suiping Zhou "*A Dynamic Load Sharing Algorithm for Massively Multiplayer Online Games*", October 2007

[2] Carlos Eduardo Benevides Bezerra, Claudio Fernando resin Geyer "*A load balancing scheme for massively multiplayer online games*", May 2009

[3] Dr. Sowmya Ramachandran, Randy Jensen, Oscar Bascara, Tamitha Carpenter, Todd Denning "*Automated Chat Thread Analysis: Untangling the Web*", ITSEC 2010

[4] Mani Malarvannan, S. Ramaswany "*Rapid Scalability of Complex and Dynamic Web-Based Systems: Challenges and Recent Approaches to Mitigation*", IEEE International Conference on System Engineering 2010

[5] Jinesh Varia "*Amazon Web Services, Architecting for the Cloud: Best Practices*", January 2011

[6] Paul B. Beskow, Pal Halvorsen, Carsten Griwodz "*Latency Reduction in Massively Multi-player Online Games by Partial Migration of Game State*", 2008

[7] Yla Rebecca Tausczik, B.A. "*Linguistic Analysis of Computer-Mediated Communication*", August 2009

[8] Danier Cooper "*Live Data Analysis of Chat Rooms*", 2010

[9] Fengyun Lu, Simon Parkin, Graham Morgan "*Load Balancing for Massively Multiplayer Online Games*", 2007

[10] Jin Chen, Baohua Wu, Margaret Delap, Bjorn Knutsson, Honghui Lu, Cristina Amza "*Locality Aware Dynamic Load Management for Massively Multiplayer Online Games*", June 2005

[11] Evrim Acar, Deyit A. Camtepe, Mukkai S. Krishnamoorthy, Bulent Yener "*Modelling and Multiway Analysis of Chatroom Tensors*", 2006

[12] Murat Cakir, Fatos Xhafa, Nan Zhou, Gerry Stahl "*Thread-based Analysis of Patterns of Collaborative Interaction in Chat*", 2010

[13] C.M. Greenhalgh "*Awareness Management in the MASSIVE Systems*", September 1998

[14] A. Hevner, S. March, J. Park, S. Ram "*Design Science in Information Systems Research*", 2004

[15] S. Singhal, M. Zyda "*Networked Virtual Environments, Design and Implementation*", 1999

[16] S. E. Parkin, P. Andras, G. Morgan "*Managing Missed Interactions in Distributed Virtual Environments*", 2008

[17] W. Cai, P.Xavier, S. Turner, B. S. Lee "*A Scalable Architecture to Support Interactive Games on the Internet*", May 2002

[18] G. Carneiro, C. Arabe "*Load Balancing for Distributed Virtual Reality Systems*", October 1998

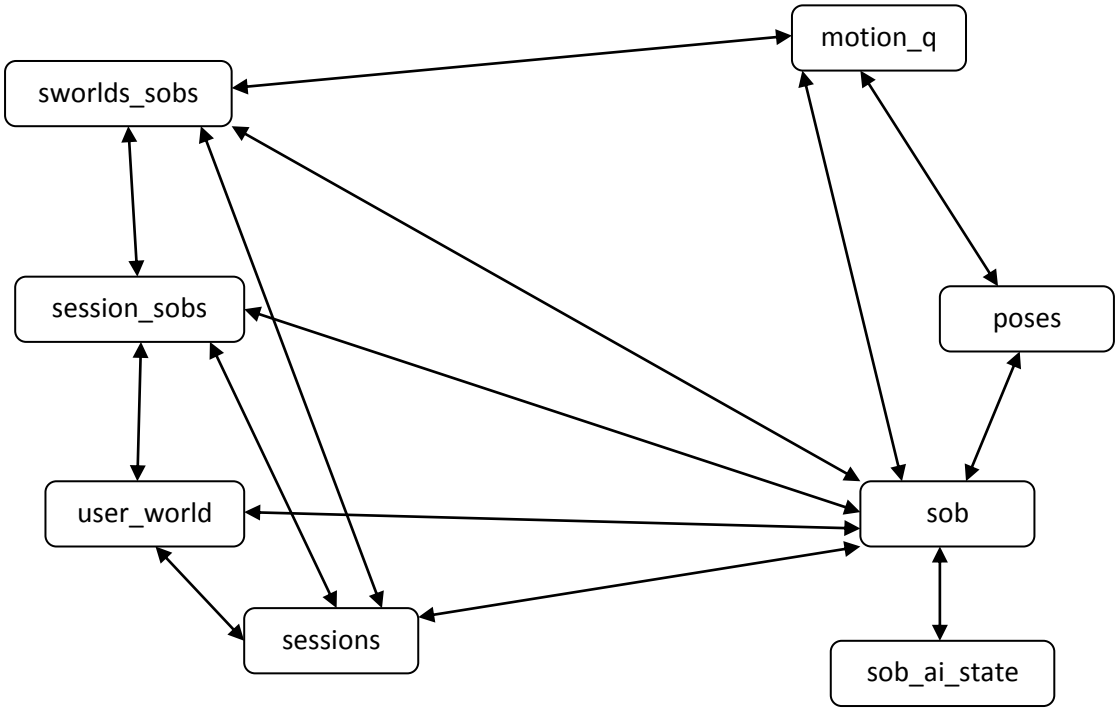
7. ANNEXES

Annex no.1: a general description of the database contents for the game *Tailplanet*

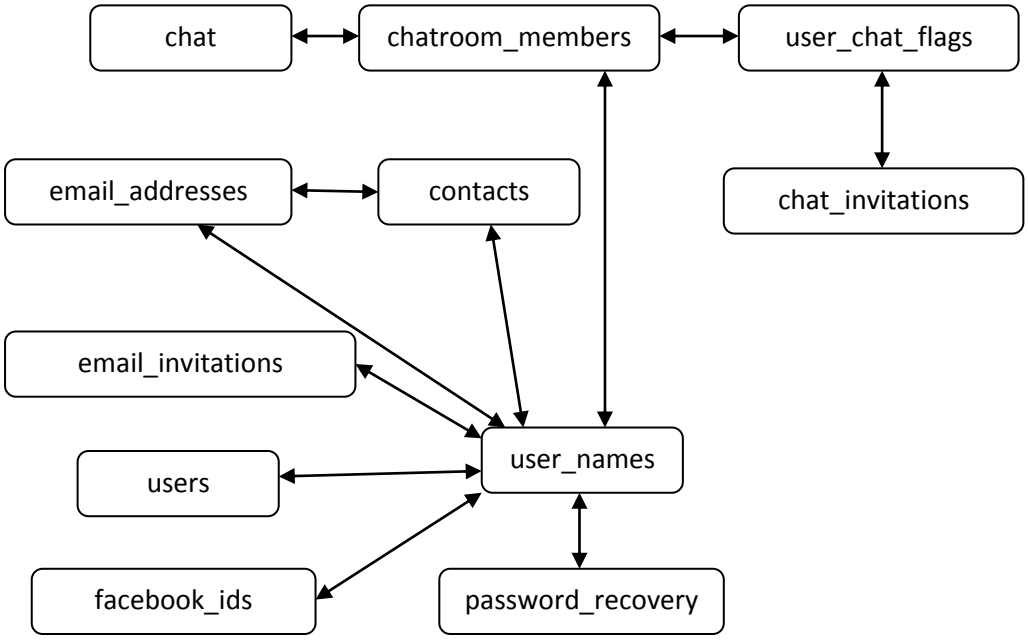
Table name	Category	Access frequency
chat	Chat related	Sparse access (~1 per 3 min)
chat_invitations	Chat related	Sparse access (~1 per 12 min)
chatroom_members	Chat related	Sparse access (~1 per 3 min)
chatroom_permission	Chat related	Sparse access (~1 per 3 min)
chatrooms	Chat related	Sparse access (~1 per 3 min)
contacts	Chat related	Sparse access (~1 per 2 min)
currency	Webshop related	Sparse access (~1 per 10 min)
email_addresses	User related	Sparse access (at login)
email_invitations	User related	Sparse access (at login)
facebook_ids	User related	Sparse access (at login)
individual_sync_clock_measurements	Webshop related	Sparse access (at login)
keyvalue	Game-play related	Sparse access (at login)
min_sync_clock_measurements	Game-play related	Frequent access (~5 per sec)
motion_q	Game-play related	Frequent access (~5 per sec)
old_sessions	Game-play related	Sparse access (at logout)
password_recovery	User related	Sparse access (<1 per session)
payment_feedback	User related	Sparse access (<1 per session)
payment_requests	User related	Sparse access (<1 per session)
poses	Game-play related	Frequent access (~5 per sec)
price	Webshop related	Sparse access (~1 per 4 min)
price_alternative	Webshop related	Sparse access (~1 per 5 min)
price_component	Webshop related	Sparse access (~1 per 5 min)
price_group	Webshop related	Sparse access (~1 per 5 min)
price_leaf	Webshop related	Sparse access (~1 per 5 min)
products	Webshop related	Sparse access (~1 per 4 min)
session_sobs	Game-play related	Frequent access (~4 per sec)

sessions	Game-play related	Frequent access (~5 per sec)
sob	Game-play related	Frequent access (~4 per sec)
sob_ai_state	Game-play related	Frequent access (~1 per sec)
sworld	Game-play related	Frequent access (~1 per sec)
sworlds_sobs	Game-play related	Frequent access (~2 per sec)
update_queue	Game-play related	Frequent access (~5 per sec)
user_chat_flags	User related	Sparse access (~1 per min)
user_names	User related	Frequent access (~5 per sec)
user_world	User related	Frequent access (~2 per sec)
users	User related	Sparse access (at login)
ws_offer	Webshop related	Sparse access (at login)
ws_offer_metadata	Webshop related	Sparse access (at login)
ws_offer_presentation	Webshop related	Sparse access (at login)
ws_offer_presentation_metadata	Webshop related	Sparse access (at login)
ws_offer_style	Webshop related	Sparse access (at login)
ws_offerprice	Webshop related	Sparse access (at login)
ws_order	Webshop related	Sparse access (at login)
ws_order_item	Webshop related	Sparse access (at login)
ws_product	Webshop related	Sparse access (at login)
ws_product_metadata	Webshop related	Sparse access (at login)
ws_product_quantity_pair	Webshop related	Sparse access (at login)
ws_product_set	Webshop related	Sparse access (at login)
ws_product_set_entries	Webshop related	Sparse access (at login)
ws_shopping_cart	Webshop related	Sparse access (at login)
ws_shopping_cart_item	Webshop related	Sparse access (at login)
ws_template	Webshop related	Sparse access (at login)
ws_template_styles	Webshop related	Sparse access (at login)

Annex no.2: the primary key field links between the game-play related tables



Annex no.3: the primary key field links between the chat related and user related tables



Annex no.4: the JSON string fields of the packets sent from client to server:

- "push" - sent across when a new animation is pushed in the motion queue of an object.
- "start" - sent across when an animation for a given object is started.
- "ttw" - it stands for "transfer to world"; it is sent across when an object needs to be transferred from one world to another.
- "msg" - this field is included in the string when a chat related event takes place; this can be a contact being added, a message being sent, a contact being accepted, or a conversation ending.
- "fb" - this field is linked to Facebook related events; because the player can login using Facebook credentials, some in-game interactions can be linked to Facebook data.
- "pr" - it stands for "payment request"; this field is included in the string when the client wants to buy premium items with real money in the game.
- "ani" - this field is filled when the client requests the animation details for a given object.
- "ai" - sent across when a persistent AI variable changes its value.
- "hotspots" - sent across when the client requests the hotspot information for an object.
- "fo" - stands for "fetch objects"; sent when the client requests to fetch the object information for a given player from a given room.
- "ws" - it stands for "webshop request".
- "fv" - sent when the value of an AI variable needs to be known by the client; used mostly at login.
- "do" - stands for "delete object".
- "so" - stands for "spawn object".
- "trigger" - used to send across persistent trigger information so it can be stored on the server.
- "dw" - used when a user no longer has any meaningful connection to a world, so that world link needs to be deleted.
- "cpr" - stands for "change password request".

Annex no.5:

TABLE of CRITICAL VALUES for STUDENT'S *t* DISTRIBUTIONS

Column headings denote probabilities (α) ***above*** tabulated values.

d.f.	0.40	0.25	0.10	0.05	0.04	0.025	0.02	0.01	0.005	0.0025	0.001	0.0005
1	0.325	1.000	3.078	6.314	7.916	12.706	15.894	31.821	63.656	127.321	318.289	636.578
2	0.289	0.816	1.886	2.920	3.320	4.303	4.849	6.965	9.925	14.089	22.328	31.600
3	0.277	0.765	1.638	2.353	2.605	3.182	3.482	4.541	5.841	7.453	10.214	12.924
4	0.271	0.741	1.533	2.132	2.333	2.776	2.999	3.747	4.604	5.598	7.173	8.610
5	0.267	0.727	1.476	2.015	2.191	2.571	2.757	3.365	4.032	4.773	5.894	6.869
6	0.265	0.718	1.440	1.943	2.104	2.447	2.612	3.143	3.707	4.317	5.208	5.959
7	0.263	0.711	1.415	1.895	2.046	2.365	2.517	2.998	3.499	4.029	4.785	5.408
8	0.262	0.706	1.397	1.860	2.004	2.306	2.449	2.896	3.355	3.833	4.501	5.041
9	0.261	0.703	1.383	1.833	1.973	2.262	2.398	2.821	3.250	3.690	4.297	4.781
10	0.260	0.700	1.372	1.812	1.948	2.228	2.359	2.764	3.169	3.581	4.144	4.587
11	0.260	0.697	1.363	1.796	1.928	2.201	2.328	2.718	3.106	3.497	4.025	4.437
12	0.259	0.695	1.356	1.782	1.912	2.179	2.303	2.681	3.055	3.428	3.930	4.318
13	0.259	0.694	1.350	1.771	1.899	2.160	2.282	2.650	3.012	3.372	3.852	4.221
14	0.258	0.692	1.345	1.761	1.887	2.145	2.264	2.624	2.977	3.326	3.787	4.140
15	0.258	0.691	1.341	1.753	1.878	2.131	2.249	2.602	2.947	3.286	3.733	4.073
16	0.258	0.690	1.337	1.746	1.869	2.120	2.235	2.583	2.921	3.252	3.686	4.015
17	0.257	0.689	1.333	1.740	1.862	2.110	2.224	2.567	2.898	3.222	3.646	3.965
18	0.257	0.688	1.330	1.734	1.855	2.101	2.214	2.552	2.878	3.197	3.610	3.922
19	0.257	0.688	1.328	1.729	1.850	2.093	2.205	2.539	2.861	3.174	3.579	3.883
20	0.257	0.687	1.325	1.725	1.844	2.086	2.197	2.528	2.845	3.153	3.552	3.850
21	0.257	0.686	1.323	1.721	1.840	2.080	2.189	2.518	2.831	3.135	3.527	3.819
22	0.256	0.686	1.321	1.717	1.835	2.074	2.183	2.508	2.819	3.119	3.505	3.792
23	0.256	0.685	1.319	1.714	1.832	2.069	2.177	2.500	2.807	3.104	3.485	3.768
24	0.256	0.685	1.318	1.711	1.828	2.064	2.172	2.492	2.797	3.091	3.467	3.745
25	0.256	0.684	1.316	1.708	1.825	2.060	2.167	2.485	2.787	3.078	3.450	3.725
26	0.256	0.684	1.315	1.706	1.822	2.056	2.162	2.479	2.779	3.067	3.435	3.707
27	0.256	0.684	1.314	1.703	1.819	2.052	2.158	2.473	2.771	3.057	3.421	3.689
28	0.256	0.683	1.313	1.701	1.817	2.048	2.154	2.467	2.763	3.047	3.408	3.674
29	0.256	0.683	1.311	1.699	1.814	2.045	2.150	2.462	2.756	3.038	3.396	3.660
30	0.256	0.683	1.310	1.697	1.812	2.042	2.147	2.457	2.750	3.030	3.385	3.646
31	0.256	0.682	1.309	1.696	1.810	2.040	2.144	2.453	2.744	3.022	3.375	3.633
32	0.255	0.682	1.309	1.694	1.808	2.037	2.141	2.449	2.738	3.015	3.365	3.622
33	0.255	0.682	1.308	1.692	1.806	2.035	2.138	2.445	2.733	3.008	3.356	3.611
34	0.255	0.682	1.307	1.691	1.805	2.032	2.136	2.441	2.728	3.002	3.348	3.601
35	0.255	0.682	1.306	1.690	1.803	2.030	2.133	2.438	2.724	2.996	3.340	3.591
36	0.255	0.681	1.306	1.688	1.802	2.028	2.131	2.434	2.719	2.990	3.333	3.582
37	0.255	0.681	1.305	1.687	1.800	2.026	2.129	2.431	2.715	2.985	3.326	3.574
38	0.255	0.681	1.304	1.686	1.799	2.024	2.127	2.429	2.712	2.980	3.319	3.566
39	0.255	0.681	1.304	1.685	1.798	2.023	2.125	2.426	2.708	2.976	3.313	3.558
40	0.255	0.681	1.303	1.684	1.796	2.021	2.123	2.423	2.704	2.971	3.307	3.551
60	0.254	0.679	1.296	1.671	1.781	2.000	2.099	2.390	2.660	2.915	3.232	3.460
80	0.254	0.678	1.292	1.664	1.773	1.990	2.088	2.374	2.639	2.887	3.195	3.416
100	0.254	0.677	1.290	1.660	1.769	1.984	2.081	2.364	2.626	2.871	3.174	3.390
120	0.254	0.677	1.289	1.658	1.766	1.980	2.076	2.358	2.617	2.860	3.160	3.373
140	0.254	0.676	1.288	1.656	1.763	1.977	2.073	2.353	2.611	2.852	3.149	3.361
160	0.254	0.676	1.287	1.654	1.762	1.975	2.071	2.350	2.607	2.847	3.142	3.352
180	0.254	0.676	1.286	1.653	1.761	1.973	2.069	2.347	2.603	2.842	3.136	3.345
200	0.254	0.676	1.286	1.653	1.760	1.972	2.067	2.345	2.601	2.838	3.131	3.340
250	0.254	0.675	1.285	1.651	1.758	1.969	2.065	2.341	2.596	2.832	3.123	3.330
inf	0.253	0.674	1.282	1.645	1.751	1.960	2.054	2.326	2.576	2.807	3.090	3.290